AD-A209 140

DARPA Status Report -- November 1988

Sun Young Hwang

1. Introduction

Simulation now requires vast amounts of cpu time. This severly limits the size of a design that can be tested thoroughly. Incremental simulation is a possible solution to these limit.

Specific synthesis and analysis tools are also proposed. The primary purpose of this project is to allow the designer to make his changes and optimizations at as high a level as possible, but allowing him to observe the ramifications of his changes at a lower level and to help guide the synthesis routines in the selection of a good design. This user guidance is necessary because of the huge design search space faced by synthesis programs.

DTIC ELECTE APR 27 1989 S H D

2. Progess

2.1. Incremental Simulation

We proposed two incremental simulation algorithms, the *incremental-in-space* and *incremental-in-time* algorithms, and implemented them in our *THOR* simulation system. These two algorithms are comparable to each other: one shows better performance for some circuits over the other, depending on the circuit structure and topology of the circuit under simulation.

2.2. Behavioral Synthesis: Hermod System

The primary purpose of this project is to allow the designer to make changes and optimizations at as high a level as possible, while allowing him to observe the ramifications of his changes at a lower level and to help guide the synthesis routines in the selection of a good design. This user guidance is necessary because of the huge design search space faced by synthesis programs. The synthesis system displays the data/control flow graph extracted from a functional model on the window screen. And register-transfer representation of a behavioral level description is displayed on the screen after optimization effort by the system.

891614

DISTRIBUTION STATEMENT A

Approved for public release; Distribution Unlimited

089 4 26 037

3. Future Work

1. *Simulation*: More experiment on a hybrid version of the incremental simulator, and install incremental THOR simulator at ~cad directory.

2. *behavioral synthesis*: Partitioning in behavioral synthesis will be investigated. In the VLSI design, it is important to partition the hardware at the early stage of design to generate good quality designs. Partitioning of algorithmic/behavioral descriptions should provide the synthesizer with the capability to explore design space effectively. Algorithmic partitioning can be achieved by splitting a procedure into multiple processes that can be executed concurrently or be pipelined. Algorithmic partitioner will be designed for the behavioral descriptions written HardwareC, the high level language for Hercules synthesis system.

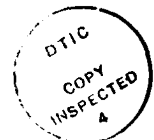
4. Publications

One paper is published and another one is accepted for possible publication around the end of 1988. They are listed below:

[1] S.Y. Hwang, T. Blank, and k. Choi,
"Fast Functional Simulation: An Incremental Approach",
IEEE Trans. on Computer-Aided Design of Integrated Systems and Circuits,
CAD-7 (7), July 1988, pp. 765-774.

[2] M. Odani, S.Y. Hwang, T. Blank, and T. Rokicki,
"The Hermod Behavioral Synthesis System", *Journal of Systems and Software*,
to appear in 1989.

Eeb:
18 papers



Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By per letter
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Suitability of Message Passing Computers for Implementing Production Systems

Anoop Gupta
Dept. of Computer Science
Stanford University
Stanford, CA 94305

Milind Tambe
Dept. of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Two important parallel architecture types are the shared-memory architectures and the message-passing architectures. In the past researchers working on the parallel implementations of production systems have focussed either on shared-memory multiprocessors or on special purpose architectures. Message-passing computers have not been studied. The main reasons have been the large message-passing latency (as large as a few milliseconds) and high message reception overheads (several hundred microseconds) exhibited by the first generation message-passing computers. These overheads are too large for the parallel implementation of production systems, where it is necessary to exploit parallelism at a very fine granularity to obtain significant speed-up (subtasks execute about 100 machine instructions). However, recent advances in interconnection network technology and processing node design have cut the network latency and message reception overhead by 2-3 orders of magnitude, making these computers much more interesting. In this paper we present techniques for mapping production systems onto message-passing computers. We show that using a concurrent distributed hash table data structure, it is possible to exploit parallelism at a very fine granularity and to obtain significant speed-ups from parallelism[1].

1. Introduction

Production systems (or rule-based systems) occupy a prominent place in the field of AI. They have been used extensively in the attempts to understand the nature of intelligence as well as to develop expert systems spanning a wide variety of applications. Production system programs, however, are computation intensive and run slowly. This slows down research and limits the utility of these systems. In this paper, we examine the suitability of message-passing computers (MPCs) for exploiting parallelism to speed-up the execution of production systems.

To obtain significant speed-up from parallelism in production systems it is necessary to exploit parallelism at a very fine granularity. For example, the average number of instructions executed by subtasks in the parallel implementation suggested in [10] is only about 100. In the past, researchers have explored the use of special-purpose architectures and shared memory multiprocessors to capture this fine-grained parallelism [10, 16, 17, 18, 11, 21]. However, the performance of MPCs for production systems has not been analyzed. Considering MPCs is important, because MPCs represent a major architectural and programming model in current use. Previously, the communication delays in the MPCs made them impossible to be used for the purpose of exploiting fine grained parallelism. However, recent developments in the implementations of MPCs [3], have reduced the communication delays and the message processing overheads by 2-3 orders of magnitude. The presence of these new generation MPCs such as the AMETEK-2010 [19] makes it interesting to consider MPCs for implementing production systems.

This paper is organized as follows. Section 2 describes the OPS5 production system and the Rete matching algorithm used in implementing it. Section 3 describes recent developments in the MPCs and presents the assumptions about their execution times which we will use in our analysis. Section 4 presents our scheme for implementing OPS5 on the MPCs. We then evaluate its performance and compare it with other parallel implementations of production systems.

2: Background

2.1. OPS5

An OPS5 [2] production system is composed of a set of *if-then* rules called productions that make up the *production memory*, and a database of temporary assertions, called the *working memory*. The individual assertions are called working memory elements (WMEs), which are lists of attribute-value pairs. Each production consists of a conjunction of condition elements (CEs) corresponding to the *if* part of the rule (the left-hand side or LHS), and a set of actions corresponding to the *then* part of the rule (the right-hand side or RHS).

The CEs in a production consist of attribute-value tests, where some attributes may contain variables as values. The attribute-value tests of a CE must all be matched by a WME for the CE to match; the variables in the condition element may match any value, but if the variable occurs in more than one CE of a production, then all occurrences of the variable must match identical values. When all the CEs of a production are matched, the production is satisfied, and an instantiation of the production (a list of WMEs that matched it), is created and entered into the *conflict set*. The production system uses a selection procedure called *conflict-resolution* to choose a production from the conflict set, which is then *fired*. When a production fires, the RHS actions associated with that production are executed. The RHS actions can add, remove or modify WMEs, or perform I/O.

The production system is executed by an interpreter that repeatedly cycles through three steps: *match*, *conflict-resolution*, and *act*. The matching procedure determines the set of satisfied

[1]This research was sponsored by Encore Computer Corporation, Digital Equipment Corporation and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Air Force Avionics Laboratory. Anoop Gupta is supported by DARPA contract N00014-87-K-0828 and an award from the Digital Equipment Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Encore Computer Corporation, Digital Equipment Corporation and the Defense Advanced Research Projects Agency or the US Government.

In Proceedings of AAAI-88

productions, the conflict-resolution procedure selects the highest priority instantiation, and the act procedure executes its RHS.

2.2. Rete

Rete [7] is a highly efficient match algorithm that is also suitable for parallel implementations [9]. Rete gains its efficiency from two optimizations. First, it exploits the fact that only a small fraction of working memory changes each cycle by storing results of match from previous cycles and using them in subsequent cycles. Second, it exploits the commonality between CEs of productions, to reduce the number of tests performed.

Rete uses a special kind of a data-flow network compiled from the LHSs of productions to perform match. The network is generated at compile time, before the production system is actually run. The entities that flow in this network are called *tokens*, which consist of a *tag*, a *list of WME time-tags*, and a *list of variable bindings*. The tag is either a + or a – indicating the addition or deletion of a WME. The list of WME time-tags identifies the data elements matching a subsequence of CEs in the production. The list of variable bindings associated with a token corresponds to the bindings created for variables in those CEs that the system is trying to match or has already matched.

There are primarily three types of *nodes* in the network which use the tokens described above to perform match:

1. *Constant-test nodes*: These are used to test the constant-value attributes of the CEs and always appear in the top part of the network. They take less than 10% of the time spent in Match.

2. *Memory nodes*: These store the results of the match phase from previous cycles as state. This state consists of a *list* of the tokens that match a part of the LHS of the associated production. This way only changes made to the working memory by the most recent production firing have to be processed every cycle.

3. *Two-input nodes*: These test for joint satisfaction of CEs in the LHS of a production. Both inputs of a two-input node come from memory nodes. When a token arrives from the *left memory*, i.e., on the left input of a two-input node, it is compared to each token stored in the *right memory*. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action occurs when a token arrives from the right memory. We refer to such an action as a *node-activation*.

Figure 2-1 shows the Rete net for a production named P1.

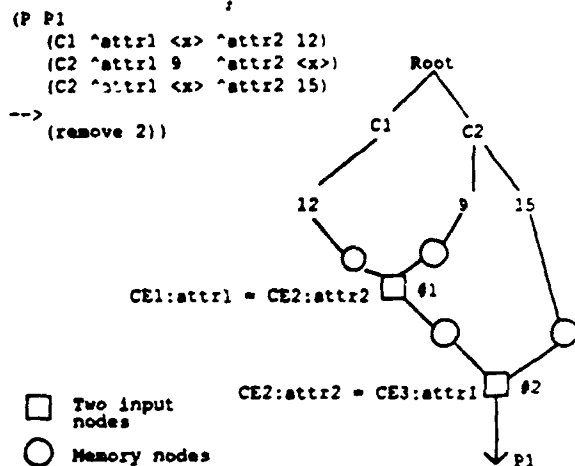


Figure 2-1: The Rete network.

3. Message-Passing Computers and Assumptions

MPCs are MIMD computers based on the programming model of concurrent processes communicating by *message passing*. There is no global *shared* memory and hence communication between the concurrent processes is explicit as in Hoare's CSP [12], though not necessarily synchronous. The early MPCs such as the Cosmic Cube [20] had a high network latency of about ~2 millisecond (ms) and a high overhead of message handling of about ~300 microseconds (μs). As a result, it was impossible to exploit parallelism at the fine granularity of 50-100 μs as is necessary in production systems.

Recent developments in MPCs such as *worm-hole routing* [4] have reduced the network latencies to 2-3 μs and the use of special processors such as the MDP (Message Driven Processor) [5] can potentially reduce the message reception overhead by an order of magnitude. With today's VLSI technology, it is possible to construct MPCs with thousands of processing nodes and hundreds of megabytes of memory [3]. Thus very fine grain parallelism can now be exploited easily with the MPCs.

This raises the issue of whether production systems can be implemented efficiently on the MPCs to give good speedups, which we analyze in detail in this paper. For the purpose of this analysis, we assume a *32-ary 2-cube architecture* (1024 nodes), with a *4 MIPS* processor at each node similar to the MDP. The various times that required for our analysis are as follows. The latency of wormhole routing is given by

$$T_{wh} = T_C(D + L/W)$$

Where —

T_C	Channel Delay, assumed to be 50 nanoseconds (ns), as in [3].
W	Channel Width, assumed to be 16 bits.
L	Length of the message in bits.
D	Distance or number of hops traveled by the message. If two processing nodes are selected at random in a k-ary n-cube, then number of hops is $n*(k^2 - 1)/3k = 22$ for our 32-ary 2-cube.

We assume that the MDP is driven by a 100 ns clock and that the time to execute a send (broadcast) command is

$$T_s = (5 + N*Q) \text{ clock cycles.}$$

where a message of Q words is to be sent to N sites [5]. The overhead of receiving messages is assumed insignificant [5]. Thus there are two delays associated with a message: T_s in transmission, T_{wh} in its communication.

4. Mapping Rete on the MPC

In this section we describe our mapping of Rete on the MPCs. We draw heavily from our previous work with the *PSM* implementations of production systems on shared-memory multiprocessors [9, 10, 21].

One possible scheme for implementing OPS5 on the MPCs arises

from viewing Rete in an *object-oriented* manner, where the nodes of Rete are objects and tokens are messages. This scheme maps a single object (node) of Rete onto a single processor of the MPC. However, there are two serious problems: (1) The mapping requires one processor per node of the Rete net, and the processor utilization of such a scheme is expected to be very low; (2) Often, the processing of a WME change results in multiple activations of the same Rete node, which in the above mapping would be processed sequentially on the same PE, thus causing that PE to be a bottleneck.

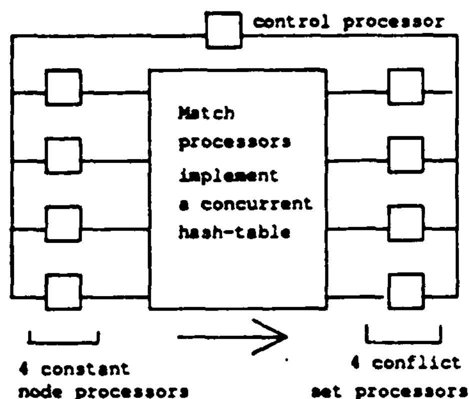


Figure 4-1: A high level view of the Mapping on the MPCs.

To overcome the limitations of above mapping, we propose an alternative mapping, a high-level picture of which is shown in Figure 4-1. At the heart of this mapping is a *concurrent distributed hash-table* [6] data structure that enables fine-grain exploitation of concurrency. The details are described later in this section. As shown in the figure 4-1, the parallel mapping consists of 1 *control processor*, 4 *constant-node processors*, 4 *conflict set processors*, and the rest are *match processors*. The constant-test nodes of the Rete net are divided into 4 parts and assigned to the constant-node processors. The match processors perform the function of the rest of the Rete net. The conflict-set processors perform conflict-resolution on the instantiations sent to them. Subsequently, they send the best instantiation to the control processor. The control processor is responsible for performing conflict-resolution among the best instantiations, evaluating the RHS and performing other functions of the interpreter.

As mentioned in Section 2.2, most of the time in match is spent processing two-input node activations. Hashing the contents of the associated memory nodes, instead of storing them in linear lists, reduces the number of comparisons performed during a node-activation and thus improves the performance of Rete. One hash table is used for all left memory nodes in the network and the other for all right memory nodes. The hash function that is applied to the tokens takes into account (1) the variable bindings tested for equality at the two-input node, and (2) the unique node-identifier of the destination two-input node. This permits quick detection of the tokens that are likely to pass the equal variable tests.

In our mapping, to allow the parallel processing of (1) tokens destined for the *same* two-input node and (2) tokens destined for different two-input nodes, the hash tables buckets storing the tokens are distributed among the PEs of the processor array. In particular, a small number of corresponding buckets from the left and right hash tables are assigned to each *processor pair* in the array -- the left-buckets to the left processor and the right buckets to the right processor. (Note that when processing a node activation, the left and right buckets at only one index need to be accessed.) This mapping is pictorially depicted in Figure 4-2. There is one restriction on the communication with the processor-pair — it can only be done through the *left-processor*. Allowing communication with both left and right processors can result in creation of duplicate tokens leading to incorrect behavior, and it does not gain as much in concurrency.

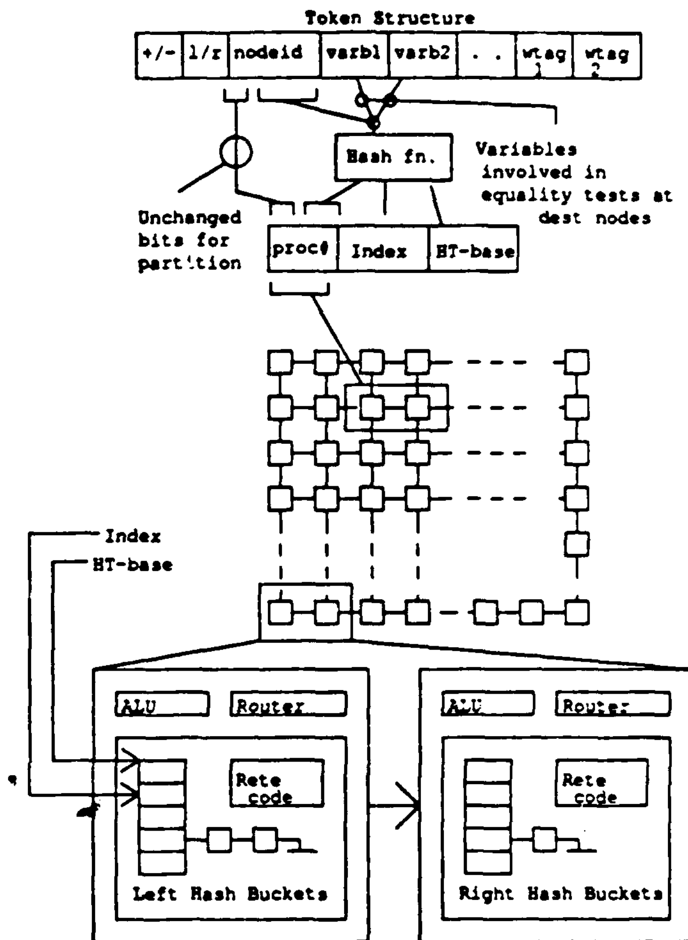


Figure 4-2: The detailed mapping.

A processor-pair together performs the activity of a single *node activation*. Consider the case when a token corresponding to the left-activation of a two-input node arrives at a processor-pair. The left processor immediately transmits the token to the right processor. The left processor then copies the token into a data-structure and adds it to the appropriate hash-table bucket. Meanwhile, the right processor compares the token with contents of the appropriate right bucket to generate tokens required for successor node activations. The right-processor then calculates the hash value for the newly created tokens, and sends each token to the processor pair which owns the buckets that it hashes to. The activities performed by the individual processors of the processor pair are called *micro-tasks*, and all the micro-tasks on the various processor pairs are performed in parallel.

The performance of this scheme depends on the discriminability of hashing. Two observations can be made in this respect:

1. Hashing is based on equality tests in CEs and 90% of the tests at two input nodes are equality tests [9].

2. The locks on the hash tables in the PSM implementations have not been seen to be bottlenecks [10, 21].

Thus hashing is not expected to be a problem in general. However, in certain production systems, a large number of two-input nodes do not have any tests. For such nodes, various schemes as proposed in [1], can be used to introduce discriminability into the tokens generated. Furthermore, when the compiler does come across nodes which cannot be hashed, it can assign a larger number of processors for that pair of buckets, (since all the tokens would end up in a single pair of buckets) thus breaking up the processing.

The code for the Rete net is to be encoded in the OPS83 [8] software technology. With this encoding, large OPS5 programs (with ≈ 1000 productions) require about 1-2 Mbytes of memory — a problem, since each MPC processor has only 10-20 kbytes of local memory. We therefore use two strategies to save space:

1. Partition the nodes of Rete such that each processor evaluates nodes from only one partition. This partitioning is easily achieved if the hash function preserves some bits from the node-id. To avoid contention, nodes belong to a single production are put into different partitions.

2. One cause of the large memory requirement is the in-line expansion of procedures. We can instead encode the two-input nodes into structures of 14 bytes, indexed by the node-id. A small performance penalty of loading the required information into registers is then paid in the beginning of the computation.

The system's overall operation is as follows:

1. The control processor evaluates a WME change and transmits it to the constant node processors.

2. The constant node processors match the WME with the constants in the CEs. The result of this match is tokens that have bindings for the variables in matched CEs. These tokens represent individual node activations and are sent to appropriate processor pairs.

3. The following steps are then repeated by the processor-pairs until completion of match:

    - Split the *node-activation* into *micro-tasks* and perform them in parallel.

    - Count the number of successor tokens generated due to this token; if no successors are generated, then send an acknowledgement (ack) message to this processor pair's activator.

    - Accept ack messages from the successors. If accounted for all successors of a token, send an ack message to the activator.

Detecting termination in a distributed system is a complex problem in itself [15]. The ack messages provide an easy and reasonably efficient method of informing the conflict-set processors about the completion of the match. Thus after the processing of the last activation in the current match cycle, a single stream of ack messages flows back, finally to the control processor, which then informs the conflict set processors that the match is completed.

5. Performance Analysis

We now evaluate the MPC implementation using the measurements on the Rete net from [9].[2] The point of the analysis is to establish that the MPCs will provide good speedups compared to other previously proposed parallel implementations, rather than to estimate the exact performance that will be obtained on a real machine.

One of the important numbers for this analysis is the time spent in the processing of one node activation. Using that, we can estimate the time for a micro-task. A node activation is identical to a *task* on the PSM, which takes 200 μs on a 1 MIPS processor [10]. Measurements of the number of instructions executed indicate that about 50% of that time is spent in updating the hash bucket and 50% in performing tests with tokens in opposite memory. We therefore assume that on our 4 MIPS processor, performing a micro-task will take about 25 μs, which is 200 μs * 1/4 (due to processor speed) * 0.5 (due to partitioning of the node-activation into micro-tasks).

Since the processor-pairs communicate via tokens, we also need to calculate the overhead of a token message. The length of a token-message is dependent on the number of variable bindings and the number of WME timetags carried by the token. There are on average four variable bindings per production [9]. The number of WME timetags is dependent on the number of CEs in a production. Assuming the number of CEs to be ($M = 5$) for the moment, we use the token-structure in Figure 4-2 to estimate 42 bytes of information per token. The overhead of sending the token message will therefore be equal to $T_s = (5 + Q * N)$ clock cycles, with $Q = 42/4$ words and $N = 1$ processor (see section 3). Substituting, we get $T_s = 1.6$ μs. The communication delay T_{wh} is given by $T_C(D + L/W)$. This communication will be between a random pairs of processors. Therefore, $D = 22$. We have assumed T_C to be 50ns and W to be 16. Our L is 42 * 8 = 336 bits. Substituting, we get $T_{wh} = 2.2$ μs. The total delay will be therefore 1.6 + 2.2 = 3.8 μs per token message between processor-pairs.

We can now estimate the cost of one match cycle. The steps below correspond to the algorithm in the previous section.

Step 1: The WME changes are transmitted to the 4 constant-node processors. The cost of addition of a WME is as follows: The average WME consists of 24 attribute value pairs, which can be encoded in 24 bytes for attributes + 24 words for the values = 30 words. Broadcasting this WME takes $T_s = (5 + 30 \text{ words} * 4 \text{ processors})$ clock cycles i.e., 12.5 μs.

For the communication delay, T_{wh}, $D = 1$ since the constant node processors are one hop away from the control processor. The value of L is 30 words * 32 bits/word = 960 bits; $W = 16$ and the value of T_C is fixed at 50. Substituting, we get $T_{wh} = 3.1$ μs. Thus the total time spent in communication during WME-addition is 15.6 μs.

For deleting a WME, only the timetag of the WME to be deleted is passed on to the constant-node processors. Calculating T_s and T_{wh} in a similar fashion, we get the total time spent in delete to be 1.1 μs. There is an average of 2.5 WME changes per cycle. Assuming equal proportions of adds and deletes, the cost of the first step is 1.25(1.1 + 15.6) ≈ 21 μs.

[2] We do not analyze the conflict-resolution and action parts of the match since these take less than 10% of the time in a serial implementation. Since we have divided up the conflict set and pipelined the action part with the match, these should take even less time than that. In case they do become bottlenecks, various schemes discussed in [9] can be used to reduce their overheads.

Step 2: The constant tests are now evaluated. Assuming that the constant tests are implemented via hashing, there are 20 constant-node activations per WME change [9]. On average, each partition will have 5 activations per WME change. Thus about (5 * 2 / 4 MIPS) = 2.5 μs are spent in matching the constant nodes. A token structure is then generated and bindings are created for the variable(s) of the CEs which passed the tests. Measurements [9] show that there will be about 5-7 such tokens generated per WME change, which we assume to take 20 μs. This whole operation of processing a WME-change by a constant-node processor is therefore estimated to take about 22.5 μs. For the 2.5 WME-changes, (22.5 * 2.5) = 56 μs will be spent in processing the constant nodes and generating the initial tokens in a cycle. The generation of these tokens is pipelined with sending the tokens to the match processors.

Step 3: The processor-pairs perform the rest of the match. The node-activation typically go to different processor-pairs, and are processed in parallel. Therefore, the total time to finish the match is determined by the longest chain of dependent node-activations, since the micro-tasks in the chain have to be processed sequentially. On an average, the chain will be generated after 50% of the initial tokens in a cycle have been generated. A constant-node processor takes 56 μs to generate all the initial tokens; therefore, we assume that the initial token generating the long chain will be created after 28 μs. Including the constant-node processors, let the longest chain be of length M = 5.

When a token arrives at the left processor, it is immediately transmitted to the right processor. For this transmission, T_s is still 1.6 μs. But, T_{wh} = 50(1 channel + 42 * 8/16) = 1.1 μs. Thus, after a token arrives at the left processor, it will take 1.6 + 1.1 = 2.7 μs to reach the right processor. The right processor will take 25 μs to finish the micro-task. It will then take 3.8 μs for the successor token to reach its destination. Thus, the time to complete a micro-task is 25 + 2.7 + 3.8 = 31.5 μs. A chain of length 5 will therefore take 31.5 * 4 + 28 μs (due to the constant nodes) = 154 μs. (Similar analysis could be done if the successors are generated by the left processor).

The ack messages are propagated back through the node activation chain, after the last activation is processed. It is 1 word of information and so we estimate T_{wh} = 1.2 μs and T_s = 0.6 μs. Assuming that the ack is processed in 1 μs, the time spent in the chain of ack messages is (M = 5) * (1 + 1.2 + 0.6) = 14.0 μs. Adding all the numbers together, we get the time for MPC to match to be approximately 154 + 14 + 21 = 189 μs.

A production system generates 200 micro-tasks on an average/cycle, and therefore a uniprocessor will take 200 * 25 = 5000 μs per cycle. Using this we get about *26 fold* speedup for the above system with the longest chain of M = 5. This is ~60% of the maximum parallelism exploitable on an ideal multi-processor at this granularity. Our calculations show that the speedups is ~14 fold if M = 10 and ~9 fold if M = 15. Again, this is ~60% of the maximum available parallelism. This is comparable with the estimate of 60% exploitable parallelism in shared memory multiprocessors at the node-activation level [9]. This coarser grain node-activation level parallelism can be exploited on the MPCs by allocating both the left and right buckets to one processor. Our calculations show that the micro-task based scheme is capable of exploiting 1.5 time more speedup than a scheme to exploit the node-activation level parallelism.

6. Discussion

Comparing the MPC implementation to a shared memory multiprocessor implementation, we see that the principle advantage of the MPC implementation is the *absence* of a centralized task-scheduler, which can be a potential bottleneck. As shown in [9], in shared-memory implementations, a slow scheduler forces saturation speedup with relatively small number of processors, irrespective of the inherent parallelism in the system. However, the MPC implementation suffers from a static partitioning of the hash tables. It is possible that distinct tokens, which could potentially be processed in parallel, are processed sequentially because they hash to the same processor pair. Such a possibility does not arise in the shared-memory implementation, since the size of the hash table is independent of the number of processors.

Another tradeoff to be considered is between processor utilization and the number of processors. With a higher number of processors, the processor utilization will be low, but the message contention in the network will be reduced. As the number of processors is reduced, processor utilization will be improved; but again, this will also increase the hash table contention. Thus there are some interesting tradeoffs involved in moving towards the MPCs.

A mapping similar to one proposed in this paper has been used to implement production systems on the simulator for *Nectar*, a network computer architecture with low message passing latencies [13]. These simulations show that good speedups can be obtained by implementing production systems on MPCs with low latencies [22]. The simulations also indicate that the constant node processors can quickly become bottlenecks if the initial tokens are not generated and sent fast enough. In our current implementation, we have hashed the constant nodes to take care of such a possibility. If the constant node processors continue to be bottlenecks inspite of this, then schemes proposed in [22] can be used to remove them.

Finally, we would like to reiterate the importance of mapping production systems on MPCs. Current production systems offer limited (10-20 fold) parallelism [9]. We have shown that the MPCs are capable of exploiting this limited parallelism. However, production systems with more inherent parallelism are getting designed [14]. In such production systems, the parallelism is expected to be much higher [21]. For such production systems, it becomes necessary to analyze easily scalable architectures such as the MPCs for their implementations.

7. Summary

Recent advances in interconnection network technology and processing node design have reduced the latency and message handling overheads in MPCs to a few microseconds. In this paper we addressed the issue of efficiently implementing production systems on these new-generation MPCs. We conclude that it is indeed quite possible to implement production systems efficiently on MPCs. At a high level, our mapping corresponds to an object oriented system, with Rete network nodes passing tokens to each other using messages. At a lower level, however, instead of mapping each Rete node onto a single processor, the state and the code associated with a node are distributed among the multiple processors. The main data structure that we exploit in our mapping is a concurrent distributed hash-table that not only allows activations of distinct Rete nodes to be processed in parallel, but also allows multiple activations of the same node to be processed in parallel. A single node activation is further split into two micro-tasks that are processed in parallel, resulting in very high expected performance.

Acknowledgements

We would like to thank H. T. Kung for questioning our assumptions about shared memory architectures. We would like to thank Charles Forgy, Brian Milnes, Allen Newell and Peter Steenkiste for many useful comments on earlier drafts of this paper. We would also like to thank Kathy Swedlow for technical editing.

References

[1] Acharya, A., Kalp, D., Tambe, M.
Cross Products and Long Chains.
Technical Report, Carnegie Mellon University Computer Science Department, In preparation.

[2] Brownston, L., Farrell, R., Kant, E., Martin, N.
Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.
Addison-Wesley, 1985.

[3] Dally, W. J.
Directions in Concurrent Computing.
In *Proceedings of ICCD-86.* October, 1986.

[4] Dally, W. J.
Wire Efficient VLSI Multiprocessor Communication Networks.
In *Stanford Conference on Advanced Research in VLSI.* 1987.

[5] Dally, W. J., Chao, L., Chien, A., Hassoun, S., Horwat, W., Kaplan, J., Song, P., Totty, B., Wills, S.
Architecture of a Message-Driven Processor.
In *International Symposium on Computer Architecture.* 1987.

[6] Dally, W. J.
A VLSI Architecure for Concurrent Data Structures.
PhD thesis, California Institute of Technology, 1987.

[7] Forgy, C. L.
Rete: A fast algorithm for many pattern/many object pattern match problem.
Artificial Intelligence 19:17-37, 1982.

[8] Forgy, C. L.
The OPS83 Report.
Technical Report 84-133, Carnegie Mellon University Computer Science Department, May, 1984.

[9] Gupta, A.
Parallelism in Production Systems.
PhD thesis, Carnegie Mellon University, March, 1986.

[10] Gupta, A., Forgy, C. L., Kalp, D., Newell, A., Tambe, M. S.
Parallel OPS5 on the Encore Multimax.
In *Proceedings of the International Conference on Parallel Processing.* August, 1988.

[11] Hillyer, B. K. and Shaw, D. E.
Execution of OPS5 production systems on a Massively Parallel Machine.
Journal of Parallel and Distributed Processing 3:236-268, 1986.

[12] Hoare, C. A. R.
Communicating sequential processes.
Communications of ACM 21(8):666-677, 1978.

[13] Kung, H. T., Steenkiste, P., Bitz, F.
The Nectar computer architecture.
Personal Communication.

[14] Laird, J. E., Newell, A., & Rosenbloom, P. S.
Soar: An architecture for general intelligence.
Artificial Intelligence 33:1-64, 1987.

[15] Mattern, F.
Algorithms for distributed termination detection.
Journal of Distributed Computing 2:161-175, 1987.

[16] Miranker, D. P.
TREAT: A New and Efficient Algorithm for AI Production Systems.
PhD thesis, Columbia University, 1987.

[17] Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
PhD thesis, Carnegie-Mellon University, March, 1987.

[18] Schreiner, F. , Zimmerman, G.
Pesa-1 — A Parallel Architecture for Production Systems.
In *International Conference on Parallel Processing.* 1987.

[19] Seitz, C., Athas, W., Flaig, C., Martin, A., Seizovic, J., Steele, C., Su, W.
The Architecture and Programming of the AMETEK 2010 Multicomputer.
In *Hypercube concurrent computer and applications.* 1988.

[20] Sietz, C. L.
The Cosmic Cube.
Communications of ACM C-33(12), 1984.

[21] Tambe, M. S., Kalp, D., Gupta, A., Forgy, C. L., Milnes, B., Newell, A.
Soar-PSM/E: Investigating match parallelism in a learning production system.
In *Proceedings of the PPEALS-88.* 1988.

[22] Tambe, M., Bitz, F., Steenkiste, P.
Production Systems on the Nectar: Simulation Results and Analysis.
Technical Report, Carnegie Mellon University Computer Science Department, In preparation.

Temperature Measurement of Simulated Annealing Placements

Jonathan Rose
Computer Systems Laboratory, Stanford University, Stanford CA

Wolfgang Klebsch and Juergen Wolf
Siemens AG, Munich, Federal Republic of Germany

Abstract

One way to reduce the computational requirements of Simulated Annealing placement algorithms is to use a faster heuristic to replace the early phase of Simulated Annealing. Such systems need to know a starting temperature for the annealing phase that makes the best use of the existing structure, yet does an appropriate amount of improvement. This paper presents a method for *measuring* the temperature of an existing placement based on analysis of the probability distribution of the *change* in cost function. Using this view a new definition of equilibrium is given and the *equilibrium temperature* of a placement is defined. Temperatures of placements produced both by a Simulated Annealing and a Min-Cut placement algorithm are measured.

1 Introduction

The success of the Simulated Annealing algorithm for automatic placement [Sech85] has been hindered by its excessive computational requirements. Recent work on standard cell placement algorithms [Rose86, Grov87, Rose88] has suggested alleviating this by using a two-stage approach: begin with a good heuristic such as the Min-Cut algorithm [Dunl85] and follow it with a Simulated Annealing-based approach for more fine optimization. This allows a tradeoff between execution time and quality. A critical issue in this approach is to decide the starting temperature of the Simulated Annealing phase. If it is too high, then some of the structure created by the first phase will be destroyed and unnecessary extra work will have to be done in the Simulated Annealing phase. If the temperature is too low then solution quality is lost, similar to the case of a quenching cooling schedule [Whit84].

This paper presents a technique for *measuring* the temperature of a placement for use in such two-stage systems. To do so, we present a new view of Simulated Annealing state different from those articulated in [Rome84, Whit84, Aart85]. The principal difference is that we look at probability distributions of the *change* in cost function of a Simulated Annealing state, rather than the absolute cost function. Using this view we give a definition of equilibrium from which follows the notion of the *equilibrium temperature* of a placement.

From this we develop a measure that quantifies the nearness of a Simulated Annealing placement to equilibrium and give experimental evidence of its ability to detect equilibrium. This leads to a method for measuring the equilibrium temperature of a placement, and we show that it works both for placements produced by a Simulated Annealing and a Min-Cut placement algorithm.

This work was supported by an NSERC Post-Doctoral Fellowship and DARPA Contract #N00014-87-K-0828.

The determination of starting temperature for Simulated Annealing in two-stage systems has not been seriously addressed before. Both [Rose86,Rose88] and [Grov87] introduce the question but avoid answering it by choosing a starting temperature based simply on prior experience.

2 Definition of Equilibrium and Temperature

In previous discussions of cooling schedules and convergence [Rome84, Whit84, Aart85], the Simulated Annealing *state* has been represented either as the probability distribution of the absolute cost $P(C)$, or the set of transition probabilities from every state i to every other state j, T_{ij}. We suggest a different view that gives more information about equilibrium dynamics: the probability distribution of the *change* in cost function *from* the current state. $P(\Delta C)$ is the probability that a given state under a Simulated Annealing process with a particular generation function [Rome84] will generate a move with a change in cost function of ΔC. $P(\Delta C)$ varies with temperature (T) and as moves are made.

We can use this view to give a different perspective on the equilibrium of a Simulated Annealing process. Since at equilibrium the absolute cost function no longer changes, this implies that the expected value of the *change* in cost function is zero:

$$E(\Delta C) = 0 \tag{1}$$

An expression for $E(\Delta C)$ can be formed assuming that $P(\Delta C)$ is known:

$$E(\Delta C) = \int_{-\infty}^{\infty} \Delta C \; P(\Delta C) \, P_{Accept}(\Delta C) \, d\Delta C \tag{2}$$

$P_{Accept}(\Delta C)$ is the probability that the acceptance function will accept a move with cost ΔC [Rome84]. It commonly has the value 1 for $\Delta C \leq 0$ and $e^{\frac{-\Delta C}{T}}$ for $\Delta C > 0$ [Sech85]. We note here that $P(\Delta C)$ in equation (2) must be the distribution measured on a *running* Simulated Annealing process *at* the equilibrium temperature. This distribution is difficult to measure, and will be discussed further in Section 3.1.

Using this $P_{Accept}(\Delta C)$ we can split equation (2) into two parts and, at equilibrium from equation (1) we can equate it to zero:

$$\int_{-\infty}^{0} \Delta C \; P(\Delta C) \, d\Delta C \; + \int_{0}^{\infty} \Delta C \; P(\Delta C) \, e^{\frac{-\Delta C}{T}} \, d\Delta C = 0 \tag{3}$$

Thus *equilibrium* can now be defined as the state where, at a given $T = T_{eq}$, the distribution $P(\Delta C)$ satisfies equation (3).

Conversely, the *equilibrium temperature* of a placement with a distribution $P(\Delta C)$ is the temperature, T_{eq}, for which equation (3) is satisfied.

2.1 An Equilibrium-Nearness Measure

Using equation (3) we can invent a measure of the nearness of a given Simulated Annealing state to equilibrium. Define E_- to be the absolute value of the first term in the equation, that is

$$E_- = \left| \int_{-\infty}^{0} \Delta C \; P(\Delta C) d\Delta C \right|$$

Similarly let E_+ be the second term of equation (3):

$$E_+ = \int_{0}^{\infty} \Delta C \; P(\Delta C) \, e^{\frac{-\Delta C}{T_m}} \, d\Delta C$$

Where T_m is the temperature of the Simulated Annealing process. We can now define the *Cost Force Ratio*, (CFR) as:

$$CFR = \frac{E_-}{E_+ + E_-} \times 100 \qquad (4)$$

The closer CFR is to 50% (the expected value of the good moves being equal to the expected values of the bad moves, $E_- = E_+$) the closer the system is to equilibrium.

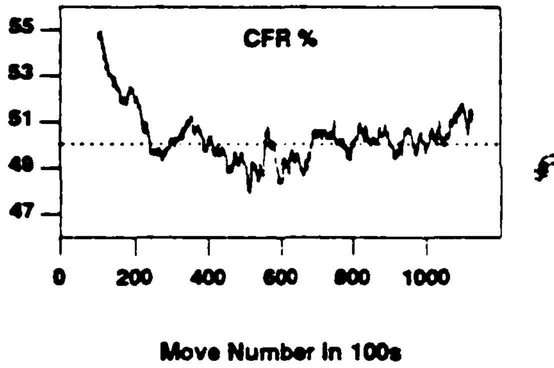


Figure 1 - *CFR vs Move as Process Achieves Equilibrium*

All experiments in this paper use a placement of the 833 standard cell Primary1 circuit from the Preas-Roberts standard cell benchmark suite [Prea87]. The placement was produced by the SALTOR Simulated Annealing placement program [Rose86,Rose88], which is based on the ideas of the Timberwolf standard cell placement program [Sech85]. Figure 1 is a plot of CFR versus generated move number for a Simulated Annealing process running on circuit Primary1, as it goes from non-equilibrium to equilibrium at temperature 400 changing to 300. CFR is determined by keeping a window of ΔC values multiplied by the P_{Accept} function and using this to calculate E_+ and E_-. In this figure the CFR comes down from an initial value of 55% and hovers around 50%. This shows that the CFR indicates when equilibrium has been achieved. It varies about the 50% point due to the stochastic nature of the algorithm and the approximation of measuring the CFR in a finite window.

3 Measuring Temperature

As defined in Section 2, the temperature of a placement is the temperature at which the Simulated Annealing process running on the placement is in equilibrium. In this section we present a method for measuring the temperature of an arbitrary placement.

The method is called the CFR Binary Search and has the following outline:

1. Measure $P(\Delta C)$ for the given circuit under the Simulated Annealing process. This is discussed in detail in Section 3.1.

2. Set the starting search temperature, T_m, arbitrarily.

3. Based on the current T_m, calculate $P_{Accept}(\Delta C) = e^{\frac{-\Delta C}{T_m}}$ for $\Delta C > 0$ and $= 1$ for $\Delta C \leq 0$.

4. Calculate the Cost Force Ratio, CFR, using $P_{Accept}(\Delta C)$ and equation (4).

5. If CFR < 50, reduce T_m according to a binary search and go to step 3;
   If CFR > 50, increase T_m according to a binary search and go to step 3.

6. When CFR = 50, T_m is the equilibrium temperature, T_{eq}. Finish.

Each iteration of the CFR Binary Search requires only the recalculation of the positive portion of the acceptance function probability, $P_{Accept}(\Delta C)$, and subsequently E_+ and CFR since E_- does not change with T_m. Note also that $P(\Delta C)$ need only be generated once. This is important since it takes many moves (10^4 to 10^5) to get an accurate picture of the probability distribution.

3.1 Measurement of the Probability Distribution

A key and difficult step in the CFR Binary Search temperature measurement procedure is the measurement of the distribution $P(\Delta C)$. There are two possible methods:

1. Static Measurement. $P(\Delta C)$ is measured by generating (but not accepting) moves in the Simulated Annealing process on the placement, and recording the frequency with which each cost occurs. These *virtual* moves do not change the placemen .

2. Dynamic Measurement. $P(\Delta C)$ is measured by generating *and* accepting moves on the placement. Here the placement does change as the measurement is made.

For the general case of any Simulated Annealing application a static measurement will not give the correct distribution. This is because a static measurement of $P(\Delta C)$ could be taken when the system was at a local (but not global) optimum. In this case there would be no good (negative) moves generated and since

E_- would thus be 0 the temperature would appear to be 0, which is incorrect in the case of a local optimum. Similar problems can occur when the state is at or near discontinuities in the energy landscape.

The dynamic measurement approach must run the Simulated Annealing process *at* its equilibrium temperature. Using a different temperature would cause the placement's temperature to change. Unfortunately the equilibrium temperature is the quantity we are seeking, and is not known. This is a dilemma not unlike the Heisenberg uncertainty principle: the act of measuring the temperature this way can cause the temperature to change.

An alternative is to measure $P(\Delta C)$ using the static method, and to determine how accurate this is as an approximation. The accuracy is entirely problem dependent - it depends on the energy landscape of the underlying Simulated Annealing formulation. We have experimented to determine the accuracy for the standard cell placement problem and have found that the static measurement of $P(\Delta C)$ is almost exactly the same as the dynamic measurement. Figure 2 shows a plot of a static distribution and a dynamic distribution measured on circuit Primary1 at temperature 300. Measurements and numerical comparisons on this and several other circuits at various temperatures have shown very small differences between the static and dynamic measurements. Thus we will use the static measurement of $P(\Delta C)$ in the temperature measurement algorithm.

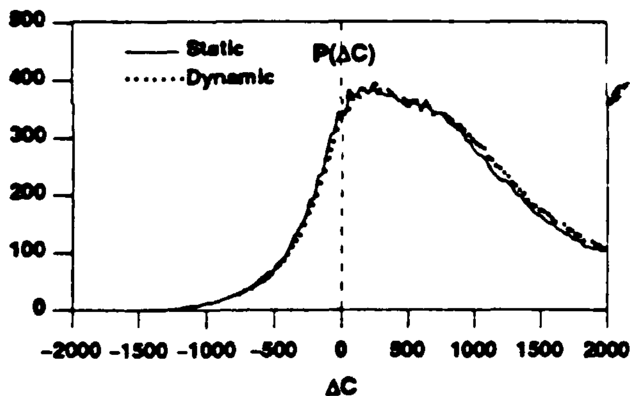


Figure 2 - *Comparison of Static and Dynamic Measurement*

3.2 Measurement of Annealing Placements

The CFR Binary Search was used to measure the temperature of a set of Primary1 placements produced by the SALTOR Simulated Annealing placement program [Rose86,Rose88]. Each placement was measured statically using N = 100,000 virtual moves to experimentally determine $P(\Delta C)$. Table 1 gives the temperature at which each placement's Simulated Annealing process was terminated (while in equilibrium), and the measured temperature using the CFR Binary Search.

The measured temperature is quite accurate at the higher temperature, usually less than 7% error. The lower temperature measurements are *proportionately* less accurate. The error is due to two effects: First, there is a slight difference, as discussed above, between the static and the (more correct) dynamic measurement of $P(\Delta C)$. Second, at lower temperatures, there are fewer negative moves, and so the accuracy of E_- decreases, decreasing the accuracy of CFR and hence the temperature measurement.

SA Produced Temperature	CFR Binary Search Measured Temp	Difference
500	496	-4
405	420	+15
294	285	-11
213	215	+2
153	164	+11
99	97	-2
57	60	+3
28	28	0
9	15	+6
2	4	+2

Table 1 - *Temperature Measurement of Annealing Placements*

This last point can be seen experimentally: figure 3 is a plot of the percentage standard deviation of the measured temperature as a function of the number of virtual moves, N, for temperatures 28, 153 and 405. The standard deviation was calculated from five runs at each number of virtual moves.

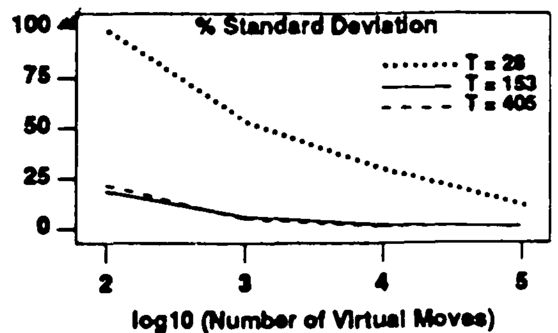


Figure 3 - *Variation of Temperature vs. Number of Moves*

The variation is a decreasing function of N, as would be expected. The figure illustrates the increase in percentage variation at lower temperature.

4 Measurement of Min-Cut Placements

Our goal is to determine the starting temperature when switching from a non-annealing placement algorithm to an annealing-based one. In this section we test the ideas presented above on the Min-Cut placement algorithm [Dunl85].

Several terms first need to be defined for Min-Cut placements, as shown in Figure 4. A Min-Cut placement

algorithm is characterized by, among other things, the order and spacing of the cut lines applied. In Figure 4, the rec' ıgle represents the entire placement, over which is laid a set of vertical and horizontal cut lines. If the spacing of the vertical cut lines is V and of the horizontal cut lines is H, then the *cut area*, A, is given by $A = V \times H$.

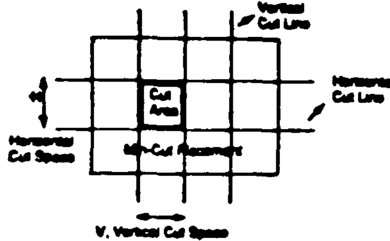


Figure 4 - *Definition of Cut-Area*

One difficulty with measuring the temperature of non-annealing produced placements is that the definition of temperature presented in Section 2 depends on the associated Simulated Annealing process being in equilibrium. It is clear, however, that a placement produced by a non-annealing algorithm is not in equilibrium. Thus we must make an approximation and assume that a min-cut placement can be thought of as being in equilibrium at some temperature. The effect of this approximation is measured in the next section where we compare the CFR Binary Search method with a more direct method.

4.1 Measurements

Using the CFR Binary Search method we measured the temperature of several Min-Cut placements with different cut areas. These placements were produced by the ALTOR standard-cell placement program [Rose85]. Table 2 gives the measured temperature for each placement and its cut area.

Cut Area $\mu m^2 \times 10^4$	Temperature Measured Binary Search	Delta Method	Difference
2021	398	374	+24
1011	234	200	+34
505.3	162	132	+30
252.6	124	96	+28
126.3	91	67	+24
63.22	73	50	+23
31.58	49	40	+9
25.24	40	32	+8
12.60	34	30	+4
7.697	29	27	+2
3.139	28	26	+2

Table 2 - *Temperature Measurement of Min-Cut Placements*

To check the CFR Binary Search measurements, the placements were measured using a different approach, called the *Delta Method*. It finds the temperature of a placement by running a dynamic annealing process on the placement over a range of temperatures. The percentage difference in absolute cost function after (100 moves per cell are made) is measured. When a temperature is found for which this difference is less than 2%, that is taken as the temperature of the placement. This is a direct way of experimentally finding the temperature at which the change in cost function is near 0. Table 2 gives the temperatures determined by the Delta Method, and the difference between the CFR Binary Search and the Delta Method. The CFR Binary Search measurement for Min-Cut placements is not as accurate as those for Annealing-produced placements, yet it does track the temperature reasonably well.

The CFR Binary Search method consistently overestimates the equilibrium temperature due to the fact that a min-cut placement is not in equilibrium, as discussed above.

5 Conclusions

We have presented a method for determining the *temperature*, in the Simulated Annealing sense, of an arbitrary placement. It uses a new view of Simulated Annealing state that is based on the probability distribution of the change in cost function. The temperature of several Simulated Annealing placements have been measured with good accuracy. The temperature of a set of Min-Cut placements has also been measured. This method is useful for determining the starting temperature when switching from a non-annealing based placement strategy to an annealing-based one.

6 References

Aarts85

E.H.L. Aarts, P.J.M. van Laarhoven, "A New Polynomial-Time Cooling Schedule," Proc. ICCAD 85, November 1985, pp. 206-208.

Dunl85

A. Dunlop, B. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," IEEE Trans on CAD, Vol. CAD-4, No. 1, Jan 1985, pp 92-98.

Grov87

L.K. Grover, "Standard Cell Placement Using Simulated Sintering," Proc. 24th DAC, June 1987, pp. 56 - 59.

Preas87

B.T. Preas, "Benchmarks for Cell-Based Layout Systems," Proc. 24rd Design Automation Conference, June 1987, pp. 319-320.

Romeo84

F. Romeo, A. Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications," Memorandum No. UCB/ERL M84/34, March 1984, University of California, Berkeley.

Rose85

J.S. Rose, W. Snelgrove, Z. Vranesic, "ALTOR. An Automatic Standard Cell Layout Program," Proc. Canadian Conf on VLSI, Nov 1985, pp. 168-173.

Rose86

J.S. Rose, D. Blythe, W. Snelgrove, Z. Vranesic, "Fast, High Quality VLSI Placement on an MIMD Multiprocessor," ICCAD 86, Nov. 86, pp. 42-45.

Rose88

J.S. Rose, W.M. Snelgrove, Z.G. Vranesic, "Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing," IEEE Transactions on CAD, Vol. 7, No.3, March 1988, pp. 387-396.

Sech85

C. Sechen, A. Sangiovanni-Vincentelli, "The Timberwolf Placement and Routing Package," IEEE JSSC, Vol. SC-20, No. 2, April 1985, pp 510-522.

Whit84

S.R. White, "Concepts of Scale in Simulated Annealing," Proc. Int. Conf. on Computer Design, October 1984, pp. 646-651.

The Effect of Logic Block Complexity on Area of Programmable Gate Arrays

Jonathan Rose
Computer Systems Laboratory, Stanford University, Stanford, CA 94305

Robert J. Francis, Paul Chow, David Lewis
Dept. of Electrical Engineering, University of Toronto, Toronto, Ont. Canada

1 Introduction

The Programmable Gate Array (PGA) is an exciting new idea in semi-custom integrated circuits that reduces the IC *manufacturing time from months to minutes* and *prototype cost* from tens of kilodollars to under $100. The PGA was introduced in [Cart86] and newer versions have been presented in [Hsie87,Hsie88,ElGa88a,ElAy88]. It is similar to a gate array in structure, but can be field-programmed to specify the function of the basic logic blocks and their interconnection. This paper studies the effect of logic block complexity on total circuit area for PGAs.

The *architecture* of a PGA consists of its logic block function, interconnection scheme, I/O block design and the global structure. There are many tradeoffs between architecture, area, and speed, each of which depends heavily on the *programming technology*. Programming technology is the underlying method by which the logic function is set and the connections are implemented at program time. For example, the programming technology used in [Hsie88] is based on static RAM and pass transistors, while that of [ElGa88a] uses an anti-fuse. In this paper we focus on the effect of logic block complexity on PGA area, ignoring speed considerations. While circuit speed is very important, this work represents an initial exploration into plausible architectures from an area perspective.

We address two questions: First, should the basic logic block contain a significant amount of fixed hardware, such as a D flip-flop? Our experimental results indicate that a D flip-flop is desirable for large programming technologies (like SRAM [Hsie88]) but that it is inefficient for smaller technologies such as the anti-fuse. Second, for logic blocks containing arbitrary combinational logic functions, (i.e. *any* K to 1 logic function) what is the best number (K) of inputs to use? Surprisingly, the best number of inputs remains nearly constant over a wide range of programming technologies and was almost the same whether or not the block contained a D flip-flop.

This work was supported by DARPA Contract #N00014-87-K-0828, and NSERC Operating Grants #A4029 and #A4053.

2 Experimental Procedure

To answer these questions, our approach is to implement a set of circuits in a variety of logic blocks and programming technologies, and determine the area required for each. This data will indicate an appropriate choice of logic block, in terms of area, for a given technology.

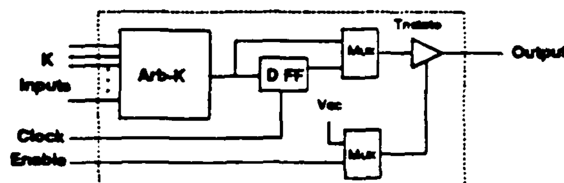


Figure 1 - *General Model of Logic Block*

Figure 1 depicts the general architectural model used for the logic block. It consists of a K-input arbitrary combinational logic function (referred to as an "Arb-K"), connected to a D flip-flop followed by a multiplexer that selects either the flip-flop output or the Arb-K output. Its output is passed to a tristate driver that can be enabled by another input or left permanently on. To determine if the D flip-flop is beneficial, two variations of this basic model will be considered: one that contains the D flip-flop, and one that does not.

The global architecture of the PGA under consideration is shown in Figure 2. It is a regular array of logic blocks, separated by horizontal and vertical routing channels. The number of tracks in all of the routing channels, W, is the same. Since we want to know the area requirements of a logic block architecture, a crucial concept in this procedure is that W is *determined* by the placement and routing for each circuit.

The following procedure performs the circuit implementation:

Input: a logic circuit, a range of K's indicating how many inputs on the Arb-K block, and a set of programming technologies.

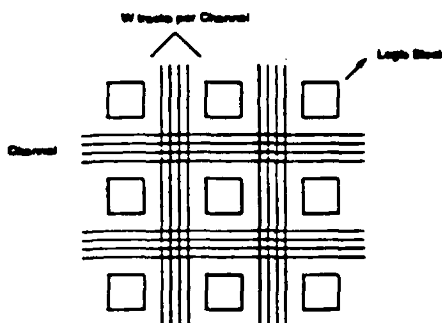


Figure 2 - *Routing Model of PGA*

Output: for each (K, programming technology) pair the area required to implement the circuit with a logic block that contains a D flip-flop, and the area for a logic block that does not.

Procedure: For each logic block type:

1. Partition the original circuit into the current logic block. This is sometimes called *technology mapping* [Detj87], but is a more difficult problem for PGAs because each logic block can collapse many combinational logic functions. The *Chortle* program was developed to do this mapping [Fran88]. It uses a greedy algorithm that tries to collapse as many standard cells as it can into each logic block.

2. Perform the placement of the resulting circuit. This is done using the Altor placement program [Rose85], which is based on the min-cut placement algorithm [Breu77]. Altor makes the array as square as possible.

3. Perform the global routing of the circuit. Global routing determines the path of channels that each wire is to take, and hence determines the maximum number of tracks required in each channel, W. The algorithm used is similar to the one described in [Rose88], but is changed to fit the routing model pictured in Figure 2.

4. Section 3 describes a model for the logic block area and routing area as a function of K and programming technology. With this model, W, and the placement dimensions, the circuit area for a range of programming technologies is calculated.

The above procedure makes the approximation that the global routing track count determines the number of tracks required in a channel. This is generally accepted as true for unconstrained channel routers, but may not be true for switch-based routing schemes. We have reason to believe, however, that the error in this assumption is only a few tracks [ElGa88b].

3 Architecture Model

The area of a logic block is a function of the number of its inputs, the amount of fixed hardware it contains, and the programming technology. The pitch of the routing track can be approximately modeled as a function of the programming technology.

The programming technology is represented by one parameter: the area required to store one bit in the technology, or Bit Area (BA). For example, in the Xilinx PGA [Hsie88], the Bit Area is the area of a static RAM bit. In the Actel PGA [ElGa88a] the bit area is much smaller, close to the space required by an anti-fuse. The overhead required to access the Arb-K block and the area required by the D flip-flop (if it is present) and all other non-arbitrary logic function hardware is represented by a second parameter, called the Fixed Overhead Area (FA).

An Arb-K block, because it can implement *any* K to 1 logic function, requires 2^K bits of information to be stored and so must have area proportional to 2^K. Using this, we can derive the following expression for logic block area:

$$Logic\ Block\ Area = BA \times 2^K + FA$$

where BA is the bit area in the programming technology and FA is the fixed overhead.

The basic technology is assumed to be 1.25μm CMOS. FA has been calibrated using data acquired from Xilinx [Cart88], giving FA = 1200μm^2 for logic blocks without a D flip-flop and 16000μm^2 for logic blocks with a D flip-flop. The corresponding Bit Area for an SRAM programming technology is 400μm^2 and is roughly 40μm^2 for an anti-fuse technology. In our experiments, we will vary the Bit Area between and above these two values.

Though the PGA interconnection scheme is not addressed directly in this paper, the *area* required by routing is an important factor in determining the logic block. We need to know the *pitch* of the routing track as a function of programming technology. Each routing track will need at least one bit of information in it, and probably several — to determine if a set of switches or fuses is open or closed. Since it is difficult to lay out a bit with highly non-square aspect ratios, the pitch of a routing track is approximated as the square root of the area required by a bit, i.e. $Routing\ Pitch = \sqrt{BA}$.

4 Experimental Results

The circuits used in these experiments are five standard-cell circuits obtained from Bell-Northern Research [Mart88],

ranging in size from 420 to 1681 standard cells. They consist of a mix of random logic and data path circuits. Figure 3 is a plot of absolute area for the PGA versus number of inputs to the arbitrary combinational logic block, K, for a 1073 standard cell circuit.

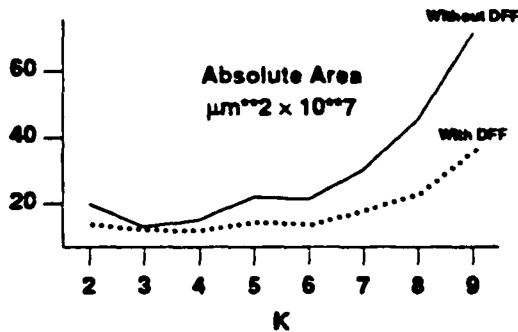


Figure 3 - *Area for versus K for One Circuit*

There are two curves - one giving area when the logic block contains a D flip-flop, and one without. The programming technology, BA = 415μm^2, corresponds to an SRAM-based approach [Hsie88]. Using similar data for all of the circuits, with more programming technologies, the questions raised in the introduction were addressed.

4.1 Number of Inputs to Logic Block

Figure 4 shows the sum of the normalized areas over all of the circuits, versus K. The normalized area for a circuit is determined by dividing the area using logic block K by the best area over all K. The logic block used in this data contains a D flip-flop. Figure 4 gives several plots for different bit areas (programming technologies).

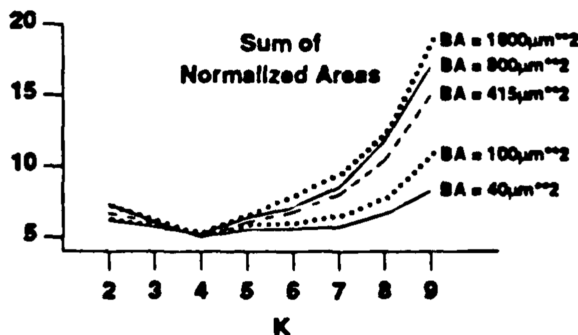


Figure 4 - *Sum of Normalized Areas versus K Using DFF*

It is clear, from the dip at K = 4, that a 4-input arbitrary logic block consistently achieves the lowest area. Surprisingly, this number is constant over a wide range of bit areas. It is due to the fact that, for a given K, the area is predominantly a linear function of the bit area.

The number of logic blocks increases when the logic block has no flip-flop because the D flip-flops must then be implemented in combinational gates. Since the size of each logic block is less, the final area may or may not be smaller. Figure 5 is a normalized area plot for logic blocks that do not contain D flip-flops. The best number of inputs in this case is three, only slightly different than the D flip-flop case. Again, this number is independent of programming technology.

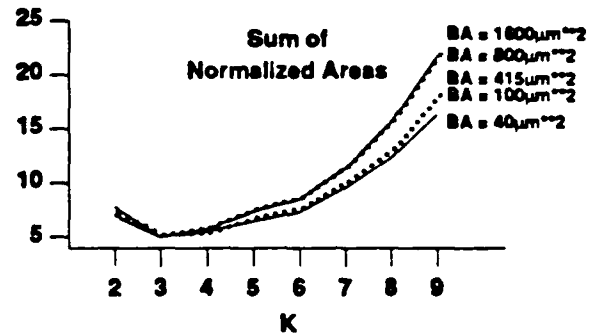


Figure 5 - *Sum of Normalized Areas versus K Without DFF*

In both cases, the best K is low (3 or 4) primarily because the circuits cannot make effective use of the larger K blocks. because the increasing functionality comes at the cost of a much greater active area, which exponentially increases in K, it doesn't pay to use the larger logic blocks.

4.2 Utility of the D Flip-Flop

Figure 6 is plot of circuit area using and not using a D flip-flop versus Bit Area for a 1073 standard cell circuit. The circuit area used is the one obtained with the lowest area K.

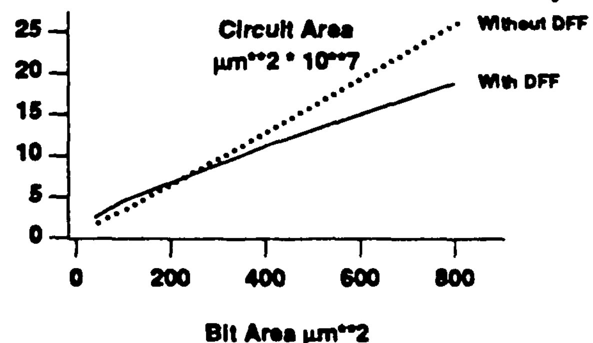


Figure 6 - *Area versus Bit Area 1 Circuit*

This figure shows that, for very small bit areas, it is advantageous not to use a D flip flop, but larger bit areas

Jonathan Rose, Center for Integrated Systems, Stanford CA 94305 (415)-725-3635

perform better by using a D flip flop. This is true for all of the circuits, but the cross-over point is different for each.

Figure 7 is a plot of $\frac{\textit{Area Without Flip-Flop}}{\textit{Area With Flip-Flop}}$ versus Bit Area for each circuit, indicating when it is advantageous to use a flip-flop. In the smaller bit areas, corresponding to an anti-fuse programming technology, the use of a D flip flop is unprofitable. This is the case in the Actel PGA [ElGa88]. The middle and larger bit areas, corresponding to the SRAM program technology, can benefit by including a D flip-flop, and in fact the Xilinx PGA [Hsie88] uses two D flip-flops.

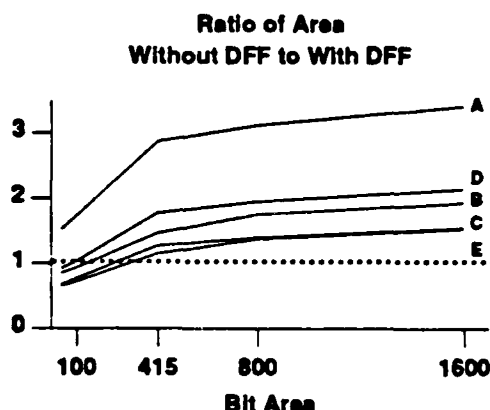


Figure 7 - *Without DFF:With DFF versus Bit Area*

5 Conclusions and Future Work

We have presented a procedure and model to evaluate different logic block architectures for Programmable Gate Arrays, on the basis of circuit area. Using this method, for a particular set of circuits, we have demonstrated a good number of inputs to use for the arbitrary combinational logic block. In addition, we have displayed the trade-off between programming technology area and utility of a D flip-flop in the logic block.

There is an enormous amount of future work to be done in this field. We would like to investigate more kinds of logic blocks - in particular those with less arbitrary logic functions. Other work will directly address questions dealing with circuit speed. This relates to the specific architecture of the interconnection scheme. All of this work needs to be implemented on a wider range of circuits. New CAD algorithms need to be developed for PGAs. Our technology mapper needs more development, and the placement and routing needs to address the specific needs of PGAs. PGAs, because they promise such enormous economic advantages, are a fertile and growing field of research and development.

6 Acknowledgements

The authors are grateful to Grant Martin of Bell-Northern Research for supplying the circuits and cell functional descriptions.

7 References

[Breu77]
M.A. Breuer, "Min-Cut Placement," Journal of Design Automation and Fault-Tolerant Computing, pp. 343-362, Oct 1977.

[Cart86]
W. Carter et. al, "A User Programmable Reconfigurable Gate Array," Proc. 1986 CICC, May 1986, pp. 233-235.

[Cart88]
W. Carter, Private Communication.

[Detj87]
E.Detjeus et. al, "Technology Mapping in MIS", Proc. ICCAD 87, Nov 1987, pp. 116-119.

[ElAy88]
K. El-Ayat, et. al, "A CMOS Electrically Configurable Gate Array," Proc. 1988 ISSCC, pp. 76-77.

[ElGa88a]
A. El Gamal, et. al, "An Architecture for Electrically Configurable Gate Arrays," Proc. 1988 CICC, May 1988, pp. 15.4.1 - 15.4.4.

[ElGa88b]
A. El Gamal, Private Communication.

[Fran88]
R.J. Francis, "Chortle: A Technology Mapping Algorithm for Programmable Gate Arrays," in preparation.

[Hsie87]
H. Hsieh et. al, "A Second Generation User Programmable Gate Array," Proc. 1987 CICC, May 1987, pp. 515-521.

[Hsie88]
H. Hsieh, et. al "A 9000-Gate User-Programmable Gate Array," Proc. 1988 CICC, May 1988, pp. 15.3.1 - 15.3.7.

[Mart88]
Grant Martin, Bell-Northern Research, private communication.

[Rose85]
J. Rose, et. al, "ALTOR: An Automatic Standard Cell Layout Program," Proc. Can. Conf. on VLSI, Nov. 1985, pp. 168-173.

[Rose88]
J. Rose, "LocusRoute: A Parallel Global Router for Standard Cells," Proc. 25th DAC, June 1988, pp. 189-195.

Memory-Reference Characteristics of Multiprocessor Applications under MACH

Anant Agarwal* and Anoop Gupta
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Shared-memory multiprocessors have received wide attention in recent times as a means of achieving high-performance cost-effectively. Their viability requires a thorough understanding of the memory access patterns of parallel processing applications and operating systems. This paper reports on the memory reference behavior of several parallel applications running under the MACH operating system on a shared-memory multiprocessor. The data used for this study is derived from multiprocessor address traces obtained from an extended ATUM address tracing scheme implemented on a 4-CPU DEC VAX 8350. The applications include parallel OPS5, logic simulation, and a VSLI wire routing program. Among the important issues addressed in this paper are the amount of sharing in user programs and in the operating system, comparing the characteristics of user and system reference patterns, sharing related to process migration, and the temporal, spatial, and processor locality of shared blocks. We also analyze the impact of shared references on cache coherence in shared-memory multiprocessors.

1 Introduction

Although we now have a reasonably good understanding of memory system design for uniprocessors, very little is understood about memory system design for multiprocessors. A major reason for this has been the lack of real data about memory reference patterns for multiprocessors, because of the difficulty of tracing such machines. The problem of getting realistic trace data is even more acute if one wishes to the study the effects of operating system references, process migration, and other such real system events. This paper attempts to correct this situation and analyzes memory reference patterns of several parallel applications running under the MACH operating system on a shared-memory multiprocessor. The address traces used in our study were obtained from a 4-processor VAX 8350 multiprocessor using an extended version of the ATUM [1] address tracing technique. These traces contain both system and user memory references, including process migration information.

Analysis of shared-memory reference patterns is needed to determine the most suitable organization of the memory hierarchy in multiprocessors. For example, several cache consistency algorithms proposed in the literature are based on subtle differences in the expected memory reference patterns; lacking detailed data, the benefits of one scheme over another cannot be assessed accurately. While some previous studies have looked at shared-memory reference patterns, e.g., [2], they did not fully address issues such as the temporal, spatial, and processor locality of shared data, sharing in the operating system, and the impact on cache consistency. For example, we show that shared references display a significant amount of processor locality. The average number of read and write references to a write-shared block before a remote reference are 4 and 2 respectively. This locality is exploited by the write-back class of cache coherence schemes to significantly reduce the cost of references to shared data. Another surprising result that we observed for shared data references is that the total bus bandwidth required is minimized when block size is 4 bytes and increases as the block size is increased. We also observe that processor migration causes a large increase in the sharing level as observed by the caches, which can greatly increase cache coherence traffic on the bus.

This paper is organized as follows. Section 2 presents background information about the ATUM address tracing technique, the applications measured, and the MACH operating system. Section 3 defines our multiprocessor model and the terminology used throughout the paper. Section 4 constitutes the bulk of the paper and is devoted to analyzing the traces. This section characterizes shared-memory reference patterns and looks at the impact of the reference characteristics on cache consistency algorithms. Specifically, in Section 4.1 we present data about the general characteristics of the traces, including statistics about interlocked instructions. Section 4.2 assesses the temporal and processor locality of shared references. Section 4.3 focuses on how the memory reference characteristics affect the performance of various cache consistency algorithms. Section 5 concludes the paper.

2 Background and Methodology

Our study is based on trace analysis. The traces are obtained using a multiprocessor extension of the ATUM tracing scheme [1]. ATUM stands for Address Tracing Using Microcode and works as follows: During the execution of each instruction, the microcode writes out the memory references

*Anant Agarwal is currently with the Laboratory for Computer Science (NE43-418), M.I.T. Cambridge, MA 02139

In Proceedings of SIGMETRICS '88

made by the processor to a portion of memory reserved for tracing. In the multiprocessor extension of ATUM, each access to trace memory is interlocked to enable the microcode in several processors to write their references to this memory. Thus a trace contains interleaved address streams of several processors. The traces used for this study were gathered on a 4-CPU VAX 8350 machine running the MACH operating system. ATUM traces are "complete" in that they capture all operating system and multiprogramming activity. Each trace is roughly 3.5 million references long. In addition to addresses, ATUM records the opcodes, and the virtual-to-physical translations that occur during translation-lookaside-buffer (TLB or TB) misses. A location is considered shared when it is referenced by more than one CPU. Because different processes could access a given shared location with different virtual addresses, sharing is detected by translating the various virtual addresses of a shared location to its common physical address.

The traces used in this paper are obtained from three programs: POPS, THOR, and PERO. POPS [3] is a parallel implementation of a rule-based programming language called OPS5, which is a widely used languages for the building expert systems. It exploits parallelism at a fine granularity and makes extensive use of the shared memory provided by the architecture. THOR is a parallel implementation of a logic simulator done by Larry Soule at Stanford University. The simulator transforms the task of circuit simulation into a series of node evaluations, where each node corresponds to a device in the circuit. The parallel implementation evaluates these nodes in parallel, while taking care of the dependencies between them. PERO is a parallel VLSI router written by Jonathan Rose at Stanford [4].

We briefly describe the MACH operating system, since some of the shared references in the traces belong to it, and also because the programming style used in the applications was influenced by it. MACH is a multiprocessor operating system developed at Carnegie Mellon University. It is binary compatible with Berkeley Unix, and provides several new facilities to support parallel processing. It provides facilities for multiple tasks to share memory permitting the exploitation of very fine grained parallelism. All three programs make use of multiple tasks that share memory to communicate with each other and to share information. MACH is not a totally symmetric operating system in that kernel interrupts are handled by processor zero. This causes the memory reference pattern of processor zero to be different from that of the remaining processors. In parallel programs, where many tasks are performing I/O, the high level of OS interrupts can also cause excessive process migration. Fortunately, none of the programs that we study in this paper do very much I/O.

Table 1 presents general trace characteristics for the three programs. The columns denote the total number of references, instruction references, data reads, data writes, user and system references. Instruction and data references are about equal, while there are roughly three reads to every write. About 12% of all references are system.

The ATUM traces used for this study do have some limitations. The machine used had only 4 CPUs and it is not clear how to extend the results to a larger number of processors. Work on this issue is in progress. Another problem is the unavailability of a large number of applications, but the number is growing.

Table 1: Summary of trace characteristics. All numbers are in thousands

Trace	Refs	Inst	DRead	DWrt	Usr	Sys
POPS	3142	1624	1257	261	2817	325
THOR	3222	1456	1398	368	2727	495
PERO	3508	1834	1266	409	3242	266

3 Multiprocessor Model and Definitions

The multiprocessor model we assume for our analyses in this paper is quite straightforward. We assume that the system consists of several processors each with its own cache memory. The caches are connected to a common system bus on which shared main memory is located. We also make the simplifying assumption that caches are infinite in size, since we would like to concentrate on traffic caused due to shared data and not mix it up with traffic due to limited cache size.

We introduce some nomenclature to help explain memory access patterns. A *block* is the unit of data transfer between the cache and main memory. For the rest of the paper, we assume block size to be 1 word (4 bytes). The small block size is chosen so that the reference behavior for each data object can be derived. However, characterization using larger block sizes is also important to study the spatial locality of shared objects, and is dealt with in Section 4.3.

A *read-shared* block is one that is shared (accessed by multiple processors), but never written into. A *write-shared* block is one that is shared, and both read and written into. A reference to a block B by processor i is said to *ping* if the previous reference to that block was by processor j, where $j \neq i$. We call such a reference a pinging reference. Conversely, a reference to a block B by processor i is said to *cling* if the previous reference to that block was also by processor i. Such a reference is called a clinging reference. By these definitions, a ping can only occur on a reference to a shared block. Pings and clings to a block are determined simply by keeping track of which processor last referenced a block. References are read references or write references depending on whether the operation performed is a read or write. The state of a block (clean/dirty) is determined by the references of the processor accessing it currently. A block is said to be dirty if it has been written into after the previous pinging reference to it. Therefore, a block always starts out clean after a pinging reference to it.

The notion of clings and pings yields useful insights on how various shared-memory multiprocessor architectures would perform. The appealing feature of clings and pings is that they do not depend on implementation details such as cache sizes. Assuming a local cache, clinging read references never need the bus; pinging read references need to use the bus only if the read misses or if the block is dirty in another cache. A bus transaction must occur on a pinging write reference. In the ensuing discussion we will show results on the time intervals between such clings and pings, and also on the frequency of various kinds of clings and pings. The time interval plots are a useful method of depicting the temporal locality of shared-memory references, while the frequency of clings and pings is a method of showing the "processor locality" of references to a block. Besides spatial locality and temporal locality, the form of locality important in a multiprocessor

context is *processor locality* – the tendency of a processor to access a block repeatedly before an access from another processor. A direct impact of this locality is noticed in the performance of various cache consistency schemes, which exploit different locality patterns in references to read-shared or write-shared blocks. Also notice that a high temporal locality of pinging references yields a low processor locality, and negatively impacts the performance of multiprocessor caches.

To separate the effects of process migration, we also present numbers for *process-migration-shared* blocks. These are blocks accessed from processor i by process p, and also from processor $j \neq i$ by the same process p. On the other hand, *real-shared* blocks, are blocks accessed from processor i by process p, and also from processor $j \neq i$ by process q, where $q \neq p$ always holds.

It is useful to have a notion of time in the context of multiprocessor execution. Our traces contain interleaved memory accesses by the various processors in approximately the same order they occurred. However, the exact time at which the reference was made is not clear. For example, if the processors i, j, and k each made references at real time instants t, $t+1$, and so on, the trace might have references $i_t, j_t, k_t, i_{t+1}, j_{t+1}, k_{t+1}$, and so on, where the order of the tth references of the 4 processors might be random with respect to each other. The traces also show clusters of memory references by the same processor, and the time interval between references by the same processor also varies.

Due to this nature of the reference pattern, we will not try to approximate real time. Instead, we will use the order of occurrence of a reference in the trace as the index of time. So the rth reference in the trace is considered to have occurred at time r.[1] The paper considers several cases where the traces are filtered to extract specific references (e.g., user), and to enable comparisons, the time index used for a reference depends on its index in the original trace. For example, when we filter out operating system references while studying sharing in the user address space, the time index of a user reference corresponds to its position in the unfiltered trace.

4 Results and Analyses

We first present some general statistics about the traces, including data about interlocked instructions. We then present statistics about temporal and processor locality found in the traces when only user references are included and there is no process migration sharing, when both system and user references are included, and when the effects of process migration are taken into account. We then evaluate three different cache coherence schemes on the basis of the amount of traffic they generate on a shared bus. Unless stated otherwise, we assume infinite caches and a 4-byte block size.

4.1 General Statistics

The statistics in Table 2, for both instructions and data references of user and the operating system, relate to the number of unique blocks and the proportion of references to shared blocks in the traces.

Table 2: Proportion of shared references and unique shared blocks when the blocksize is 4 bytes. Both *instruction and data* references of *user and OS* are included. All numbers are in thousands. Block size is 4 bytes.

Trace	Refs	Uniq Blks	Shd Refs	Shd Blks
POPS	3142	37.8	2122	23.7
THOR	3222	78.3	1881	7.0
PERO	3508	22.6	218	4.7

Table 3 gives the same statistics, but only for data references of both user and the operating system. In addition, Table 3 presents the number of blocks that are written. Because the instruction space is usually read-only, it can be treated specially in memory management, and so most of the statistics presented later correspond to data references alone. Table 4 presents the same statistics for user data references alone.

When both user and the operating system data references are considered, the ratio of shared references to all data references (averaged over all three traces) is 0.25; the ratio is 0.27 when only user data references are considered. We see that the level of sharing in the operating system is only slightly lower than in user.

These traces have an insignificant amount of process-migration-related sharing. We also looked at some other traces for the same applications with a large amount of process migration, and the levels of sharing are drastically different in these traces. The ratio of shared to total is 0.9 for user data references when process migration is high; when process migration effects are excluded (only references to real-shared blocks are counted), the ratio of user data references and all data references falls to 0.2.

4.1.1 Statistics for Interlocked Instructions

The VAX architecture provides seven interlocked instructions for synchronization. These are: BBSSI – branch on bit set and set interlocked; BBCCI – branch on bit clear and clear interlocked; ADAWI – add aligned word interlocked; INSQHI, INSQTI, REMQHI, REMQTI – four instructions to manipulate linked lists (queues) in an interlocked manner. The usage of these instructions is presented in Table 5, with separate numbers given for operating system code and user code.

Table 5 shows that only BBSSI and BBCCI instructions occur in the trace. The ADAWI instruction is used in the POPS code, although it does not occur in the instruction references that our trace contains. These statistics show the strong preference of programmers to use the simpler test&set type instructions for synchronization, rather than using the more complex queue manipulation instructions.

The number of interlocked instructions as a fraction of all instruction references is 0.1%-1.6% for the three programs. While the fraction is as high as 1.2%-1.6% for POPS and THOR, the fraction is only 0.1% for PERO. The reason is simply that the author of PERO had made an explicit decision not to use locks for the most frequently used data structure, thus trading the quality of the final solution for extra performance. Since executing an interlocked instruction may be as much as 10-20 times more expensive than an ordinary

[1] We believe that fine time distinctions are not significant in our study. To approximate real time, one can keep a virtual system time incremented by one unit for every n references in the trace, where n is the number of processors. In other words, the times specified in our paper can be divided by 4 to get a rough idea of the real time.

Table 3: Proportion of shared references and unique shared data blocks when the blocksize is 4 bytes. Only *data* references to both *user and OS* are included. All numbers are in thousands.

Trace	Refs	Uniq Blks	Written	Shd Refs	Shd Blks	Shd Wrt
POPS	1518	31.1	9.4	597	19.9	4.0
THOR	1766	74.4	16.6	530	5.2	1.5
PERO	1674	14.0	4.3	136	3.4	0.8

Table 4: Proportion of shared references and unique shared data blocks when the blocksize is one word (4 bytes). Only *data* references of *user* are included. All numbers are in thousands.

Trace	Refs	Uniq Blks	Written	Shd Refs	Shd Blks	Shd Wrt
POPS	1346	29.3	9.1	576	19.8	4.0
THOR	1527	71.9	15.9	473	4.8	1.3
PERO	1528	11.6	3.8	119	3.3	0.73

instruction on some multiprocessors, a small percentage of interlocked instructions can consume a large percentage of total execution time. We also note that most of the interlocked instructions result from the user code and not from the operating system code.

4.2 Temporal and Processor Locality of User Data References

This section deals with dynamic memory access patterns and characterizes the temporal and processor locality of real-shared user data references. The first few figures plot the cumulative distributions and the frequency distributions of the time intervals between clinging and pinging references to demonstrate the temporal locality of data references. All figures use a block size of 4 bytes.

Figure 1(a) shows the cumulative frequency distribution of the time interval between clinging references to a shared block. In other words, a point (x, y) on a curve means that y references occur to a block with the time interval between these references not more than x. The corresponding frequency distribution plot for one of these programs is also shown in Figure 1(b). Due to the wide range of time intervals in which the references occur, the bins on the X-axis increase in powers of two. Therefore a bar at x with height y in the frequency plot, implies that y references occur to a block with an interval t such that $x \leq t < 2x$. For brevity we plot the frequency distributions only for THOR.

The average interval of time between accesses to the same shared block is 1165 time units in THOR. This number is unusually large because even one reference with a very large interval (or an *outlier*) can skew the average towards large values. Therefore, in the context of time intervals, a more interesting number is the median, or the time interval over which half the clinging references occur. It is easy to see that over 50% of the intervals are 25 time units or less in THOR. (The much larger average is due to the bias brought in by a few outliers.) Not surprisingly, these numbers indicate that blocks are re-referenced at small intervals of time, which is simply a reconfirmation of the fact that memory references display a high temporal locality, and is the precise reason why caching is successful. The values at 4K-8K time units form a second peak (Figure 1(b)), although the height is much smaller than the first peak at 16-32 time units. This second peak can be explained as clinging references that occur when the process resumes execution on the same processor after being switched out. The first peak, clearly, is due to references within a context switch interval. The height of the second peak is much larger in traces that show significant process migration. This low temporal locality component of clinging references introduced by process migration can be deleterious to cache performance.

These results are compared with those for pinging references, or for a reference to a block by a processor followed by a reference from another processor. Figure 2(a) shows the cumulative distribution, and Figure 2(b) the frequency distribution. The time intervals in this case are interestingly lower than for clinging references, which says that references to shared blocks by different processors are usually at least as finely interleaved as references by the same processor. Doubtlessly, the fact that our applications exploit parallelism at a fine granularity is the cause of the high temporal locality.

The small second peak at 256 time units in Figure 2(b) is due to the process migrating to another processor following a context switch. If the level of process migration is high, this peak at a large time interval can become much taller, which *falsely* suggests that process migration lowers the temporal locality of shared references. In reality, process migration simply makes a large fraction of the logically private blocks appear shared, and it is references to these shared blocks alone that give rise to the tall second peak.

Our analysis also shows that roughly a fourth of the data references are to shared data. However, a large part of the shared references need not generate bus traffic because in most multiprocessor architectures, the large number of clinging references to shared blocks (especially reads) can be treated in much the same manner as references to private blocks, in other words, blocks can be treated as private during large windows of time.

The previous figures did not distinguish between read and write references. Making this distinction is necessary because in many high-performance multiprocessor architectures, only pinging references to dirty blocks cause bus traffic when the new value of the dirty block must be somehow transmitted to the requesting processor. Figure 3 shows the distribution of the time interval between pinging references to a dirty block. The total number of pinging references to dirty blocks is far less than all the pinging references. As we shall show later in our discussion on cache consistency performance, sophisticated cache management schemes that take advantage of such features can have significant advantages over simpler schemes.

Table 5: Interlocked instruction statistics. Note the numbers are *not* in thousands.

Trace	BBSSI User	BBSSI OS	BBCCI User	BBCCI OS	OTHERS User	OTHERS OS	TOTAL User+OS	TOTAL %of-all-Irefs
POPS	9741	302	9375	301	0	0	19719	1.2%
THOR	11619	490	11600	487	0	0	24196	1.6%
PERO	109	437	109	437	0	0	1098	0.1%

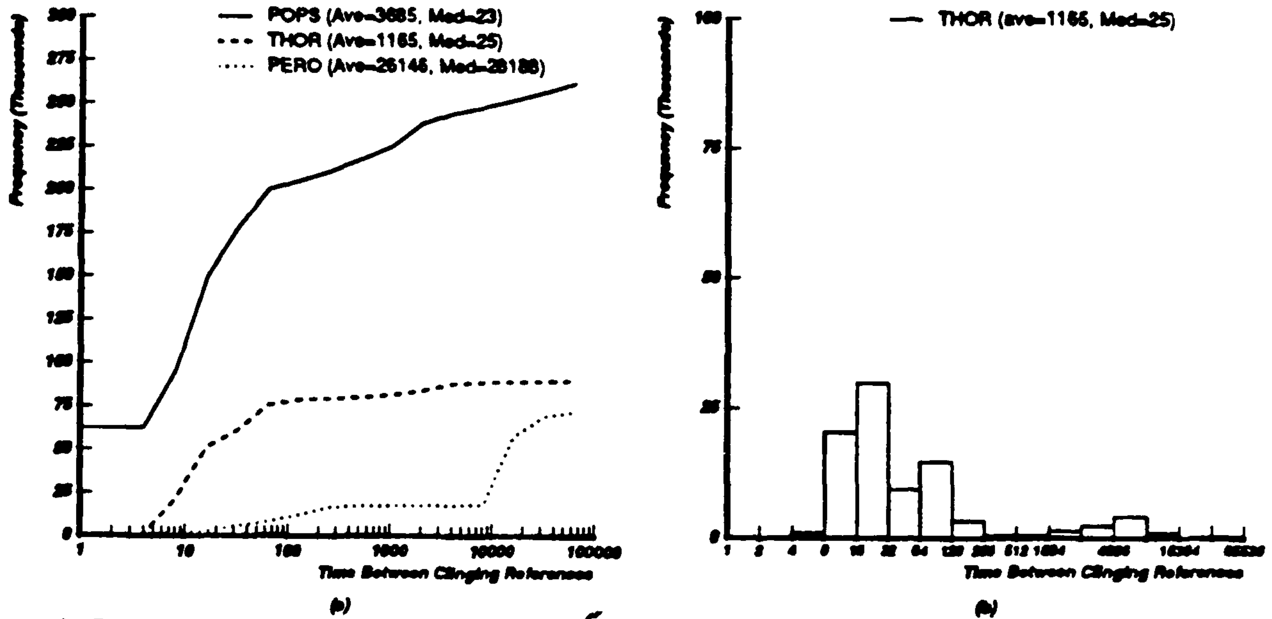


Figure 1: Distribution of the time interval between clinging references to a shared block. Only *real-shared data* references of *user* included.

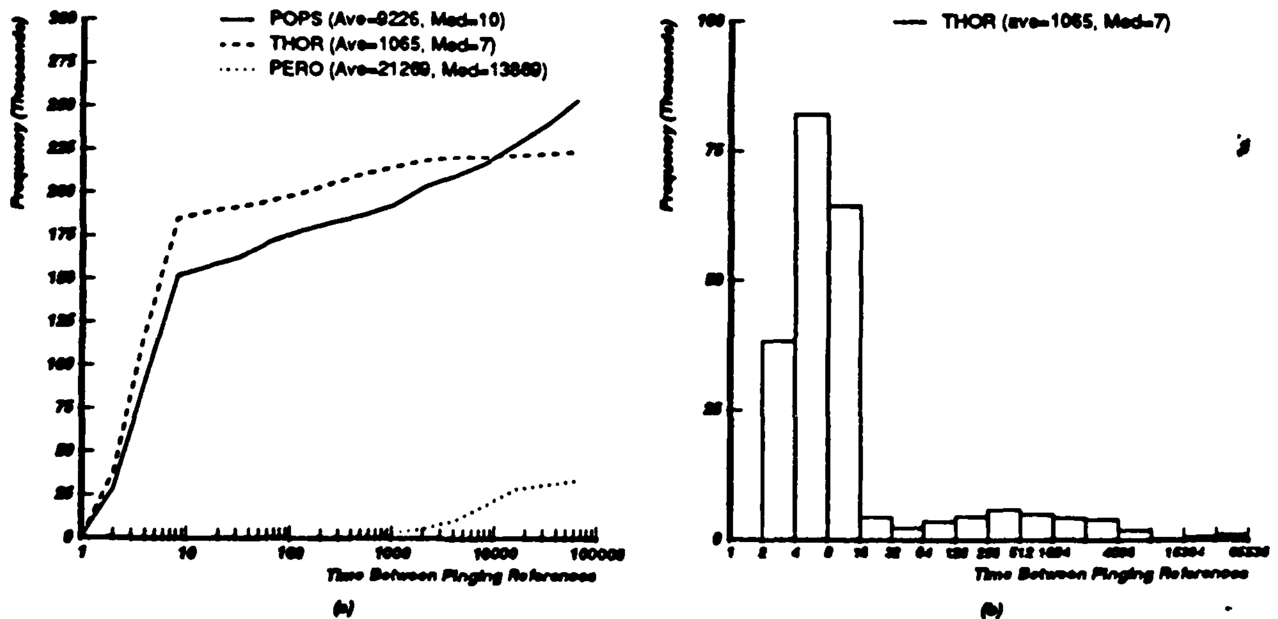


Figure 2: Distribution of the time interval between pinging references to a block. Only *real-shared data* references of *user* included.

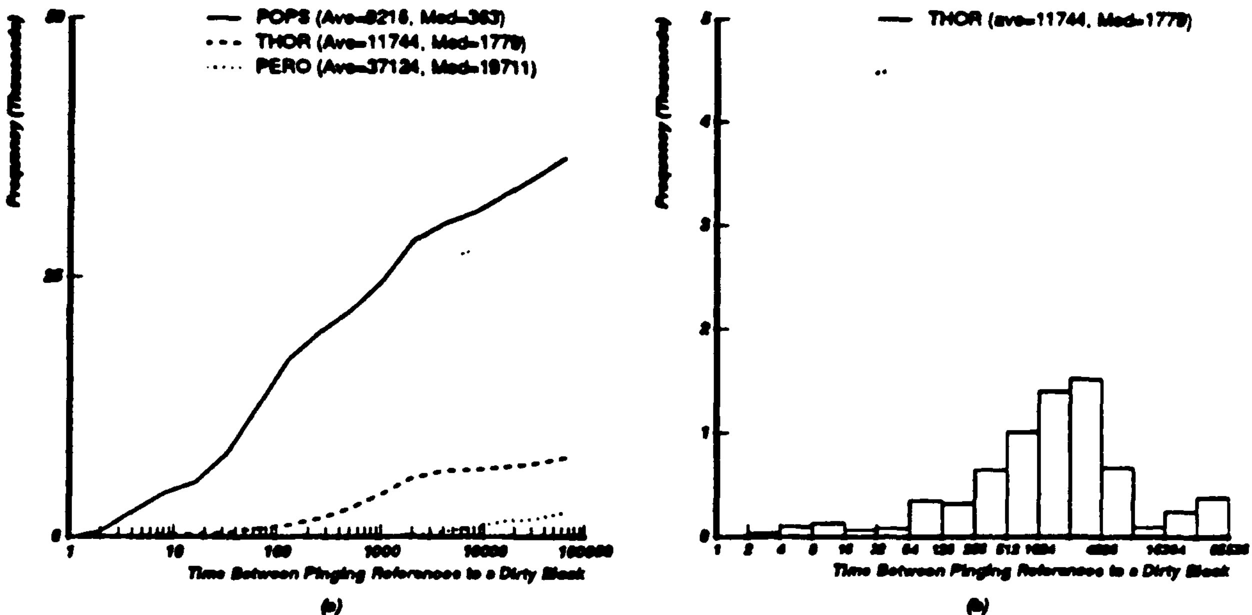


Figure 3: Distribution of the time interval between pinging references to a dirty block. Only *real-shared data* references of *user* included.

Comparing Figures 2(b) and 3(b), we see that the peak around the time interval 4-8 in Figure 2(b) is caused by reference to read-shared objects. Because Figure 3(b) does not show this early peak, we believe that references to write-shared blocks have less temporal locality than references to read-shared blocks, which benefits multiprocessor caches. A possible case is the test-and-test&set synchronization sequence, where one might expect multiple reads from several processors, but less frequent writes. The low temporal locality in pinging references to dirty blocks encourages us to believe that for large time periods blocks can be considered as private and no traffic need be generated in maintaining consistent caches.

As caches grow bigger, blocks are expected to stay in the cache for long periods of time. In such a situation, a better characterization uses the notion of processor locality. (A similar characterization has also been used in [5]). We will address processor locality in two ways. The first looks at the number of references to a block before a pinging references to it, and the second looks at the number of references to a block before a pinging reference to it, given that at least one of the references was a write. Each of the above two characterizations is pertinent to some cache consistency scheme. For example, the first one indicates the potential of a cache consistency scheme that allows only one cached copy of a block.

Figure 4(a) shows the cumulative distribution of the number of references to a block before a pinging reference, and Figure 4(b) the frequency distribution. In Figure 4(b) for THOR, there are about 200,000 pinging references to a block referenced only once by the previous processor. Unlike in the distributions of time intervals, where we used the median as a measure of temporal locality, here the average is more indicative of processor locality, because outliers represent a large number of references, and must be weighted accordingly. The low average of 1.3 indicates that interleaved references by different processors are as frequent as clinging references, implying low processor locality. We evaluated a cache consistency scheme that allowed only one cached copy of any block [6], and it performed abysmally for this very reason.

One of the chief differences between some of the snooping cache consistency schemes is the way they treat write references. One set of schemes, e.g., DRAGON [7] or FIREFLY [8], allow caches to hold valid copies of blocks that are being written into by others, and update the values on writes. Another set of schemes prefer to allow only one copy of a written block (e.g., Berkeley Ownership [9], or various flavors of directory schemes [6]). The performance of one or the other method is predicated on the locality of references to write-shared blocks, which we address next.

Figure 5 shows the number of read and write references - at least one reference a write - before a pinging reference. Several observations can be made from this figure. First, the average number of references to write-shared blocks by the same processor before a pinging reference is 5.6 for POPS, 3.6 for THOR, and 7.5 for PERO. Write references are relatively fewer than reads and contribute 1.6, 1.7, and 1.2 respectively to these averages. These averages indicate that the processor locality of shared-writable blocks is higher than that of read-shared blocks. (Recall that the corresponding numbers for all references were 1.8, 1.3, and 2.5). The higher processor locality indicates that a shared written datum is accessed multiple times by a processor before being relinquished.

A more important observation from Figure 5 is that the total number of these pings are approximately an order of magnitude lower than all pinging references, which lessens the adverse impact of the low processor locality of write references on the performance of cache consistency schemes.

As noted earlier, the average number of writes to a block before a pinging reference is small (1.7 for THOR): there are several possible reasons for this low value. We expect a low value for references caused by spinlocks. We also expect this value to be low for shared objects which move from one processor to another, with each processor making some modifications to the object. Also mostly-read-only objects are written once, and then numerous pinging *read references* are made by other processors.

Thus far, we saw that the processor locality of shared-references is moderate, with roughly 2 writes and 4 reads

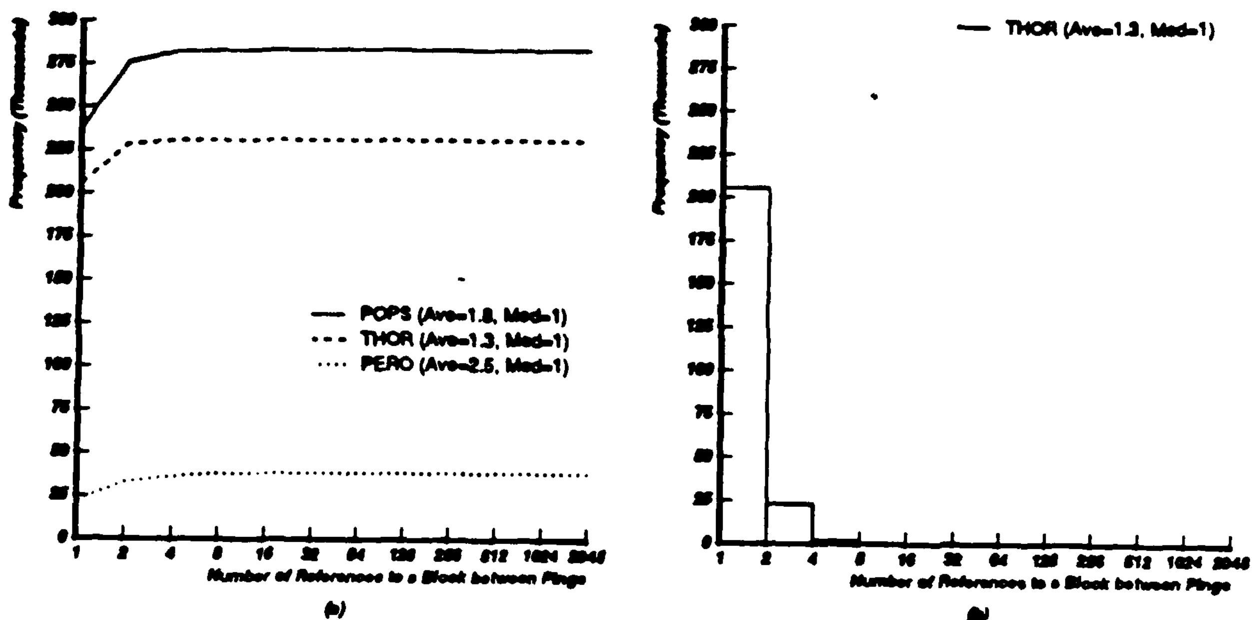


Figure 4: Distribution of the number of references to a block before a pinging reference. Only *real-shared data* references of *user* included.

on average to write-shared objects before a pinging reference. Therefore, a good cache consistency scheme must ensure effective handling of repeat read-references to shared blocks. Given the moderate processor locality of shared-data, we cannot directly determine whether invalidating cache consistency schemes such as the Berkeley Ownership protocol or directory schemes, or the updating protocols such as the Dragon and Firefly schemes are superior. More detailed evaluation that takes into account the cost of updating versus invalidating must be undertaken to make a decision.

4.2.1 Sharing Characteristics of Both User and OS References

The following discussion focuses on the sharing characteristics of both user and system references, where instruction references are excluded, as before. The general observation is that the sharing characteristics of user and system are not significantly different, although the temporal locality of shared system references was slightly lower, and the processor locality was slightly higher.

For the times between clinging references in POPS, THOR, and PERO, the medians occurred at 26, 27, and 27772 for user and system, while the corresponding numbers for user alone were 23, 25, and 28188. The times between pinging references were different by roughly the same ratio, while the times between pinging references to dirty blocks showed greater variation. The medians for user and system were 438, 2095, and 12446, as compared to 363, 1779, and 19711 for user alone.

The processor locality metrics also showed only small differences from the case of user references alone. In general, for the user and system references the average number of references to a block before a pinging reference were roughly 5% greater. A similar trend was observed for the number of references to write-shared blocks.

4.2.2 Effects of Process Migration

Since the three traces we have discussed so far do not show a significant amount of process migration, we used three other traces of the same applications that did. Due to space constraints we will only summarize our findings here and details are presented in [10].

The temporal locality of clinging references decreases if processes are rescheduled on the same processor, after having run on another processor (it will show up as a large increase in the height of the second peak in Figure 1(b)). One component of the interference caused by migration is similar to the interference caused by context switching.

Perhaps the most important effect of process migration is the significant increase in the number of blocks that get physically shared by several processors, although the logical sharing in the program might be much smaller. For instance, the fraction of references to shared data blocks increases from 0.2 to 0.9 with process migration. Due to the typically long intervals between process switches (thousands of references), the time interval between pinging references to these shared blocks is very large, and causes a much larger second peak in Figure 2(b). Similarly, the average number of references to a block – at least one reference being a write – before a pinging reference is 13 with process migration and less than 2 without. This perceived decrease in the temporal locality and the increase in processor locality of shared references stems from the fact that many of these references are to logically private data objects that are not referenced by other processors until the process actually migrates to another processor.

In summary, although process migration increases the processor locality and decreases the temporal locality of shared blocks, it increases the total number of shared blocks substantially, and potentially impacts both intrinsic cache performance, and the performance of cache consistency schemes adversely.

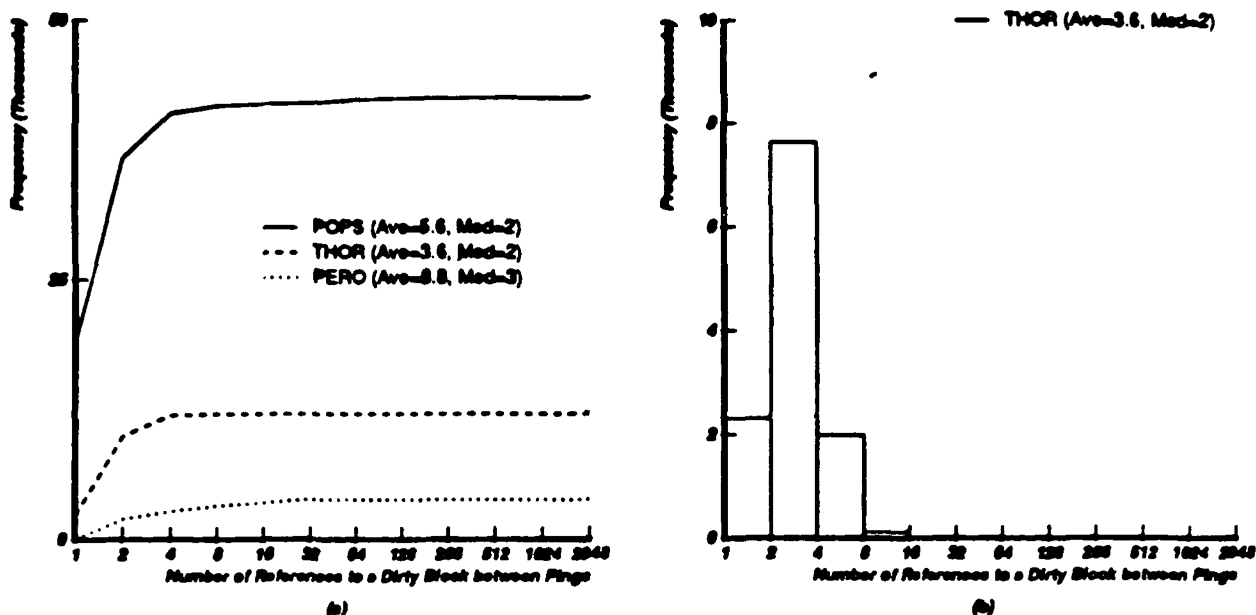


Figure 5: Distribution of the number of references to a block before a pinging reference to the same block, given that at least one reference was a write. Only *real-shared data* references of *user* are included.

4.3 Cache Consistency Implications

The memory reference traces also yield useful insights about the effectiveness of various cache consistency schemes. For example, they enable an accurate determination of the traffic caused on a shared bus by any given cache consistency scheme under realistic load conditions. While a detailed analysis of the numerous cache consistency schemes proposed in literature [11,9,7,12,8] would be interesting, it is beyond the scope of this paper. Instead, we consider one representative each from the write-through with invalidate, write-back with invalidate, and write-back with update classes of cache coherence schemes. To help explain the various phenomena observed here, we use the data presented in earlier sections. As before we assume infinite caches, and unless otherwise stated, block size is one word (or four bytes).

The first scheme discussed in this paper is *write-through with invalidate* (WTI) commonly used in commercial multiprocessors. In this scheme, every write from a processor accesses the bus both to update main memory and to invalidate that location in other caches. Examples of *write-back with invalidate* schemes are Goodman's write-once [11], Rudolph and Segall's scheme [12], Berkeley Ownership [9], and the directory scheme [13]. We consider write-once as the second scheme in this paper. In this scheme, the first write to a location uses the bus to update main memory and to invalidate that location in other caches. Subsequent writes to that location by the same processor do not result in any bus traffic, as that location is now owned locally. This scheme is labeled WBI in the following discussion to indicate the class it belongs to. Examples of the *write-back with update* schemes are Dragon [7] and Firefly [8]. We use Dragon as the third scheme, and denote it WBU. In the Dragon scheme, all writes to a shared location (a location present in multiple caches) result in a bus access to update the value of that location in other caches. For non-shared locations, the cache acts like a regular uniprocessor write-back cache.

We evaluate the performance of the above three cache coherence schemes in terms of the *bus transactions* generated on a shared-memory multiprocessor. We distinguish between three kinds of bus transactions: *block transfers*, *updates*, and *invalidations*. A block transfer transaction transfers a block from memory to cache or vice versa. For example, a block transfer into a cache on a read miss. An update transaction updates the contents of a location either in main memory (e.g., on a processor write in WTI) or in a remote cache (e.g., on a write to a shared location in WBU). The update transfers only one word, and is hence cheaper than a block transfer with a large block size. A processor uses an invalidation to purge cache blocks in other caches to get exclusive ownership of the block. No data transfer is required for this transaction, only the address of the cache block to be invalidated need be specified. Note that block transfers and updates can simultaneously serve as invalidation transactions, and this is usually exploited in most coherence schemes.

Table 6 presents the event frequencies for the three traces as a function of the cache coherence strategy. Because of our interest in characteristics of shared references, we only include cpu-shared user data references for POPS, THOR, and PERO (see Table 4 for details). Because caches are infinite, a data item brought into the cache remains there until invalidated. From Table 6 we derive the total number of block transfer transactions and update transactions that would occur in a multiprocessor and present the numbers in Table 7. The table also presents data for 16-byte and 64-byte blocks to study spatial locality in shared references.

We first examine Table 7 for 4-byte blocks. Comparing total number of transactions, the WTI scheme is worse than both WBI and WBU. WTI looses to WBI because of the processor locality displayed by write references, as shown in Figure 5. While every write generates bus traffic in WTI, clinging write references do not cause bus traffic in WBI. Comparing WTI and WBU, both schemes generate an update transaction for every write to a shared location. However, WBU saves about 25% updates because before the point that a location becomes shared (a second processor requests it), only the first read or write produces a bus transaction. WBU also has fewer block transfers because, unlike WTI, it never invalidates a location from a cache. The details of the events are in Table 6.

Table 6: Events, bus transactions, and event frequencies. Each event is a triple: event-type (read-miss, write-miss, write-hit), state in local cache (not present, clean, dirty), and state in remote cache (not present, clean, dirty). We use abbreviations *d* for block transfer, *u* for update, and *i* for invalidate. Only cpu-shared user data references are considered. All numbers are in thousands.

Event	Bus Transactions			POPS			THOR			PERO		
Type	WTI	WBI	WBU	WTI	WBI	WBU	WTI	WBI	WBU	WTI	WBI	WBU
total refs	-	-	-	575.6	575.6	575.6	473.1	473.1	473.1	119.0	119.0	119.0
read-hits (rh)	-	-	-	429.5	429.5	451.8	416.3	416.3	423.8	102.3	102.3	105.3
read-misses (rm)												
rm-np-np	1*d*	1*d*	1*d*	10.01	10.01	10.01	2.55	2.55	2.55	3.20	3.20	3.20
rm-np-cl	1*d*	1*d*	1*d*	59.24	25.11	13.46	14.21	2.14	0.54	7.13	3.57	2.53
rm-np-di	-	1*d*	1*d*	-	34.13	23.52	-	12.06	6.18	-	3.56	1.57
write-misses (wm)												
wm-np-np	1*d*, 1*u*	1*d*	1*d*	9.72	9.72	9.72	2.28	2.28	2.28	0.08	0.08	0.08
wm-np-cl	1*d*, 1*u*	1*d*	1*d*, 1*u*	12.81	4.39	1.74	0.12	0.01	0.00	0.38	0.10	0.10
wm-np-di	-	1*d*	1*d*, 1*u*	-	8.42	1.90	-	0.11	0.03	-	0.28	0.15
write-hits (wh)												
wh-cl-np	1*u*	1*u*	0	7.64	2.14	2.14	7.39	2.15	2.15	1.38	1.08	1.08
wh-cl-cl	1*u*	1*u*	1*u*	46.63	22.12	1.35	30.27	9.00	0.16	4.53	3.33	1.06
wh-cl-di	-	-	1*u*	-	-	23.87	-	-	8.75	-	-	1.95
wh-di-np	-	0	0	-	30.01	5.50	-	26.50	5.24	-	1.49	0.29
wh-di-cl	-	-	1*u*	-	-	30.58	-	-	21.45	-	-	1.65

Dividing the total number of bus transactions generated by all three programs for the WBI scheme in Table 7 (161.6K) by the total number of references that resulted in these transactions (1168.7K), we see that there are approximately 0.138 bus transactions generated per reference. This number appears quite large given infinite caches, and there are two reasons for this. First, this data represents only cpu-shared user data references, which show poor processor locality as in Figure 4, or equivalently, which display a high temporal locality of pinging references as in Figure 2). Consequently they do not benefit much from the read-sharing allowed by the WBI scheme. If one includes both user and OS references, and both data and instructions, then the number of transactions per reference falls to 0.031, which is much better. This reduction is primarily due to the large number of read-shared references generated by instruction fetches. (Consequently, allowing read sharing for instructions is crucial in multiprocessor caches.) The second reason for the high value is that block size is 4 bytes. When the block size is increased to 16 bytes, the number of transactions per reference drops down further to 0.016, primarily due to the high spatial locality of instruction fetch references.

In general, two opposing forces come into play as the block size is increased - one trying to decrease the number of transactions and the other trying to increase them. As the block size is increased the number of bus transactions is reduced because the bus access or invalidation cost is amortized over several words. Contrarily, a large block size increases the probability of unrelated objects residing in the same block, and a write to one object can unnecessarily invalidate an active unrelated object in a remote cache.

To study the spatial locality characteristics of cpu-shared user data references, we now examine the bus transactions generated by WBI in Table 7 as the block size is increased. For POPS the number of block transfers decreases from 91.86K to 47.15K to 46.23K as the block size is increased from 4 to 16 to 64 bytes. This indicates that there is high spatial locality at 16-bytes, with little cache interference due to coresiding unrelated objects. Beyond 16 bytes, either there is no spatial locality or the cache interference neutralizes the benefits due to locality. THOR behaves differently. When the block size is increased from 4 to 16 bytes, the number of block transfers increases by a factor of 1.5. This indicates that negative cache interference effects dominate.[2] In contrast to POPS and THOR, increasing block size has a very positive effect on PERO. The number of block transfers decrease by a factor of 2 as the block size is increased from 4 to 16 bytes, and further by a factor of 3.4 when the block size is increased from 16 to 64 bytes. The number of update transactions decreases steadily too. Thus the PERO program appears to have high spatial locality with almost no cache interference.

Another interesting result that can be observed by examining the total traffic lines in Table 7 is that for shared data references the total bus bandwidth required is minimized when block size is 4 bytes and increases as the block size is increased. This result is in start contrast to uniprocessor caches, where the optimal block size tends to be much larger. The only exception is the PERO program when block size equals 64 bytes.

We were interested in estimating the effects of obviating broadcasts in cache consistency schemes to enable scalability. Table 8 presents the number of caches in which blocks are actually invalidated, whenever a reference that could potentially invalidate other caches is processed in the WBI scheme. Such references for the WBI scheme are all write misses and all write-hits to a clean location in the local cache. The total number of such references is given in column three. The inv-0 column gives the number of potentially invalidating references that resulted in no actual invalidations, the inv-1 column gives the number of such references that resulted in exactly one invalidation, the inv-2 column gives the number that resulted in an invalidation in two other caches, and the inv-3 column denotes an invalidations in three other caches. Since all the traces are four-processor traces, no reference can result in invalidation in more than three other caches.

[2] Another factor contributing to the increased number of block transfers is the fact that as block size is increased, the number of cpu-shared references also increases.

Table 7: Bus transactions. Only cpu-shared data references of user are included. All numbers are in thousands.

Bus Transactions	POPS			THOR			PERO		
	WTI	WBI	WBU	WTI	WBI	WBU	WTI	WBI	WBU
Block-Size = 4 bytes									
block-xfers (d)	91.86	91.86	60.42	19.15	19.15	11.58	10.79	10.79	7.63
updates (u)	76.79	24.25	59.43	40.06	11.15	30.38	6.37	4.42	4.91
Total Traffic ($d + u$)	168.65	116.11	119.85	59.21	30.30	41.96	17.16	15.21	12.54
Block-Size = 16 bytes									
block-xfers (d)	47.15	47.15	22.97	29.77	29.77	20.27	5.05	5.05	3.44
updates (u)	78.47	15.04	61.48	49.39	12.38	34.02	6.57	2.14	5.26
Total Traffic ($4d + u$)	267.07	203.64	153.36	168.47	131.46	115.10	26.77	22.34	19.02
Block-Size = 64 bytes									
block-xfers (d)	46.23	46.23	9.30	29.75	29.75	16.68	1.50	1.50	0.94
updates (u)	79.39	20.17	65.09	86.99	16.61	73.15	6.95	0.70	5.61
Total Traffic ($16d + u$)	449.23	390.01	139.49	324.99	254.61	206.59	18.95	12.70	13.13

We would like to remark on two aspects of the data presented in Table 8: the fraction of references that invalidate multiple caches as compared to those that invalidate only one cache, and the effect of changing the cache block size. Let us examine the first aspect. The data for 4-byte blocks indicates that the fraction of references that cause invalidations in three caches (1.3%) is quite small compared to the fraction that cause invalidations in one cache (61.0%).[3] It is interesting to speculate if this phenomenon - that on an invalidate transaction, with high probability, data in only one or very few caches needs to be invalidated - is true even when the number of processors is large. If it is true, then instead of building broadcast-based cache consistency mechanisms, one can build message-based mechanisms where the invalidation message is sent only to those caches that contain that data. The resulting reduction in bandwidth requirements makes it possible to build scalable shared-memory multiprocessors. In the following paragraphs, we speculate why the above result should also hold for a larger number of processors.

There are three kinds of data objects in parallel programs: (i) non-shared, (ii) read-shared, and (iii) write-shared objects. The non-shared objects normally do not cause any invalidations except due to process migration, in which case all the invalidations go only to the processor that previously ran that process. The read-shared objects also do not cause any invalidations. So the multiple cache invalidations come from write-shared objects. We now explore some common ways in which write-shared objects are used in parallel programs.

The first common use of write-shared objects is as spin locks or other similar synchronization related structures. Let us consider the spin lock as the typical case. If the spin lock is implemented in a straightforward way using an interlocked test&set instruction, since the instruction ends in a write, at the end of each instruction only one cache contains the data, and only one cache has to be invalidated on a subsequent reference by a different processor. If the spin lock is implemented using a test-and-test&set instruction,[4] then with some probability the lock will be present in multiple caches. When the lock is set free by writing into it, these multiple caches have to be invalidated. However, if the program is "reasonable" (i.e., there is no excessive contention for the locked object), then either the lock will not have too many processes waiting on it and thus only one or a few caches will need to be invalidated, or such an occurrence will be very rare, and the probability of invalidating many caches will be very small.

The second common use of write shared objects is as mostly-read-only objects. An example is multiple programs sharing a database that is occasionally modified. By occasionally we mean that relative to the number of references made to that object, the number of writes is small. On a write to a mostly-read-only object, multiple caches may have to be invalidated, but since writes are rare, the overall fraction of multiple cache invalidations still stays low. The third common use of write-shared objects is where one process works on an object for some time, then another process, and so on. Shared objects protected by locks often behave this way. In this third case, when one process is working on an object, that object resides in the cache of the associated processor. When that object moves to another process (and possibly to another processor), the cache entries in the previous processor are invalidated, but that corresponds to invalidation in only one other cache. So it is still consistent with our conjecture that in larger multiprocessors invalidations will happen in only one or in a very small number of other caches with high probability. The above observations suggest the use of a message-based cache consistency protocol, instead of a broadcast-based protocol. We are analyzing this issue in detail and results will be presented in a future paper.

We now look at the effect of increasing the cache block size on the number of invalidations. The fraction of references that cause invalidations in multiple caches increases with block size. As an example, for POPS, consider dividing the entries in the inv-3 column by corresponding entries in the total column in Table 8. The numbers we get are 2.1%, 4.6%, and 6.2% respectively. The primary reason for this phenomenon is that as block size is increased, unrelated data objects fall into the same cache block. Multiple processors accessing these distinct objects cache the same block, and a subsequent write results in an invalidation in multiple caches.

5 Summary and Conclusions

We have presented data characterizing the memory reference patterns in shared-memory multiprocessors. Our data is based on traces obtained for three applications from a 4-

[3] The reason why this ratio is smaller for POPS and THOR for larger block sizes is discussed later.

[4] In a test-and-test&set instruction, if the first test fails we simply loop back and do not execute the test&set part of the instruction.

Table 8: Cache invalidation statistics for the WBI coherence scheme. Only user cpu-shared data references are included. All numbers are in thousands.

Trace	B	total	inv-0	inv-1	inv-2	inv-3
POPS	4	46.77	11.85	29.24	4.69	0.99
	16	27.06	3.89	18.51	3.42	1.24
	64	30.18	1.33	20.07	6.92	1.86
THOR	4	13.55	4.43	8.97	0.13	0.02
	16	14.72	5.11	8.69	0.74	0.18
	64	18.06	3.28	13.72	0.94	0.12
PERO	4	4.87	1.16	2.65	0.98	0.08
	16	2.18	0.47	1.17	0.50	0.04
	64	0.72	0.14	0.42	0.14	0.02

processor VAX 8350 using the ATUM address tracing technique. The traces used are "complete", in that they contain information about both system and user references, references due to interrupts, process scheduling, etc.

Our analyses shows that a large fraction (about one-fourth) of references in the traces are to shared objects. These shared references display a significant amount of temporal locality, and only a small amount of processor locality for both read and write references. For example, the average number of reads and writes to a write-shared block before a remote reference (a ping, which may possibly invalidate the data) are 4 and 2 respectively. Nevertheless, caching shared data is still highly useful because of the significant amount of read sharing.

We also present statistics about the use of interlocked instructions. The traces show that 0.1%-1.6% of instruction references are to interlocked instructions, and that most of these instructions references are from user code. The paper also touches on the effects of process migration. Process migration causes a large number of logically unshared references to become shared references with respect to the cache system.

The nature of shared-memory reference patterns also yields insight on how various cache consistency schemes will perform. We present the analysis for three classes of cache consistency schemes - write-through with invalidate (WTI), write-back with invalidate (WBI), and write-back with update (WBU). For shared data references, WTI performs worse than both WBI and WBU as it uses the bus on every write. Comparing WBI and WBU, the former seems to have an edge for 4-byte blocks, while WBU does better for 16-byte and 64-byte blocks. Another surprising result that we observed for shared data references is that the total bus bandwidth required is minimized when block size is 4 bytes and increases as the block size is increased. Our traces also show that when a reference that could possibly invalidate a cache is processed, with a very high probability (61.0 %) it invalidates only one other cache. The probability of causing an invalidation in all three caches is only 1.3%. We discuss why this should also be true for multiprocessors with larger number of processors, and suggest the use of message-based cache consistency schemes rather than broadcast-based cache consistency schemes.

6 Acknowledgements

Several people have helped us in obtaining the traces. We to thank Roberto Bisiani and the Speech Group at CMU for letting us use their VAX 8350. Dick Sites at Digital Equipment Corporation, Hudson, made multiprocessor ATUM possible, and Digital Equipment Corporation made the ATUM microcode available for our use. Larry Soule and Helen Davis at Stanford helped with the THOR program and Jonathan Rose with PERO. Finally, many ideas presented in this paper came up during discussions with Susan Eggers, Mark Horowitz, John Hennessy, and Rich Simoni. We appreciate their contributions. The research reported in this paper was funded by DARPA contract MDA903-83-C-0335. Anoop Gupta is also supported by a faculty development award from DEC.

References

[1] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119-127, June 1986.

[2] F. Darema-Rogers, G. F. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *Proceedings of the 1987 ACM SIGMETRICS Conference*, pages 46-58, May 1987.

[3] Anoop Gupta, Charles Forgy, and Robert Wedig. Parallel architectures and algorithms for rule-based systems. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, June 1986.

[4] Jonathan Rose. *LocusRoute: A Parallel Global Router for Standard Cells*. Technical Report, Computer Systems Laboratory, Stanford University, 1987.

[5] Susan J. Eggers and Randy H. Katz. *A Characterization of Sharing in Parallel Programs and its Applicability to Coherency Protocol Evaluation*. EECS Department, UC Berkeley, October 1987.

[6] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. *Scalable Directory Schemes for Cache Coherence*. Computer Systems Laboratory, Stanford University, October 1987. Submitted for publication.

[7] E. McCreight. *The Dragon Computer System: An Early Overview*. Technical Report, Xerox Corp., September 1984.

[8] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164-172, October 1987.

[9] R. H. Katz et al. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276-283, June 1985.

[10] Anant Agarwal and Anoop Gupta. *Memory-Reference Characteristics of Multiprocessor Applications under MACH*. Computer Systems Laboratory, Stanford University, February 1988.

[11] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124-131, June 1983.

[12] L. Rudolph and Z. Segall. Dynamic decentralized cache consistency schemes for mimd parallel processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 340-347, June 1985.

[13] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, c-27(12):1112-1118, Dec. 1978.

Competitive Management of Distributed Shared Memory

David L. Black
Carnegie-Mellon University
Pittsburgh, PA 15213

Anoop Gupta and Wolf-Dietrich Weber
Stanford University
Stanford, CA 94305

Abstract

This paper presents and analyzes algorithms for managing the distributed shared memory present in non-uniform memory access multiprocessors and related systems. The competitive properties of these algorithms guarantee that their performance is within a small constant factor of optimal even though they make no use of any information about memory reference patterns. Both hardware and software implementation concerns are covered. A case study of the Mach operating system indicates that integration of these algorithms into operating systems does not pose major problems. On the other hand, hardware support is required to obtain the full functionality of the algorithms. We also sketch possible algorithm extensions to additional hardware architectures and software programming models.

Trace driven simulations are used to evaluate our approach and compare it to other alternatives. Speedups of 5 to 10 over random assignment of pages on production applications are achieved without modifying the applications for *non-uniform memory access* (NUMA) architectures. We compare our proposed hardware support with the more aggressive approach of fully-consistent caches. An additional factor of 2 to 3 in performance can be obtained from the cache approach, but at the cost of much more hardware. These results indicate that the algorithms and their hardware support may represent a viable cost/performance tradeoff.

1 Introduction

The widespread use of *uniform memory access* (UMA) multiprocessors has sparked interest in using uniform shared memory programming models on *non-uniform memory access* (NUMA) multiprocessors. Use of a common programming model enhances the portability of applications among such machines, and can reduce the effort required to fit or tune applications to NUMA multiprocessors. New techniques are required to manage the distributed physical memory found in a NUMA multiprocessor because the location of memory used by an application (with respect to the processor(s) executing the application) directly affects performance. Optimizing the use of physical memory to minimize access costs is a major issue that must be faced by any implementation of a shared memory programming model on such machines. This paper presents techniques and algorithms for this problem, along with preliminary performance results from trace-driven simulations.

Our algorithms are competitive in a strict theoretic sense. An informal statement of this property is that the algorithms are essentially the best that can be achieved in the absence of information about future memory reference behavior. The techniques of competitive algorithm analysis are particularly useful for this work because they explicitly address the constant factors ignored by standard complexity analysis, and because they are well-suited to the analysis of resource management problems. Previous work has developed competitive algorithms for the related problems of optimizing the use of snoopy caches [8].

The performance results for these algorithms are based on trace-driven simulations of several production applications from UMA multiprocessors. These results show that the proposed algorithms attain total speedups of 5 to 10 over random assignment of pages. This indicates that significant locality (both code and data) may exist in a large class of multiprocessor applications, and that this locality can be detected and exploited automatically. As a result such applications may not require extensive design changes or modifications for use on NUMA multiprocessors; no such changes or modifications were made to our applications.

This paper concentrates on the application aspects of our work. Proofs of the competitive properties of the algorithms can be found in [2]. The next section presents a basic model that covers the systems to which our algorithms are applicable. This is followed by an introduction to competitive algorithms. Section 4 breaks down the basic problem and presents our competitive algorithms for solving it. Sections 5 and 6 continue with a discussion of implementation concerns including the difficulties imposed by most current hardware. Section 7 presents our performance results from trace driven simulations. Sections 8, 9, and 10 briefly discuss extensions of this work. Sections 11 and 12 conclude the paper with a review of related work and a short summary of results.

2 Basic Model

This section presents the basic memory model for which our algorithms were developed. We assume an idealized machine composed of processor-memory clusters, with physical memory divided entirely among the clusters. A processor-memory cluster consists of one or more processors with local memory that is equally accessible (in terms of latency) to all processors. Our idealized machine has two distinct memory access latencies; the latency to access memory in the same cluster, and a significantly larger latency to access memory in another cluster. As a result all memory within a single cluster is equivalent, and all processors within a cluster have identical memory access characteristics (latency in terms of the cluster in which the accessed memory is located). Finally all memory locations outside the cluster have the same access latency from any processor in the cluster.

This basic model subdivides memory into pages and pages into locations. Pages are the fundamental unit of memory management; locations are the fundamental unit of memory access. We assume the existence of virtual memory map-

To appear in COMPCON '89

ping mechanisms, and therefore distinguish between virtual pages (in the address space of some program or the operating system) and physical pages (actual memory in the clusters). Mapping virtual pages to physical pages is one of the responsibilities of a memory management facility. Sharing may result in more than one virtual page in one or more address spaces being mapped to the same physical page. The page size used by our algorithms can be no smaller than the hardware page size if mapping is used, but it may be a multiple thereof.

We normalize our model by assuming the difference in cost between an in-cluster memory access and a remote-cluster memory access is 1; this cost includes the effects of both increased latency and use of interconnection bandwidth. This cost only applies to accesses that actually use the interconnection network; if caches are present at the processors, we only consider accesses that miss in or bypass the appropriate cache. In addition, we are assuming that read and write costs are identical; all of our work generalizes to cases in which these costs are not identical.

This model permits us to analyze techniques for managing the performance impact of distribution in a shared memory system. We concentrate on two major tools for this management: replication and migration of virtual memory. Replication consists of making a copy of a virtual page in another cluster and updating mappings that benefit from this copy (in reduced access time). Migration consists of moving a virtual page from one cluster to another and updating all mappings to that page. We formalize the costs of replication and migration as r and m respectively in terms of access costs. These costs include latency and overhead components, but do not include the additional costs of allocating a physical page in a cluster with a page shortage (i.e. causing pageout) or the additional benefits of freeing a physical page in such a cluster (i.e. avoiding pageout). We separate the issues involved in page reclaim from migration and replication; these are addressed in section 5.1.

Our basic model applies to any machine that can implement NUMA memory. This includes NUMA machines that implement the model directly (e.g. Butterfly [5]), *no remote memory access* (NORMA) machines with uniform access costs, and network shared memory implementations on networks with uniform communication costs. For the last two classes of the machines, it is essential that the system (hardware and/or software) support access forwarding so that accesses to pages that are not in local memory can be satisfied at remote memory *without* moving the entire page to local memory (an expensive operation). Most current NORMA machines (e.g. hypercubes) and network shared memory implementations [4,9,21] do not support this functionality.

3 Basic Problem

The problem we address here is the management of distributed shared memory in architectures conforming to our model. For architectures utilizing a single copy of the operating system (NUMA multiprocessors), this includes not only memory shared explicitly, but also memory shared implicitly via copy-on-write techniques. Since we rely on replication and migration to perform this management, the problem can be restated as "When and under what circumstance should (virtual) pages be replicated into or migrated to memory in other clusters?"

There is a significant difference between this problem and the related problem of snoopy caching: our model and its realizations do not have broadcast, invalidate, or snooping mechanisms that can maintain consistency among multiple copies of a virtual page when writes occur. This prohibits replication of writable pages. Because we have separated the issue of page reclamation, migration of read-only pages make little sense: replication is less costly, and provides the benefits of local access to two clusters instead of one. As a result the overall problem splits into two sub-problems:

- Replication of read-only pages.
- Migration of writable pages.

If a virtual page is both read-only and writable at different times during the execution of an application, we consider each segment (read-only or read/write) of the page's existence to be a separate instance of the corresponding problem.

4 Basic Algorithms

Effective use of replication and migration presents an enigma. Replicating or migrating a page that will never be referenced again is very costly, but so is failing to replicate or migrate a page that will be used heavily in a remote cluster. Avoiding these situations seems to require knowledge of the future that is not available when decisions must be made; this results in a situation where any decision about replication or migration could be both wrong and costly. Problems that require these decisions to be made (affected by future system behavior, but must be made without any knowledge about this behavior) and algorithms that make these decisions are called *on-line*.

Results obtained from the analysis of competitive algorithms provide a solution to this enigma. An on-line algorithm is called *competitive* if its cumulative cost on any sequence is within a constant factor of the cost of the optimal algorithm[1] on the same sequence, and no such algorithm exists for any smaller constant. Competitive algorithms have been found for a number of problems, including list management [16], snoopy caching [8], and some server problems [10]. This paper extends past work by presenting competitive algorithms for replication and migration of distributed shared memory.

4.1 Replication

The on-line replication problem consists of determining when in a sequence of accesses a page should be replicated into other clusters, without look-ahead. Under our model all clusters are uniformly equidistant; if a page is not resident locally, the cost to access it does not depend on the cluster in which it is accessed. As a result the decision to replicate a page into a given cluster is independent of the decisions to replicate into any other clusters. Hence the general replication problem reduces to the replication problem for two clusters with the page initially resident in only one cluster. Algorithm **R** is our algorithm for this problem.

Algorithm R:

> Count remote accesses from the cluster that does not have the page. When this count exceeds the replication cost, r, replicate the page into the cluster.

[1] The optimal algorithm may look at the entire sequence before making any decisions

Results:

1. Any on-line algorithm for this problem must have a cost that is at least twice the cost incurred by an optimal off-line algorithm on some sequence of accesses.
2. Algorithm **R** is competitive, i.e. its cost is always within a factor of two of optimal on any sequence of accesses.

Algorithm **R** (and algorithm **M** to be presented later) are algorithms that perform well across the entire spectrum of possible sequences. If the specific sequence that will occur is known in advance, an on-line algorithm can be constructed that performs well on that particular sequence, but will perform worse than our algorithm on many other sequences. This embodies the optimality property of our competitive algorithms: they are essentially the best possible in the absence of knowledge about what will happen in the future.

4.2 Migration

The on-line migration problem consists of determining when in a sequence of accesses a page should be migrated to another cluster without look-ahead. Unlike the replication problem, migration depends on the number of clusters: of all the clusters that would benefit from having the page, only one can actually have the page. Decisions to migrate different pages are still independent, so the migration problem reduces to migration of a single page in response to accesses to that page. Algorithm M is our algorithm for this multiple cluster page migration problem.

Algorithm M:

> Associate a counter with each cluster: initialize the counts to zero. Access from a cluster that does not have the page increments that cluster's counter, and decrements some other cluster's counter, but not to less than zero. When a cluster's counter reaches twice the migration cost (i.e. $2m$) migrate the page to that cluster and zero its counter. Access from a cluster that has the page decrements some other cluster's counter, but not to less than zero.

All of the counters for a page will be zero after a migration due to the way they are maintained by algorithm M.

Results:

1. Any on-line algorithm for this problem must have a cost that is at least three times the cost incurred by an optimal off-line algorithm on some sequence of accesses.
2. Algorithm M is competitive, i.e. its cost is always within a factor of three of optimal on any sequence of accesses.

5 Operating Systems Issues

There are two sets of operating systems issues that must be addressed in implementing our algorithms: (i) how do we take into account the limited size of physical memory: and (ii) what are the interactions between the proposed algorithms and the memory management portion of an operating system. The second issue arises primarily if the algorithms are implemented in the operating system kernel: this is an attractive choice both because it permits direct access to mapping information and also because it makes the resulting benefits available to all applications on the system, instead of just those that are modified to use our algorithms.

5.1 Limited Physical Memory Size

Since there are many other demands on physical memory besides those generated by replication and migration (e.g. memory allocation, file mapping, internal use by the operating system, etc.), extending the replication and migration algorithms to control memory usage is not appropriate. We believe that the operating system should separate reuse of physical memory (pageout or page reclaim) from replication and migration issues. Even the fallback position of dedicating a fixed amount of physical memory to replication/migration and managing that is not a good idea: this prevents reallocation of memory to the uses for which it is in greatest demand.

We propose the use of independent pageout daemons for the management of various cluster memory pools. These daemons can respond appropriately to the potentially different memory demands from cluster to cluster. Any of several standard paging algorithms can be used to implement the daemons [18]. The migration and replication costs can be dynamically modified to feed information about page availability back into the replication and migration algorithms. These modifications should be restricted to increasing costs above their basic levels to reflect page shortages and hence discourage future use of memory in clusters with page shortages. Decreasing migration costs to encourage freeing memory in clusters with shortages, and cost-based reclamation of replicates are fraught with potential danger: this is because not all system components that use memory are or can be sensitive to costs – hence these cost-driven alternatives may result in heavily used pages being evicted in order to retain lightly used ones for cost-insensitive components.

5.2 Memory Management Interactions

Algorithms M and R can be incorporated into the operating system's memory management code on a NUMA multiprocessor. Implementing these algorithms inside the operating system allows their benefits to accrue to all uses of the machine, but also results in interactions with other memory management functions that must be dealt with as part of the implementation.

We use the virtual memory management portion of the Mach operating system [12] as a base for a case study of these interactions. Mach is a multiprocessor operating system developed at Carnegie-Mellon University; its VM system provides advanced memory management functionality including flexible sharing (both read/write and virtual copy), mapped files, and external memory management. This functionality stresses the interactions of our algorithms with the remainder of the operating system, and serves to expose potential problems.

The Mach VM implementation is cleanly split into machine-independent and machine-dependent portions. The machine-dependent portion consists of a single module, the *pmap* module, that is responsible for all physical map operations. The machine-independent portion of the system associates a pmap with each address space and invokes the pmap module as needed to perform mapping operations. Mach supports parallel execution of multiple threads within a single task's address space: this parallel execution can result in a

single pmap being used simultaneously by more than one processor.

Mach envisions support for non-uniform physical memory by adding a NUMA layer between the machine-independent and machine-dependent portions of the VM system [18]. This layer hides the non-uniformity of the memory structure by translating *logical pages* (manipulated by the machine-independent portion of the VM system) to physical pages (in the hardware) in order to implement architecture-specific memory management policies (e.g. replication, migration). A similar translation process is needed for pmaps to allow replication within a single address space if its threads are spread across multiple clusters; in this case each cluster would have its own physical map, but the collection of these pmaps would appear as a single logical pmap to the machine-independent portion of the VM system. This adds additional complexity to the NUMA layer to better support multi-threaded applications, and may complicate interfaces that allow users to modify replication and migration behavior because an address space no longer uniquely specifies a cluster.

There are two other minor interactions of the NUMA layer with the remainder of the Mach VM system, and one major interaction. The two minor interactions are:

- Pageout functionality must be moved into the NUMA layer and redesigned to use multiple pageout daemons as discussed in Section 5.1. The resulting daemons must cope with system-wide (logical) page shortages as well as page shortages in the individual clusters.
- There must be a physical page available for every free logical page. Therefore use of a physical page for replication may require stealing a logical page from the resident page subsystem to maintain this invariant. Freeing of such a replicate should cause the stolen logical page to be returned to the resident page subsystem's free list.

Neither interaction poses great difficulties for an implementation.

The major interaction involves replication and copy-on-write. If the system has replicated a shared page that must be copied if written, then the replicates can be used to satisfy write faults on the page; this avoids the costs of creating an extra copy, but imposes extra costs if the replicate was used by more than one address space and has to be recreated as a result. The easy case is if there is a replicate that is only being used by the address space that caused the write fault; this replicate can always be used to satisfy the fault. For multiple address spaces, we would propose always using the replicate unless one of the other spaces has indicated that the replicate is needed (cf. the **always replicate** operation in section 10). An additional primitive must be added to Mach's machine-dependent interface to implement this functionality; the fault handler must be able to find out if the NUMA layer has a replicate that can be used to satisfy a write fault.

6 Hardware Support

Existing multiprocessor hardware will not allow a sufficiently accurate implementation of the NUMA memory management schemes discussed in this paper. Software systems that impose a level of indirection on all accesses to memory or shared memory can not hope to recover from this performance penalty. Thus we propose an architecture with hardware support for our algorithms. For each page, a set of two

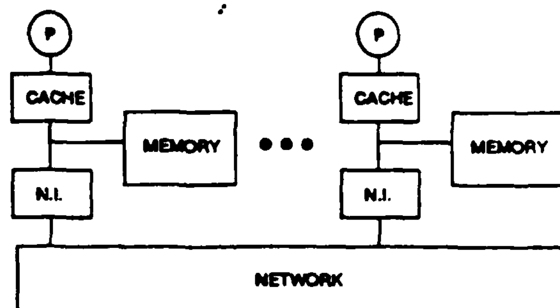


Figure 1: Architectural model

reference counters is required per processor in the system. Together with increment/decrement logic these maintain the counts required by algorithms R and M. An exception is caused when the built-in threshold for migration or replication is reached. The operating system then deals with the copying and remapping operations required.

The counters are kept with their associated memory page. For a 64-processor system and 16-bit counters, we thus require 256 bytes of memory per page. This translates to 50% overhead for 512-byte pages, 25% overhead for 1K pages, 6.25% for 4K pages and 3.125% for 8K pages. The overhead seems quite acceptable for pages in the 4K - 8K range.

For replication, we only need to increment a single counter. Migration, on the other hand, requires updating two counters, one of which must be chosen from the non-zero counters for that page (local references only update the latter counter). Since the updating of the counters must take place transparently and at the same speed of a memory reference, a sequential search for non-zero counters is not acceptable. An alternative is to pick a counter and decrement it if it is non-zero. This is much simpler to implement and our simulations indicate that its performance is similar to the original migration algorithm.

Copy on reference is the major alternative to our replication approach. It should be used where enough locality is known (e.g. from previous experimentation) or expected (e.g. code) to exist to cause replication by algorithm R; if replication is going to occur, it is always more efficient to do it in response to the first reference. The proposed algorithm R, however, has an advantage in cases where read-only data may not be accessed enough to cause replication; systems that manage large amounts of data for which locality cannot be assumed are an example. The choice of approach should depend on the situation being faced; copy on reference is probably more applicable to the most common situations than our delayed replication approach.

7 Performance Analysis

7.1 Architectural Model and Assumptions

The architectural model shown in Fig 1 was used for the analysis of the algorithms presented in this paper. It consists of several nodes linked by an interconnection network. Each node has a network interface(N.I.), its share of the global

memory, a processor and a cache. In the case of the NUMA architecture, the cache is write-through and is only used to cache memory locations in the local portion of the global memory. Global cache consistency is thus assured. The following costs were used for the various operations in our simulations:

Operation	Time
local reference	0.1 μs
remote reference	4.0 μs
replication of a page	1200 μs
migration of a page	2100 μs

The access costs are based on those found in the Butterfly; replication and migration costs were estimated by examining page fault overheads in Mach (e.g. replication is very similar to a copy on write fault). These times include the overhead of updating page tables; this results in larger migration costs because more page tables must be changed by a migration than by a replication.

We also evaluate a system that allows the caches to cache *all* memory locations and uses a directory-based scheme to keep the caches coherent [1]. The hardware requirements of this scheme are greater, both in terms of memory requirements and in terms of complexity of the directory controller. We assume the cache scheme has the following costs for comparison with the NUMA scheme:[2]

Operation	Time
cache hit	0.1 μs
cache miss	4.0 μs
invalidation	4.0 μs

The algorithms were evaluated using multiprocessor traces of three parallel applications: LocusRoute [13], MP3D [11] and P-THOR [17]. LocusRoute is a standard cell global router, which exploits parallelism at a fairly coarse grain. MP3D is a 3-dimensional particle simulator. It uses distributed loops and is a typical example of parallel scientific code. P-THOR is a parallel logic simulator.

The traces were gathered on a VAX 8350, using a combined hardware/software scheme [7]. All traces were 8-processor runs and contain about half a million references per processor (4 million references total).

A simulator was used to keep track of the location of every memory page and the values of the various counters. The initial placement of each page was random. Code pages were allowed to replicate while data pages could only migrate.

7.2 Results

Figure 2 shows the performance increases gained by applying replication and migration. We are plotting the overall runtime for four schemes. "Neither" designates a random placement of memory pages in the nodes with neither replication nor migration allowed. The other points show the effect of allowing only migration, only replication and then both. Each curve shows the results for one of the three applications. When both replication and migration are allowed, the overall runtime decreases by a factor of 5 to 10.

We also explored variations of three parameters: page size, replication threshold and migration threshold. The results from varying the page size are shown in Figure 3. We plot

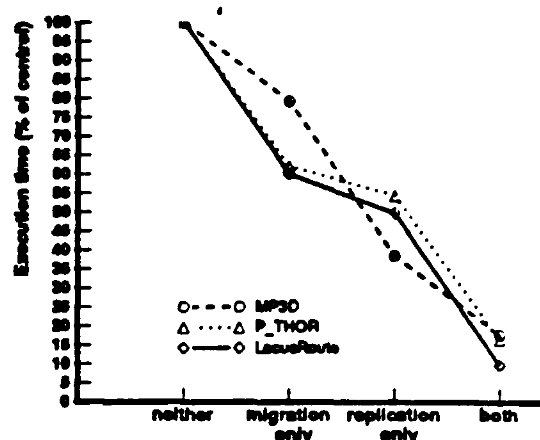


Figure 2: Performance Improvements

the effect of varying page size against the simulated run time of the trace. This time is shown as a percentage of the time required to execute the trace with replication and migration turned off (i.e. "neither" in Fig. 2).

Two effects are important when deciding the most efficient page size. Smaller pages are basically smaller units of replication/migration and would be expected to efficiently track the sharing needs of a program. At the same time, however, the fixed portion of remapping overhead makes larger pages more efficient. These two effects result in a U-shaped curve as seen in Figure 3. Although the position of the curves for the different applications varies vertically, their shape is basically identical. In every case the best page size was 512 bytes, but the effect of using larger pages was not significant.

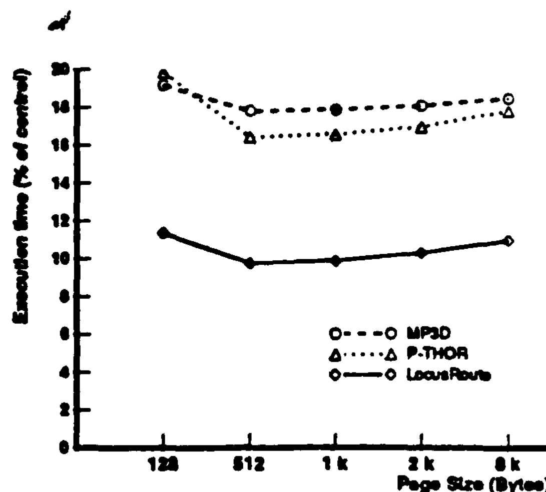


Figure 3: Effect of page size

Tuning the thresholds in these algorithms to match expected access patterns may improve average case performance without sacrificing constant factor bounds on the worst case performance. Tuning increases the constant factors in the bounds (i.e. the resulting algorithms are no longer competitive), but the increases may be offset by the improved average case behavior. For example changing the replication threshold in algorithm R from r to $0.5r$ or $2r$ increases the constant factor in the performance bound from 2 to 3. Our results show that threshold tuning has very little effect on overall

[2] Note that the cost given for invalidation is a *per remote invalidation* cost. Thus if a write reference results in invalidations in three remote caches, the total cost is assumed to be 12 μs.

performance. In each case lowering the threshold increases performance by a very small amount. Most of the pages are replicated or migrated just once, so the sooner the movement takes place, the lower the overall cost.

In the results presented above, each page was allowed to migrate any number of times. We also explored a variation where only a single migration per page was allowed – this basically allowed the program to achieve a good initial page assignment. The performance of this variation was just as good as when multiple migrations were allowed, indicating that a good initial assignment is the most critical factor. This may be due in part to the length of the traces. Longer traces may show a larger benefit for dynamic migration, as the program moves from one "working set" to another.

Tables 1 and 2 compare the performance of the NUMA memory management scheme to that of a directory-based cache scheme. Due to limitations of space, only results for LocusRoute are shown, but the relative performance was similar for the other two applications. The data shows that the cache scheme does about twice as well as the NUMA scheme. While cost for local references are comparable, the extra cost of remote references in the NUMA scheme is not offset by the extra cost of misses and invalidations in the cache scheme.

Table 1: NUMA scheme performance

NUMA		
Count	Operation	Cost (μs)
36	replication	43,200
86	migration	180,600
227,304	remote ref	909,216
4,114,180	local ref	411,418
	Total	1,544,434

Table 2: Cache scheme performance

CACHE		
Count	Operation	Cost (μs)
31,435	read miss	125,740
8,547	write miss	34,188
5,192	invalidation	20,768
4,301,493	hit	430,149
	Total	610,845

8 Extensions to Other Architectures

Competitive replication and migration algorithms have been found for certain extensions to our basic architectural model. A companion paper [2] presents competitive algorithms for replication and migration in arbitrary trees and architectures based on trees including hypercubes and meshes. The related topologies of rings and torii handle replication easily, but pose problems for migration.

Migration on rings and torii (products of rings) is problematic. Bidirectional rings exhibit the phenomenon of *pinning* [15] in which accesses in both directions from the far side of the ring can pin a page in place and prevent it from migrating closer to the accesses. Unidirectional rings or unidirectional routing structures imposed on bidirectional rings avoid this problem, but instead exhibit the phenomenon of *cycling* in which a static access pattern distributed over the ring can cause a page to cycle around the ring interminably (using up ring bandwidth) when it should stay put. It is possible that more sophisticated algorithms that keep additional information about the pattern and history of accesses can avoid these problems, but this extra state and the cost of updating it may affect the overall utility of such algorithms.

9 Replication of Writable Pages

So far we have not allowed the replication of writable pages. For portions of shared memory that are rarely written (called *mostly-read objects* in [20]), the amortized costs of the atomic updates required by the writes may not be prohibitive. Such a scheme can be implemented by using hardware mechanisms to cause a trap if a write occurs to any of the replicates. The handler for this trap can then perform the atomic update by disabling all access to all copies until the write has been propagated to all of them. Relaxed consistency constraints are preferable if the data has to be updated frequently. On the other hand, if the memory is never written after some point, then replication is a very good idea. Researchers working on the ACE project at IBM Hawthorne have found this to be the case for a parallel shortest path program: the data structures describing the graph to be searched are never written after the initialization phase, but are read heavily during the search. Replicating these structures into local memories on their machine produced major improvements in the run time of the application [3].

Algorithm R may not be appropriate for managing replicated writable shared memory because it ignores the costs of updating other replicates in response to a write. The *General-Snoopy-Caching* algorithm in [8] is a better choice if these costs are important because it takes them into account: this algorithm is competitive with a competitive factor of 3. If update costs depend on the number of replicates (e.g. if individual messages are required to update each replicate), then the algorithm must be modified accordingly in order to remain competitive.

10 Input and Feedback

If additional information is available about the access patterns for a page, the algorithms M and R can be further improved upon. We propose four primitives to help specify this additional memory usage information. The actual information may be provided by the user directly or it may come as feedback from a profiler. The primitives are:

Never replicate: On average, this page is used so infrequently in this cluster that it should never be replicated, even if it accumulates r accesses.

Always replicate: On average, this page will be used enough in this cluster to justify replication as early as possible. Alternatively, this page is read-only due to the use of copy-on-write techniques and is going to be written (which will require a copy to be made).

Never migrate: On average, this page is used so infrequently that it should not be migrated to this cluster even if it accumulates enough accesses to justify migration.

Anchor: This page will be so heavily used in this cluster that it should be anchored here and not allowed to migrate

until further notice. An option to reverse this effect is also needed.

Lazy evaluation can be used to delay the effects of **always replicate** until the memory in question is actually accessed. This is done by unmapping the page in hardware and performing the operation in response to the page fault generated by the first access. This permits greater flexibility in the use of this primitive, as no additional cost is imposed for pages that are not used; similar functionality is provided by copy on reference.

These primitives can also be used to provide feedback from the management algorithms and other instrumentation over multiple runs of an application to improve its performance by adapting its memory usage to the memory structure of the machine. This feedback may reduce the effort required to restructure data to take advantage of non-uniform memory architectures.

11 Related Work

Competitive management of distributed shared memory is a topic at the juncture of several active areas of research. Li [9], Cheriton [4], and others have implemented distributed shared memory using messages on a network. The hardware for these implementations does not support remote accesses or access forwarding; this removes the choice of the amount of data to send in response to a request that is critical to our work. Most research projects in the area of NUMA architectures have implemented a shared memory programming model; the best known is BBN's Uniform System [19], and it typifies them in that it directly exports the non-uniform memory structure to users. Our work supports automatic management mechanisms that free users from some of the details involved in managing non-uniform memory, and should make these machines easier to program. Scheurich and Dubois [15] have independently discovered an extension of our migration algorithm to mesh-connected machines and hypercubes, but not its competitive properties. They also note the pinning problem for bidirectional rings, but not the cycling problem for unidirectional rings. Rudolph and Segall [14] are investigating a bus-based hardware consistency mechanism for pages. Their work differs from ours in that it depends on a hardware consistency mechanism to permit replication of writable pages without weakening consistency. Finally our work makes contributions to the area of competitive algorithms; the migration algorithms are competitive solutions to several cases of the 'one server with excursions' problem [10]. While we would like to solve this problem in full generality (i.e. for any topology), we are of the opinion that any such solution must maintain too much state to be applicable to real systems. Finally the techniques of competitive algorithm analysis may be applicable to other resource management problems that occur in distributed systems and multiprocessors, such as load balancing [6].

12 Conclusion

This paper has presented and analyzed algorithms for managing memory in NUMA multiprocessors and related systems. Competitive algorithm analysis guarantees small constant factor bounds on performance with respect to optimal algorithms that require information on future memory reference behavior. A case study of the Mach VM system indicates that incorporation of these algorithms into an operating system kernel should not pose any great difficulties. In contrast, hardware support is required to obtain the full functionality of our approach on most multiprocessors. We have also sketched extensions of our approach to additional hardware architectures (e.g. hypercubes) and software programming models (e.g. weak consistency).

We used trace driven simulations to evaluate our approach and compare it to other alternatives. Speedups of 5 to 10 over random assignment of pages are achieved on production applications without modifying the applications for NUMA architectures. These results indicate that significant instruction and data locality may be present in many shared memory multiprocessor applications, and that this locality can be exploited automatically. We also compare our proposed hardware support with the more aggressive approach of fully-consistent caches. An additional factor of 2 in performance can be obtained from the cache approach, but at the cost of much more hardware.

Acknowledgements

Most of the theoretical results in this paper represent joint work with Daniel Sleator; complete proofs and details can be found in [2]. We would also like to thank Richard Rashid and Roberto Bisiani for encouragement and support. Anoop Gupta and Wolf-Dietrich Weber are supported by DARPA contract N00014-87-K-0828. Anoop Gupta is also supported by a faculty award from Digital Equipment Corporation.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *15th International Symposium on Computer Architecture*, 1988.

[2] D. Black and D. Sleator. Algorithms for the 1-Server problem with Excursions. Technical report, Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA, 1988. to appear.

[3] W. Bolosky. Personal Communication, September 1988.

[4] D. Cheriton. Unified Management of Memory and File Caching Using the V Virtual Memory System. Technical Report STAN-CS-88-1192, Computer Science Dept., Stanford University, Stanford, CA, 1988.

[5] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance Mmeasurements on a 128-node Butterfly Parallel Processor. In *Intl. Conf. on Parallel Processing*, pages 531-540, 1985.

[6] A. Ezzat. Load Balancing in NEST: a Network of Workstations. In *Fall Joint Computer Conference (FJCC)*, November 1986.

[7] S. Goldschmidt. Simulating Multiprocessor Memory Traces. EE390 Report, Stanford University, Dec. 1987.

[8] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive Snoopy Caching. Technical Report CMU-CS-86-164, Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA, 1986. Preliminary version appeared in 27th FOCS, 1986.

[9] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *5th Symp. on Principles of Distributed Computing*, pages 229–239, 1986.

[10] M. Manasse, L. McGeoch, and D. Sleator. Competitive Algorithms for Server Problems. In *20th Symp. on Theory of Computing*, pages 322–333, 1988.

[11] J. McDonald. A Direct Particle Simulation Method for Hypersonic Rarified Flow on a Shared Memory Multiprocessor. CS411 - Final Project Report, Stanford University, Mar. 1988.

[12] R. Rashid, A. Tevanian Jr., M. Young, D. Golub, R. Baron, D. Black, J. Chew, and W. Bolosky. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Archtectures. *IEEE Trans. Comput.*, 37(8):896–908, August 1988.

[13] J. Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Design Automation Conference*, pages 189–195, June 1988.

[14] L. Rudolph and Z. Segall. Dynamic Paging Schemes for MIMD Parallel Processors. Research notes on work in progress.

[15] C. Scheurich and M. Dubois. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. In *Int. Conf. on Distributed Computer Systems*, pages 162–169, 1988.

[16] D. Sleator and R. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM*, 28(2):202–208, February 1985.

[17] L. Soule and T. Blank. Parallel Logic Simulation on General Purpose Machines. In *Design Automation Conference*, pages 166–171, June 1988.

[18] A. Tevanian Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1987.

[19] R. Thomas and W. Crowther. The Uniform System: An approach to runtime support for large scale shared memory multiprocessors. In *Proc. of 1988 Int. Conf. on Parallel Processing, Vol II*, pages 245–254, 1988.

[20] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, Apr. 1989.

[21] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *11th Symp. on Operating Systems Principles*, pages 63–76, 1987.

Analysis of Cache Invalidation Patterns in Multiprocessors

Wolf-Dietrich Weber and Anoop Gupta
(Draft: Sep 20. 1988)
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

To make shared-memory multiprocessors scalable. researchers are now exploring cache coherence protocols that do not rely on broadcast, but instead send invalidation messages to individual caches that contain stale data. The feasibility of such directory-based protocols is highly sensitive to the cache invalidation patterns that parallel programs exhibit. In this paper. we analyze the cache invalidation patterns caused by several parallel applications and investigate the effect of these patterns on a directory-based protocol. Our results are based on multiprocessor traces with 4, 8 and 16 processors. To get insight into what the invalidation patterns would look like beyond 16 processors. we propose a classification scheme for data objects found in parallel applications and link the invalidation traffic patterns observed in the traces back to these high-level objects. Our results show that synchronization objects have very different invalidation patterns from those of other data objects. A write reference to a synchronization object usually causes invalidations in many more caches. We point out situations where restructuring the application seems appropriate to reduce the invalidation traffic. and others where hardware support is more appropriate. Our results also show that it should be possible to scale "well-written" parallel programs to a large number of processors without an explosion in invalidation traffic.

1 Introduction

One of the most critical issues in the design of shared-memory multiprocessors is the cache coherence strategy. Most multiprocessors rely on a shared bus and use a broadcast-based protocol to keep the caches coherent [8,16,18,15,23]. However, such multiprocessors are not very scalable, as the shared-bus soon becomes a bottleneck. As an alternative. researchers have started exploring cache coherence protocols that do not rely on broadcast, the most common example being directory-based protocols [2,4]. These protocols rely on the system having knowledge about which caches contain a particular piece of data. On a write. invalidation messages are sent only to these specific caches. The number of pointers in each directory entry determines how many other caches can be kept track of. Determining the performance of directory-based protocols requires the answer to several questions. We would like to know the distribution of the number of remote caches that need to be invalidated on shared writes. We would like to know how these distributions scale as the number of processors is increased. We are interested in knowing what types of data objects in the applications result in what kind of invalidation patterns. This paper attempts to answer some of these questions for directory-based protocols.

We analyze the patterns of invalidation traffic produced by a set of five application programs. Three of the five applications selected are "real" parallel programs. in the sense that they solve real-world problems and that a lot of effort has gone into obtaining good processor efficiency with them. The remaining two applications are smaller. but they are still interesting in that they could form the kernels of larger applications. Our study is based on memory reference traces obtained for the applications when simulating 4. 8. and 16 processors.[1] The traces were

[1] Previous studies [1,2] presented results using traces with only 4 processors. This study uses a more extensive set of applications. a larger number of processors. and goes more deeply into the causes of invalidations patterns.

To appear in Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, April 1989.

generated using software-traps on a 4-processor VAX-8350 and a VAX-3200 running MACH. In addition to presenting the invalidation patterns as observed directly from the traces, the paper links the invalidation patterns to the high-level program data structures (objects) that cause them. A classification of such shared objects on the basis of their expected invalidation behavior is given. Linking the invalidation patterns to the high-level objects helps us predict how the invalidation traffic would change as the number of processors is increased. It is far more accurate to extrapolate the behavior of each class of data object than to simply extrapolate the composite behavior. For the application types we have considered, our results indicate that it is quite possible to write parallel programs that do not create an enormous amount of invalidation traffic. Thus directory-based schemes with just a few pointers per entry could efficiently execute well-designed parallel programs.

The next section explains the methodology used in generating the traces and explains how the traces were analyzed. Section 3 introduces the five applications used in this study and gives a brief overview of their computational behavior. In Section 4 we present some basic trace characteristics. In the next section we present the proposed classification of shared data objects in parallel programs. Section 6 goes into a detailed analysis of the invalidation behavior of each application and relates these patterns to specific data objects in the applications. Section 7 assembles the results from the various applications and presents conclusions.

2 Methodology and Assumptions

The traces were collected using a combined hardware/software method [7]. The process creation is modified to have one master process, which controls the actual tracing, and a number of slave processes, one for each "virtual processor". Once the desired start position for tracing is reached, each of the slaves stops itself and is then single-stepped by the master. The stepping takes place in a round-robin fashion. The stepping employs the UNIX *ptrace* system call which uses the T-bit on the VAX. While stepping, the master process records data in the trace file. For each reference, the type (I-fetch, read, or write), the address, and the CPU number are recorded. Trace lengths used were 20Mbytes for 4-processor traces, 30Mbytes for 8-processor traces, and 50Mbytes for 16-processor traces. This corresponds to about 2.5, 4 and 7 million references respectively, or around 0.5 million references per processor.

The traces were gathered on a VAX-8350 with 4 processors and a VAX-3200 workstation, both running the MACH operating system. MACH allows allocation of shared memory for the processors. On the 8350 it takes about 24 hours to obtain 20Mbytes of trace, while the VAX-3200 can gather about 50Mbytes in the same time.

Once the traces were gathered, they were used as input to a program that simulates multiprocessor cache behavior and gathers statistics. Infinite caches were used for simplicity of the cache simulator. The cache coherence protocol used was an invalidation scheme similar to the Berkeley Ownership scheme [16]. For each potential invalidation, a record was written containing the CPU number, the data address, the most recent instruction address and the number of other caches actually invalidated. The data and instruction addresses were later used to associate the invalidation with the high-level language construct that caused it. Several post-processing programs were used to gather statistics from the invalidation traces.

The main advantage of the software scheme of gathering traces is that we can get traces for an arbitrary number of processors, which is not possible with hardware schemes like ATUM [20]. However, there are some disadvantages too. For example, the *ptrace* call does not trace operating system calls, but rather treats them as a single reference. This is not a major problem

in this study, since there are not many operating system calls in the sections traced. Also. each instruction takes one time unit to complete. regardless of the complexity of the instruction. This is clearly an oversimplification. but there is no reason to believe that it significantly distorts results.

3 Application Programs

In this section we describe the data structures and computational behavior of the applications. This is important background for Section 6, where we relate invalidation traffic to high-level objects. The applications used for tracing were selected to represent a variety of algorithms used in an engineering computing environment. All of the applications were written in C. The Argonne National Laboratory macro package [11.12] was used to provide synchronization and sharing primitives. The synchronization primitives used include spin locks, as well as barriers and distributed loops.

3.1 Maxflow

Maxflow [3] finds the maximum flow in a directed graph. This is a common problem in operations research and many other fields. The program is a parallel implementation of an algorithm proposed by Goldberg and Tarjan. The bulk of execution time is spent picking off nodes from a task queue, adjusting the flow along its incoming and outgoing edges, and then placing its successor nodes onto a task queue. Maxflow exploits parallelism at a fine grain.

Maxflow does not assign the nodes of the graph to processors statically. Instead, task queues are used to distribute the load. Each processor has its own local task queue and need only go to the single global task queue when the local queue is empty. Tasks are put onto the global queue only when processes are waiting there. and onto the local queue otherwise. Note that the task queues are made up of the nodes themselves, linked together with appropriate pointers. Locks are used to serialize access to each node element, but contention for these is fairly low. as there are many more nodes than processors. In Section 6 we will see that most invalidations are related to the global task queue and the migration of node data from one processor to another.

The traces were collected while solving Maxflow for a set of nodes arranged as a 10-ary 2-cube. Tracing was started as the program entered the main loop after completing the initial distance labeling. The implementation provides speedups of about 8 with 12 processors.

3.2 SA-TSP

SA-TSP [21] solves the traveling salesperson problem using simulated annealing [10]. A linear array contains the cities in tour order. At each step. a processor selects a pair of cities to swap. The swap is performed if it results in a shorter tour or if the increase in tour distance is within the margin prescribed by the cooling function. The tour is locked *only* during the actual swap. which means that errors occur when the tour has been modified between making the decision and actually performing the swap. This trades off quality of solution for greater speedup. Note that there is only one global lock for all the tour data. This becomes a major bottleneck as the number of processors increases. In the initial annealing phase - which is the section we traced - most moves are accepted and contention for the lock is especially large. While the program achieves an overall speedup of 7 with 8 processors. no more than 4 processors can be kept busy during this initial portion.

3.3 MP3D

MP3D [13,14] is a 3-dimensional particle simulator for rarified flow. It is used to study the shock waves created as an object flies at high speed through the upper atmosphere. MP3D is a good example of scientific code that is vectorizable and can be parallelized using distributed loops. A version of MP3D that runs on the Cray-2 is being used extensively at NASA for research.

The overall computation of MP3D consists of evaluating the positions and velocities of molecules over a sequence of time steps. During each time step, the molecules are picked up one at a time and moved as governed by their velocity vectors. Collisions with the boundaries and with each other are resolved. The simulator is well suited to parallelization because each molecule can be treated independently at each time step. The work is spread over the processors with the help of a distributed loop, consisting of a lock and a global index variable. Each processor obtains the lock, reads the index, increments it, and releases the lock. In this manner the processes pick up the index of the next particle to be moved. The traces cover most of one time step, i.e. each particle is moved once. No locking is employed in the various arrays that keep track of the particles and space, because collisions are impossible in the particle arrays and very rare in the space arrays. Thus, the distributed loop is the only synchronization seen in this trace.

3.4 Distributed CSIM

Distributed CSIM [22] is a parallel logic simulator developed at Stanford University. It is an interesting application based on the Chandy-Misra simulation algorithm [5], which is specially designed for highly parallel machines — unlike event-based algorithms, this algorithm does not rely on a single global time during simulation.

The primary data structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (the wires linking the elements) and the task queues which contain activated elements. Each processor has as many task queues as there are other processors. This ensures that there is no contention when adding elements to some other processor's queue. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on the element's outputs. It then looks up the net data structure to determine which elements are affected by the output change and potentially schedules those activated elements onto other processors' task queues. Newly activated elements are assigned to other processors in a round-robin fashion.

3.5 LocusRoute

LocusRoute [17] is a global router for VLSI standard cells. It is a real application in that it is a part of a system that has been used to design real integrated circuits, and it has been highly tuned to run well on a shared-memory multiprocessor. LocusRoute represents the class of parallel programs that exploit fairly coarse grain parallelism.

The LocusRoute program exploits parallelism by routing multiple wires in a circuit concurrently. Each processor executes the following loop: (i) remove a wire to route from the task queue; (ii) explore alternative routes; and (iii) pick the best route for the wire and place it there. The central data structure used in LocusRoute is a grid of cells called the *cost array*. Each row of the cost array corresponds to a routing channel for standard cells. LocusRoute uses the cost array to record the presence of a wire at each point, and the congestion of a route is used

as a cost function for guiding the placement of new wires. No locking is needed in the cost array, which is accessed and updated simultaneously by several processors, because the effect of occasional collisions is tolerable. Each routing task is fairly large grain, which prevents the task queue from becoming a bottleneck.

4 Trace Characteristics

Table 1 gives an overview of the traces of the five applications. For each application, we give the trace length in number of references and the breakdown in terms of I-fetches, reads and writes. We also show the proportion of shared writes, and the average number of invalidations caused by each shared write. In addition to absolute numbers, the columns also list the number of references in each category as a fraction of all references in the trace.

Table 1: General Trace Characteristics.

Application	num of CPUs	refs mill	I-fetches mill	I-fetches %	reads mill	reads %	writes mill	writes %	shared writes thous	shared writes %	avg. invals per sh-wrt
Maxflow	4	2.62	1.21	46	1.06	40	0.35	13	73.6	2.81	0.30
	8	4.15	1.91	46	1.69	41	0.55	13	121.6	2.93	0.49
	16	8.36	3.86	46	3.46	41	1.04	12	274.8	3.29	1.07
SA-TSP	4	2.65	1.10	42	1.12	42	0.43	16	19.5	0.74	1.27
	8	4.16	1.84	44	1.88	45	0.44	11	37.3	0.90	2.29
	16	7.11	3.30	46	3.37	47	0.43	6	77.0	1.08	2.93
MP3D	4	2.53	1.57	62	0.80	32	0.17	7	83.7	3.31	0.68
	8	3.59	2.22	62	1.13	31	0.23	6	119.9	3.34	0.80
	16	7.05	4.28	61	2.33	33	0.43	6	230.3	3.27	1.03
Dist CSIM	4	2.61	1.28	49	1.01	39	0.32	12	8.5	0.33	0.44
	8	4.13	2.04	49	1.61	39	0.48	12	19.9	0.48	0.46
	16	7.09	3.52	50	2.80	39	0.77	11	42.5	0.60	0.51
LocusRoute	4	2.60	1.31	50	0.95	37	0.33	13	5.6	0.22	0.56
	8	4.34	2.26	52	1.59	37	0.49	11	4.8	0.11	1.07
	16	7.70	3.95	51	2.83	37	0.92	12	9.2	0.12	1.28

In all of the programs, with the exception of MP3D, about 45-50% of the references are I-fetches. MP3D has a larger proportion of I-fetches because there are a lot of array references which require several instructions to compute the effective address of the reference.

The proportion of read references varies from about 30% in MP3D to over 45% in SA-TSP. In SA-TSP there are a lot of simple integer reads when determining the effect of a swap on tour distance. The read fraction is low in MP3D because of the larger proportion of I-fetches.

Writes hover around 10-15% of all references. MP3D again stands out with a very low write fraction, again due to frequent array references. The number of writes in SA-TSP stays virtually constant even though the number of references increases greatly as we move from 4 to 16 processors. This is explained by the fact that writes are only used when a swap is accepted. Contention for the lock in the portion of SA-TSP traced is so large that no more swaps are accepted in the 16-processor trace than in the 4-processor trace. This portion of SA-TSP was

chosen to demonstrate the effects that a poorly written program segment may have on directory-based coherence schemes. Details are presented in Section 6.2.

In our study. we define *shared locations* to be those that are referenced by more than one process in the trace. and we define *shared writes* to be write references to shared locations. Note that some locations that really are shared in the application are considered not-shared in our study. because within the limited length of the trace multiple processes do not reference those locations.

The second to last column in Table 1 presents the proportion of shared writes in the applications — it is important to study shared writes because they can cause invalidations in some or all of the caches. There is a general trend towards an increasing percentage of shared writes as the number of processors increases. One reason for this is larger contention over locks. The locks are implemented as test-test&set sequences and thus cause additional shared writes when several processors are contending for a lock that was just freed. Also. as more processors are added. the chances of a data item being accessed by more than one process increases.[2] resulting in a larger fraction of shared writes.

An important metric of invalidation traffic is the average number of invalidations per shared write. The values are shown in the last column of Table 1. This parameter is the largest for SA-TSP. mostly due to invalidation traffic caused by the single global spin-lock. In fact. the average number of invalidations increases steeply with more processors due to the increased contention for this global lock. The number of invalidations per shared write is the smallest for distributed CSIM. and hardly goes up as the number of processors is increased. This is mainly because there are no synchronization objects in the portion of distributed CSIM traced. Averages. however. do not carry all of the interesting information. Consequently. the detailed invalidation distributions and their analysis are presented in Section 6.

5 Classification of Data Objects

When trying to extrapolate invalidation behavior to a larger number of processors. it is important to explain the invalidation patterns in terms of the underlying high-level structures which cause the invalidations. We distinguish several types of shared objects on the basis of their significance in parallel programs and their expected invalidation behavior [1]:

1. Code and read-only data objects.

2. Migratory objects.

3. Synchronization objects.

4. Mostly-read objects.

5. Frequently read/written objects.

Code and read-only data objects obviously do not cause invalidations at all. and thus pose no problem to any coherence scheme. A fixed database such as the matrix that contains the distances between cities in SA-TSP is a good example of such read-only data.

[2]This is partly because we get a longer trace for a run with more processors. and partly because with a larger number of processors. there is a higher probability that subtasks sharing data get scheduled on different processors rather than on the same processor.

Migratory data objects are those that are manipulated by only a single processor at a time. Shared objects protected by locks often exhibit this property. While such an object is being manipulated by one processor, the object's data resides in the associated cache. When the object is later manipulated by some other processor, the cache entry of the previous processor needs to be invalidated.[3] Migratory data usually causes a high proportion of *single* invalidations. The nodes in Maxflow are a good example of migratory data. Each node is evaluated by several processors over the complete run, but there is only one processor manipulating each node at any one time.

Synchronization objects such as locks can cause a very large number of invalidations if used improperly. When locks are implemented as test-test&set, and there are processors waiting on a lock, invalidations are caused each time the lock changes hands. As a lock is freed, all waiting processors fall through the test part of the test-test&set. They then attempt the test&set, but only one of them succeeds, causing invalidations in all other waiting processors' caches. It is important to use locks in a manner that minimizes contention for them.

An example of mostly-read data is the cost-array of LocusRoute. Most of the time it is just read, but every now and then, when the best route for a wire is decided, the array is written. It is a candidate for large number of invalidations because many reads by different processors occur before each write. Thus the data is cached by many processors, and a write causes many invalidations. However, since only the writes cause invalidations and writes are infrequent, the overall number of invalidations will be quite small.

Finally, there is frequently read/written data.[4] An example is the variable that counts how many processors are waiting on the global task queue in Maxflow. Frequently read/written data has the worst invalidation behavior. Unlike mostly-read objects, this data is written quite frequently. Although each write may only cause 3 or 4 invalidations, this may exceed the number of pointers per entry in a directory scheme, thus causing frequent broadcasts. This type of data object should be avoided if at all possible.

6 Application Case Studies

In this section we present the results of the detailed analysis of the invalidation traces produced when running the cache simulator over the multiprocessor traces. For each application, we show the overall invalidation patterns, the high-level objects causing the invalidations, the expected broadcast behavior of directory-based cache coherency schemes [4,2], and the scalability of the application beyond 16 processors.

The overall invalidation behavior is presented in terms of an invalidation distribution graph as shown in Figure 1. The graph shows the fraction of shared writes that caused no invalidations, single invalidations and so on. Ideally these graphs will contain a large proportion of small invalidations, as these can be handled efficiently by directory-based cache schemes. By comparing the invalidation distributions for 4, 8 and 16 processor traces, we can begin to get a feeling for how the invalidations scale with a larger number of processors. We would prefer to see no change in the distribution as the number of processors is increased, but it is more likely that a shift towards both more and larger invalidations occurs.

For each application, we also present another kind of graph that shows the fraction of broadcasts required as a function of the number of pointers per entry in the directory (see Figure

[3] Cheriton discusses a programming model based on such objects, called *"workforms"* in [6].

[4] Frequently read/written should be interpreted as both frequently read *and* frequently written.

6). A directory-based scheme such as Dir_iB needs to use broadcast when a shared write is to a location that is contained in more caches than there are directory pointers for that entry. The data is plotted for directories with pointers varying from 1 to n, where n is the number of processors in the trace. We do not show directory schemes with 0 pointers as these require a broadcast for every shared write. Obviously, a directory with n pointers can keep track of all processors and broadcast is never required.

6.1 Maxflow

Figures 1, 2 and 3 show the invalidation distributions for Maxflow with 4, 8 and 16 processors respectively. Note that the distribution shifts to larger number of invalidations as the number of processors is increased. While at 4 processors only about 2% of shared writes cause more than one invalidation, this figure moves up to 18% with 16 processors. Analysis shows that the bulk of this increase is due to synchronization traffic involving the global task queue. Figures 4 and 5 show the invalidation distributions broken down by global queue traffic and all other invalidation traffic respectively. The global queue traffic includes all writes to the queue locks as well as the count of the number of processors blocked and the queue head pointer. It is clear that most of the spreading of the invalidation distribution is due to global-queue-related traffic.

A large fraction of the invalidations in Figures 1, 2 and 3 are single invalidations. They are caused by the manipulation of nodes and edges, which are good examples of migratory data objects. One processor picks up an active node and pushes flow through it. Later the node will get re-activated, and some other processor will pick it up and start processing it.

Some parameters of the nodes, such as its distance label, behave like mostly-read objects. Distance labels only get changed in the infrequent re-labeling steps. Between re-labeling, many processors may read a node's distance label causing re-labeling to generate a large number of invalidations. In the 16-processor trace, an average of 4.6 invalidations occur for each re-labeling write. Although 4.6 invalidations per shared write is large, the effect of these writes on the total number of invalidations is small since the writes are very infrequent.

The locks for the global task queue cause a large number of invalidations. Not only are they accessed and written frequently, but they also cause an average of about 2 invalidations per shared write in the 16-processor trace. The global queue is the major source of double or larger invalidations and should be a primary target for efforts aimed at improving the program.

The per-node locks, on the other hand, work well. They are an example of a synchronization object that causes few invalidations. There are so many more nodes than processors that contention is very limited.

The count of how many processors are waiting for the global task queue is checked frequently by all processors. It is also written frequently, namely whenever a process starts waiting on the global task queue. It is thus often read and written and causes many invalidations. It has an average of 2.8 invalidations per shared write and the highest number of shared writes to any single data object except for the global task queue locks.

A pattern of double invalidations found in Maxflow is very common when dealing with queues and is seen in several other applications. In Maxflow, one processor puts a node onto the global task queue, a second one picks it off, and a third one may later place the node on another queue. At first, the object is owned by one processor. When the node is picked up by the second processor, it becomes read-shared. Finally, the third processor writes the object, causing double invalidations in the link pointers. Many variations of this basic theme exist. Another example was found in POPS [9], a parallel rule-based expert system, where a single buffer is used for a

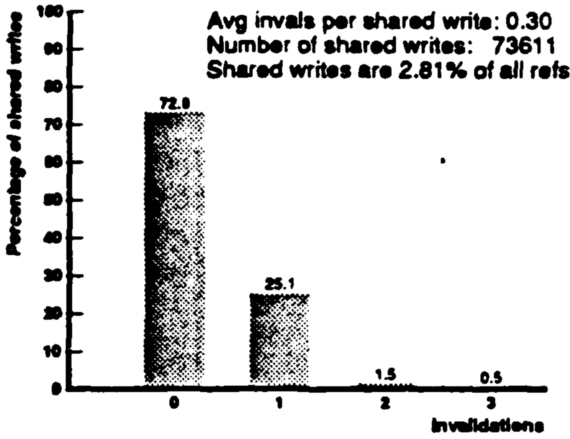


Figure 1: Maxflow 4

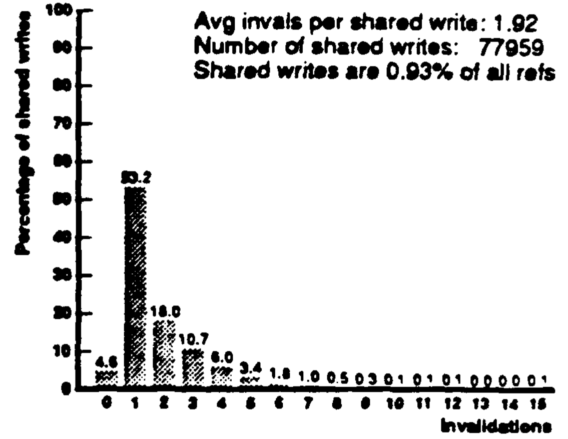


Figure 4: Maxflow 16 (Queue)

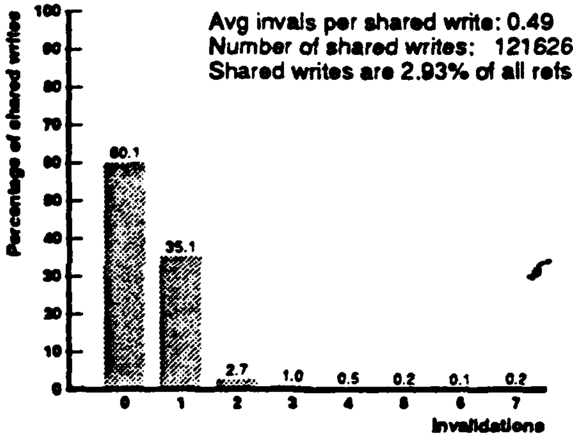


Figure 2: Maxflow 8

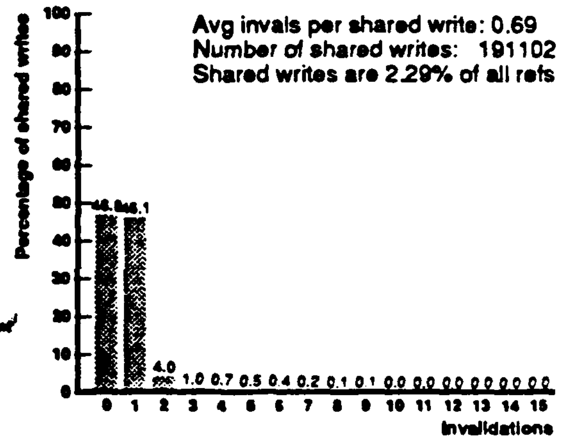


Figure 5: Maxflow 16 (Data)

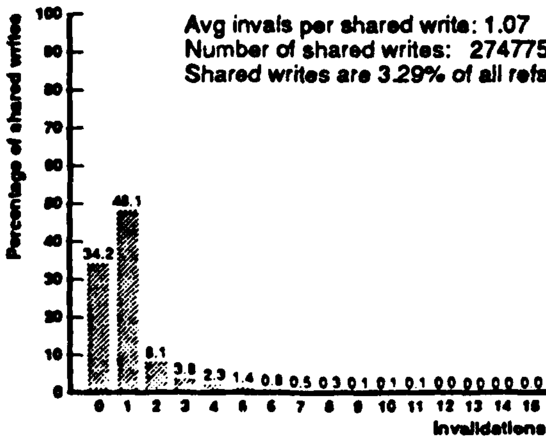


Figure 3: Maxflow 16

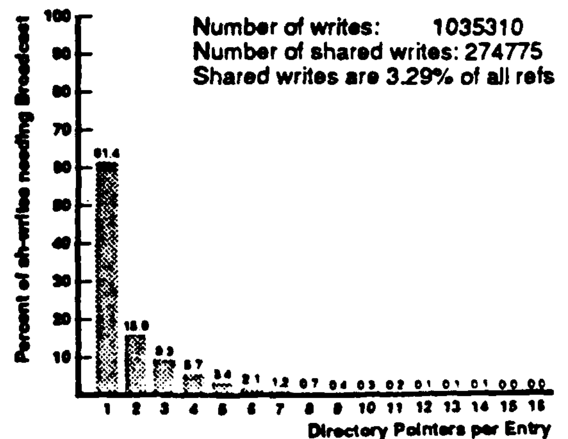


Figure 6: Maxflow 16 Directory Performance

task queue. An item is written into the buffer by one processor and read by another. Later. a third processor overwrites that item with some new data. thus invalidating the caches of both previous processors.

Figure 6 shows the proportion of shared writes that need to be broadcast for directory-based schemes with a varying number of pointers per entry. Although a scheme with two pointers per entry (Dir_2B in [2]) only needs to broadcast 1.8% of shared writes with 4 processors. this figure jumps up to 15.9% for 16 processors. The invalidation distribution keeps spreading out as the number of processors is increased. mostly due to the invalidations associated with the global queue.

Let us now use the object classification to see how the invalidation distributions will change as the number of processors is scaled. We expect little change in the invalidations produced by migratory objects which will continue to produce single invalidations. Mostly-read objects will have a slightly higher average number of invalidations per shared write because more processors are likely to have cached the data. Note though. that the average number of invalidations per write (4.6 for 16 processors) may already be beyond the number of pointers stored in the directory. so *no* additional broadcasts will result. Synchronization objects and frequently read/written objects. on the other hand. are expected to have a higher average number of invalidations per shared write. In addition. we expect to see *more* shared writes due to synchronization. Since both synchronization objects with high contention and frequently read/written objects exist in Maxflow. we will see a continued spread of the invalidation distribution towards larger invalidations per shared write. If the program is to be scaled successfully, we will have to reduce synchronization contention and eliminate frequently read/written objects.

6.2 SA-TSP

Figures 7. 8 and 9 show the invalidation distributions for SA-TSP with 4, 8 and 16 processors. Most noticeable is the hump in the invalidation distribution for 16 processors at around 12 to 13 invalidations. This hump is less obvious with 8 processors and does not appear with 4 processors. All of the invalidations that make up this hump in the 16-processor distribution are due to the single global lock. In fact as many as 94% of all invalidations are due to that lock.

Figures 10 and 11 show the invalidation distribution for the 16-processor trace. broken down into lock traffic and all other data traffic. These graphs show clearly that nearly all of the large invalidations are due to the single lock. This is a good example of how a poorly-used lock can flood a machine with invalidations. In the initial annealing phase (the portion that was traced). most moves get accepted. Thus all of the processors want to update the global tour. which requires the lock. This results in very high contention for the lock. We found that with 12 to 13 processors waiting for the lock to be released. this phase of the program could use no more than about 4 processors. As the cooling function progresses, fewer and fewer moves are accepted. contention for the lock subsides and the program achieves good speedup.

The invalidations due to the shared data range between 0 and about 8. All of these are from the array that holds the order of the cities in the tour. The large average of shared-write invalidations is due to the mostly-read nature of this data. A processor needs to look at two cities and their four neighbors to determine whether a swap is to occur. and only if the swap meets certain annealing criteria does it actually take place. This means that for each proposed swap. at least four cities are only read. not written. Each successful swap thus invalidates a large number of caches. The frequency of invalidations is due to the fact that there are relatively few data objects (36 in this case. as the program was solving a tour with 36 cities). especially when

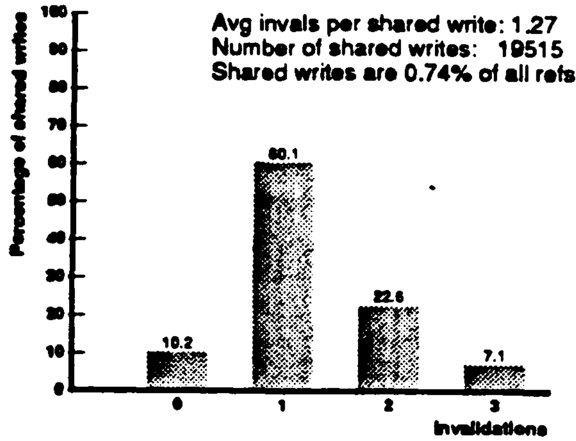


Figure 7: SA-TSP 4

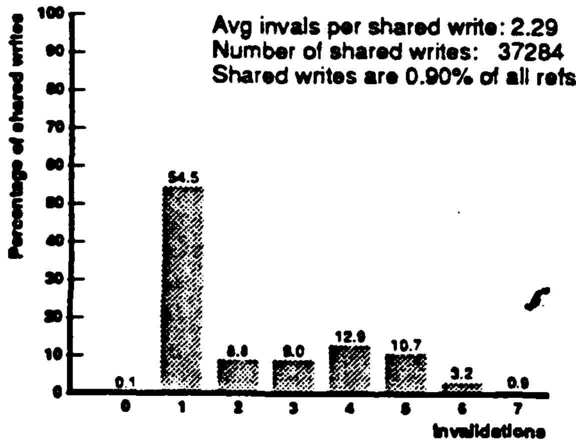


Figure 8: SA-TSP 8

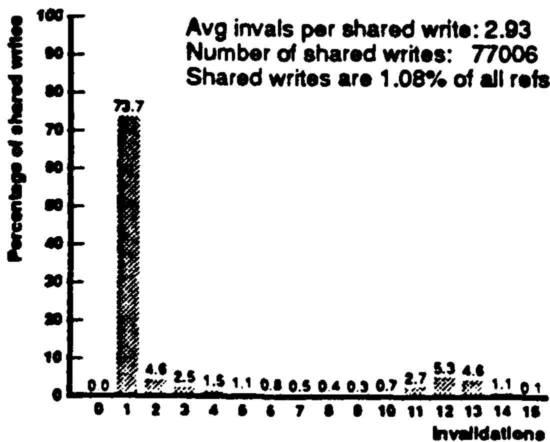


Figure 9: SA-TSP 16

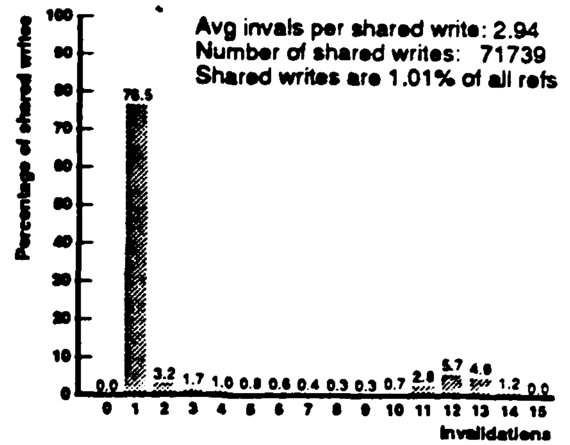


Figure 10: SA-TSP 16 (Lock)

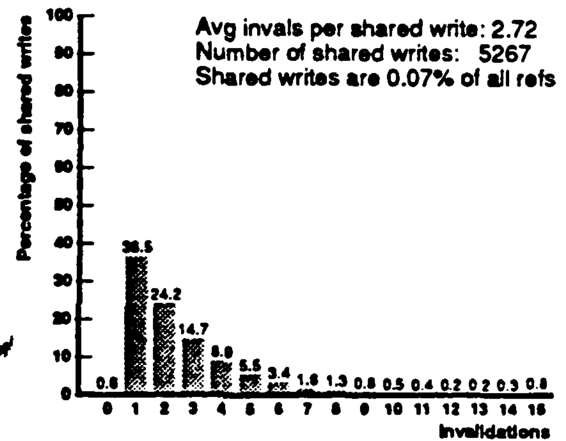


Figure 11: SA-TSP 16 (Data)

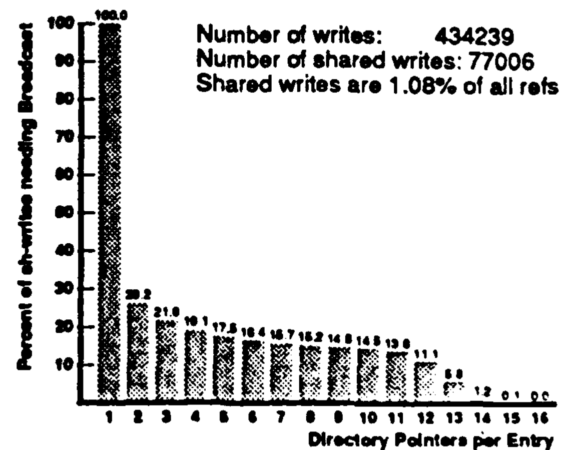


Figure 12: SA-TSP 16 Directory Performance

compared to LocusRoute or MP3D, where there are thousands of objects. Hence the chances of some other processor caching an object before it is written are much larger.

Figure 12 shows that even directory schemes with large number of pointers per entry perform poorly in the face of SA-TSP's invalidation traffic. After an initial lowering in the number of broadcasts with increasing number of directory pointers, the graph basically flattens out until we reach the hump. In the 16-processor case, a 10-pointer scheme would perform essentially as poorly as a 5-pointer scheme.

Further scaling of the number of processors would result in even larger contention for the global lock. This would move the invalidation hump to a larger number of invalidations per shared write. Essentially no additional useful work would be accomplished. A distributed locking scheme could reduce contention for the elements of the global tour. Even if the synchronization traffic is eliminated, however, we will still have a fair amount of shared data invalidation traffic. This is due to the fact that there are only a small number of data objects that are continuously read and written by several processors.

6.3 MP3D

Figures 13, 14 and 15 show the invalidation distributions for MP3D with 4, 8 and 16 processors respectively. The distributions are dominated by zero and single invalidations. As we increase the number of processors, some invalidations of 2 or more start to appear. This effect is most noticeable with 16 processors. Further analysis shows that the bulk of the double or larger invalidations are due to the monitor lock of the distributed loop. Figures 16 and 17 give the invalidation distribution for the 16-processor trace, broken down into monitor lock traffic and all other traffic. Here we note that shared data contributes very little to the invalidations of 2 or more. There are 0.02% that we do not see in the graph, and which are due to occasional collisions in the various data arrays. Unlike SA-TSP, where there are very few data elements, the number of data elements is very large in MP3D and so we do not see any large invalidations. The monitor lock traffic distribution, however, is seen to have significant portions beyond single invalidations. The ratio of time spent doing useful work to time spent in the monitor was found to have an average value of about 16. If there are fewer than about 16 processors, they manage to stagger themselves in the first round of contention. Contention in subsequent rounds is very limited because staggering has occurred. This means that with any more than about 16 processors, we will see a step-increase in invalidations for each processor added. In this manner, a well-behaved program can suddenly produce a very large number of invalidations as it is being scaled.

It is interesting to note that a much faster implementation of the distributed index is possible with some hardware support. This would shift the ratio of unlocked to locked time to a much higher value and would enable the program to be scaled beyond 16 processors. A similar result could be achieved by increasing the grain size — for example by letting each processor move 5 molecules instead of one at a time.

The monitor lock illustrates another phenomenon. When contention for a critical section is low, the lock references cause few invalidations. As more processors are added, the critical section becomes a bottleneck and contention for the lock increases. This in turn raises the number of invalidations caused by lock references. By fixing the program to remove the bottleneck we can also fix the problem of generating a large number of invalidations. In conclusion, synchronization objects themselves are not a problem unless contention for them is high. Since distributed loops and barriers are usually built out of spin locks, this conclusion applies to these synchronization

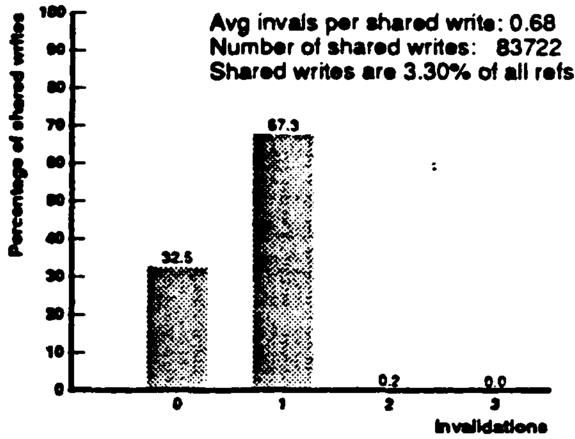


Figure 13: MP3D 4

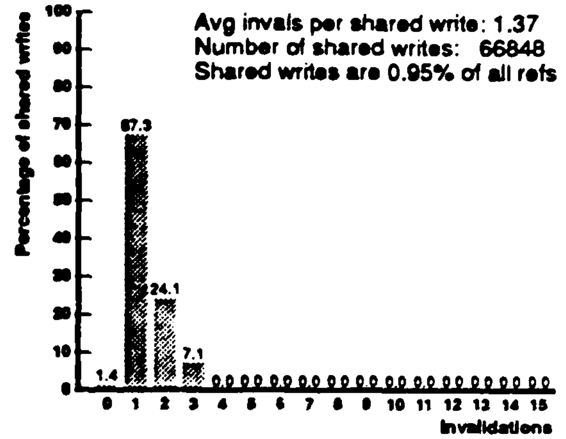


Figure 16: MP3D 16 (Synchronization)

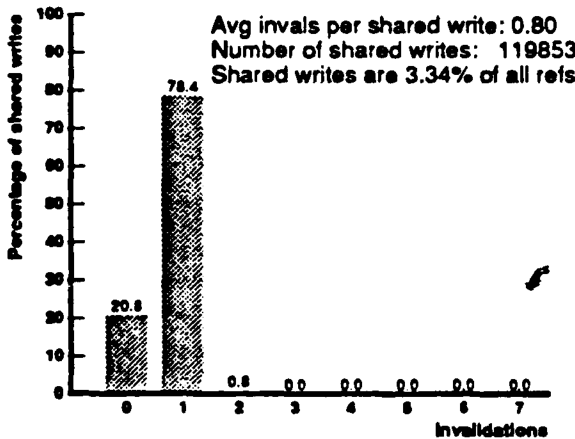


Figure 14: MP3D 8

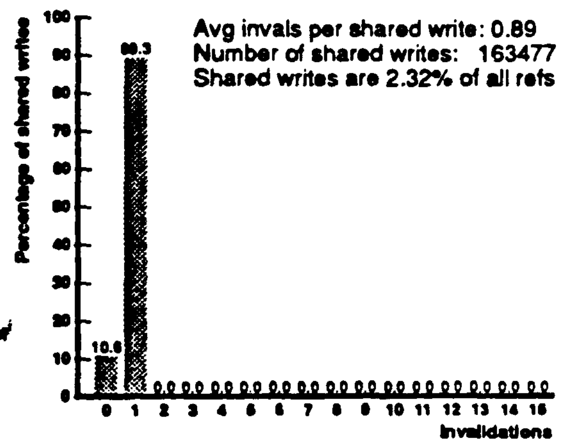


Figure 17: MP3D 16 (Data)

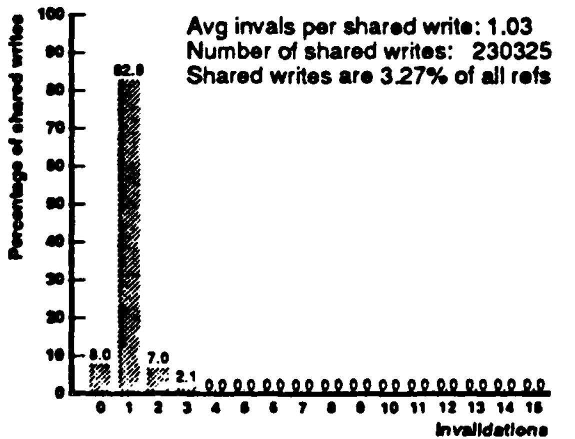


Figure 15: MP3D 16

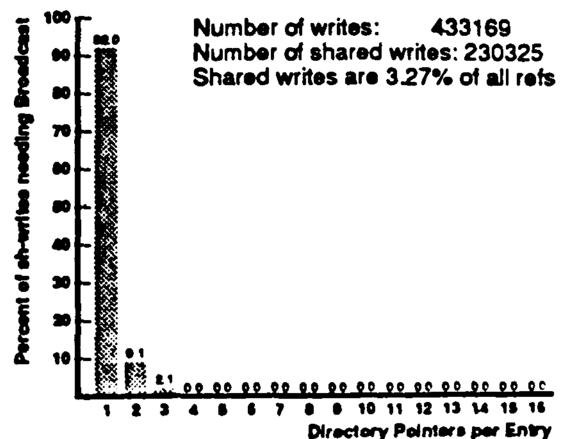


Figure 18: MP3D 16 Directory Performance

objects as well.

Most accesses to shared data by MP3D consist of a read followed immediately by a write. This will allow at most one other cache to be invalidated, unless two processors are accessing the exact same portion of data at the same time. Chances of such a collision are very low and their effect can be tolerated in MP3D, hence no locks are required for the shared data. Update-type data objects such as the shared data of MP3D, can be considered to be a special case of migratory objects, and their invalidation behavior is very similar. The only difference is that each data object is kept for only a short period of time before it moves on to the next processor.

As Figure 18 indicates, directories with just two or three pointers per entry would do extremely well with MP3D. For 3-pointer directory schemes we reduce broadcasts to 2.1% of shared writes, even in the 16-processor case. A re-coding of the distributed loop as suggested above could hold the broadcast percentage to below 1%, even if the number of processors is scaled to well above 16. For MP3D a broadcast fraction of 1% of shared writes corresponds to 0.33 broadcasts per thousand references, which is low enough to be supportable in fairly large machines.

6.4 Distributed CSIM

Figures 19, 20 and 21 give the invalidation distributions of Distributed CSIM. We note that the number of shared writes is a much smaller fraction of all references than in the previous three applications. Furthermore, very few shared writes cause more than 2 invalidations. Note that this trace covers a section of code that does not have any synchronization at all, and this is why we do not show a further breakdown of the 16-processor distribution. The distributions we see are for shared data only. Most shared writes cause only zero or single invalidations.

The basic data objects of Distributed CSIM are the element and net structures. Some parts of these structures behave like mostly-read data (e.g., the activation flags) and some parts like migratory data (e.g., next input event pointers). The invalidation patterns vary accordingly.

The activation flag of an element is set as a processor changes one of the element's input values. Many processors can check this flag to see if an element is activated. Later, the element is evaluated and the activation flags are reset. While the *setting* of the activation flag causes only one invalidation, the *resetting* can cause many because many processors may have read the flag in the meantime. The resetting of the activation flags causes about 60% of the shared writes that result in more than single invalidations.

The next input event pointers, on the other hand, are used when an element is being evaluated, and are thus only read and written by one processor while it is updating the element. Hence we see mostly single invalidations – the pattern typical for migratory data.

Another factor that affects the number of invalidations is the connectivity of the circuit being evaluated. Nets that are connected to many elements, clock lines for example, are more likely to cause large invalidations when they are updated.

Figure 22 shows that Distributed CSIM is well suited for directory-based cache schemes. A single-pointer directory captures 17% of broadcasts and a second pointer diminishes this fraction to 3.2%. Further reduction of broadcasts could only be achieved if the program exploited processor locality in some way.

A scaling in the number of processors would result in a larger invalidation average per shared write, but not in more shared writes, since no synchronization objects are present in this portion of Distributed CSIM.

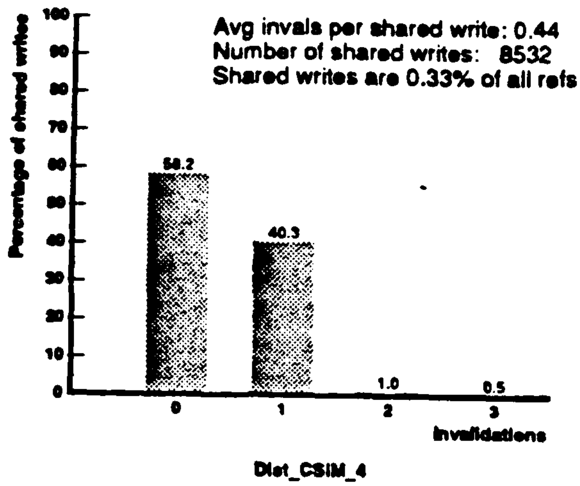


Figure 19: Distributed CSIM 4

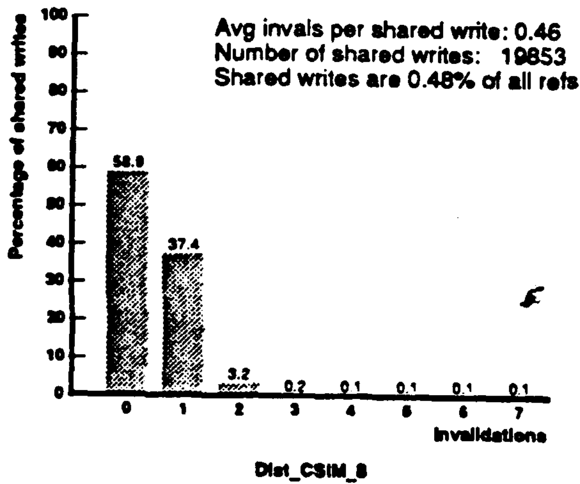


Figure 20: Distributed CSIM 8

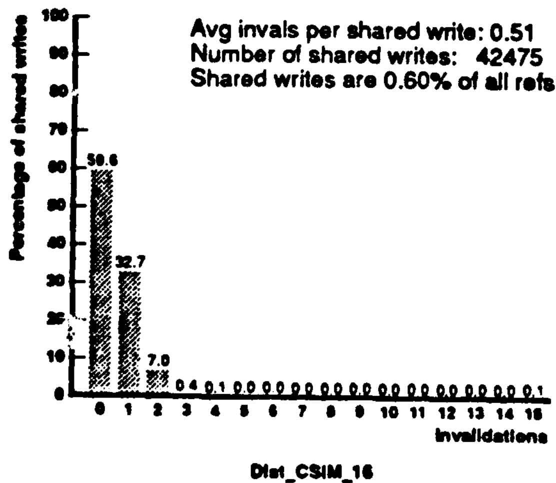


Figure 21: Distributed CSIM 16

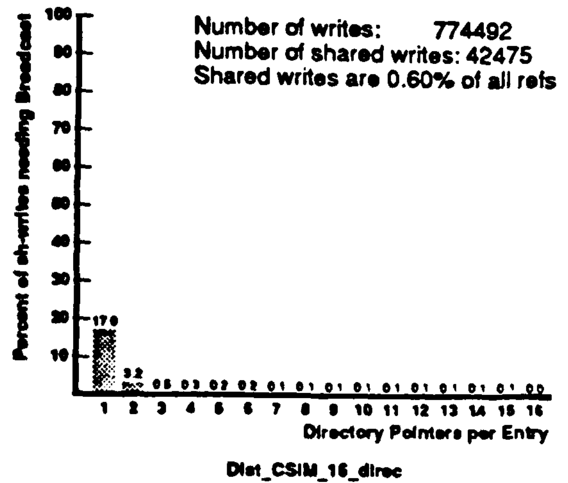


Figure 22: Distributed CSIM 16 Directory Performance

6.5 LocusRoute

Figures 23, 24 and 25 show the invalidation distributions for LocusRoute. It is noted that there are very few shared writes per reference. This shows how a well-designed parallel program can avoid excessive interprocess communication. Most of the invalidations are due to data objects. The only synchronization object that shows up is a lock used to control the access to the shared memory allocation routine (ShMalloc).

The single largest source of invalidations is the global cost array. It is a good example of mostly-read data. It is frequently read while testing different routes for a wire, but is written only when the wire route is decided. The average number of invalidations per shared write of the cost array is about 2 with 16 processors, but some writes can cause up to 6 invalidations, depending on how many processors have cached a given portion of the cost array (see Figure 27). Note that there are only 7400 shared writes to the cost array in the 7.7 million reference 16-processor trace.

Invalidations due to the ShMalloc lock are very infrequent in this portion of the program as the program keeps its own free lists and will have allocated most of its shared memory requirement by the time the trace was gathered. As contention for the lock is non-existant, all shared writes to the lock cause only zero or single invalidations (see Figure 26).

LocusRoute would be expected to scale well beyond 16 processors. The shared data is mostly-read and shared writes are very infrequent. As more processors are added, the average number of invalidations per shared write will increase slightly (because more processors are likely to have cached a given portion of the cost array), but the number of shared writes is not expected to increase.

7 Generalizations and Conclusions

We have proposed several classes of data objects that can be distinguished by their use in parallel programs and by their invalidation traffic patterns. By merging the invalidation behavior found in the applications discussed in the previous section, we can gain more general insights into the invalidation patterns of certain high-level constructs. We also have the opportunity to predict behavior beyond the 16 processor limit of the case studies.

Little needs to be said about code and read-only data. Since they are never written, they never cause invalidations. Some directory schemes do not allow a memory location to be present in more caches than there are entries (for example Dir_iNB schemes in [2]). This kind of scheme is not suitable for shared code and read-only data.

Migratory data objects move from processor to processor as execution progresses, but they are never manipulated by more than one processor at any one time. The node structures of Maxflow and the global arrays of MP3D are good examples of this data type. Migration of the data object causes at most single invalidations, because each processor writes to the object before relinquishing control of it. Single invalidations are expected, even as the number of processors is scaled. We note that a large number of these invalidations could be avoided if the processors were smart enough to flush the data items out of their cache when they are no longer needed. Hardware and operating system support for this feature seems desirable.

Synchronization primitives were found in all applications. In well-designed applications such as Distributed CSIM and LocusRoute, contention for the critical sections protected by the locks was minimized and this effectively reduced the invalidation traffic caused by the locks. It is seen

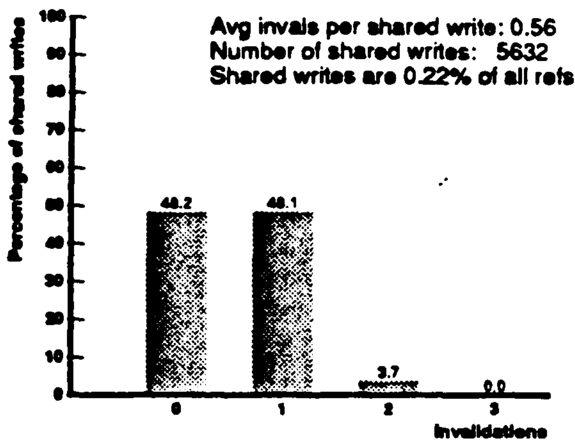


Figure 23: LocusRoute 4

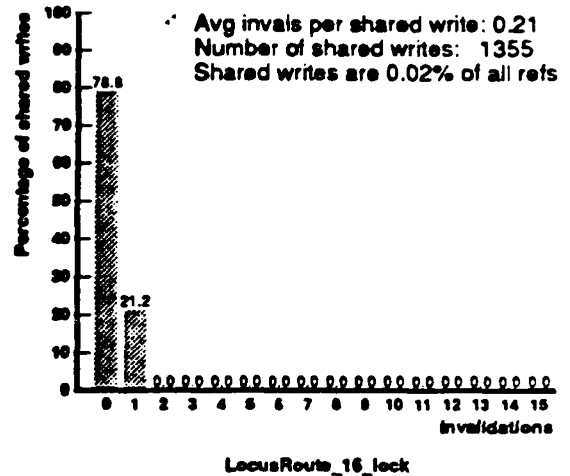


Figure 26: LocusRoute 16 (ShMalloc lock)

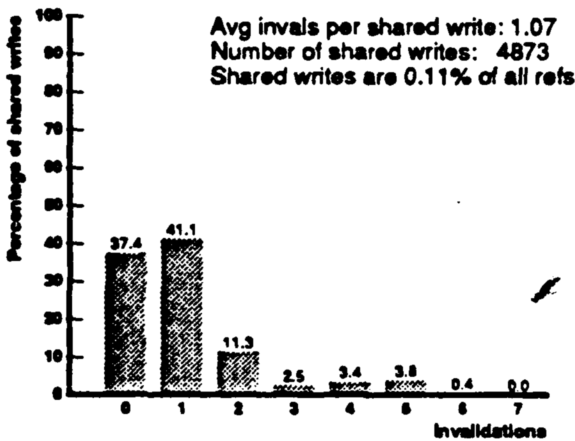


Figure 24: LocusRoute 8

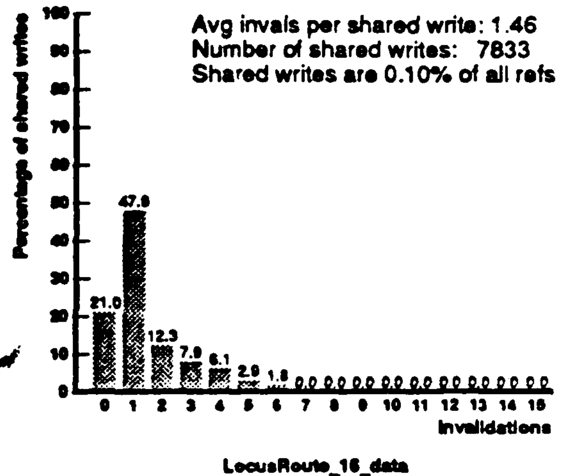


Figure 27: LocusRoute 16 (Data)

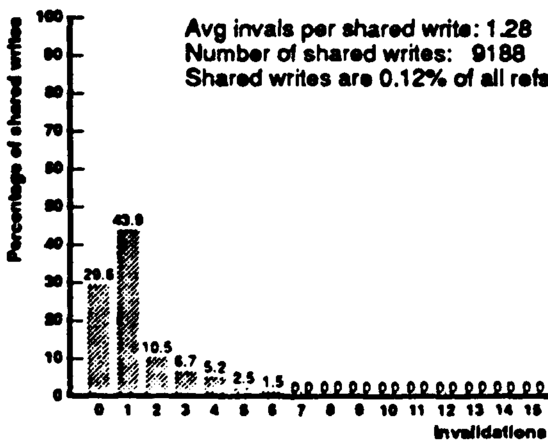


Figure 25: LocusRoute 16

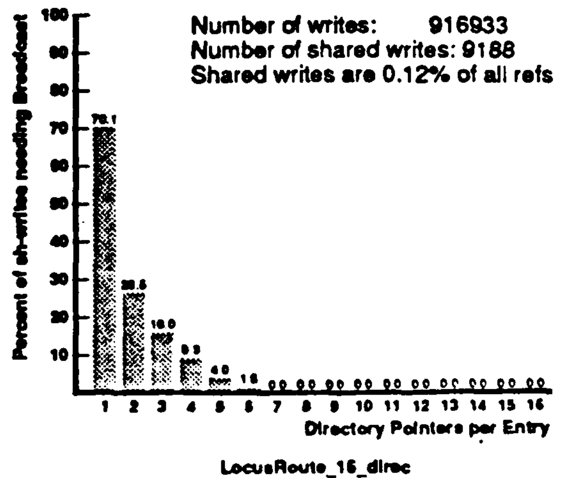


Figure 28: LocusRoute 16 Directory Performance

then, that proper program design will allow the use of locks without a large volume of invalidation traffic. As more processors are used, an ever increasing amount of effort will have to go into the program design to avoid contention over locks. Alternatively, a separate mechanism for dealing with synchronization traffic may be provided.

Mostly-read data such as the global cost array in LocusRoute has potential for causing a large number of invalidations, since each write is preceded by a number of reads from various processors. The average number of invalidations caused by each write is thus high. The good news is that writes to this kind of data tend to be relatively infrequent and hence the total invalidation traffic is not very large. With more processors, we expect an increase in the average number of invalidations per shared write, because it is likely that more processors will have touched the data object before a write to it takes place. Some of this effect may be mitigated by taking advantage of locality, i.e., assigning work in a local area of the problem to a relatively small section of the processors available. We are currently exploring such issues of locality, which we think will be critical in the design of highly scalable machines.

Frequently read/written data presents the largest problem in terms of invalidations. Not only does each write cause several invalidations, but writes are also frequent. A good example of this type of data is the variable in Maxflow that keeps track of how many processors are waiting on the global queue. Frequently read/written data will show increased invalidations as more processors are used, because more reads and more writes to the data item will take place. This type of data object should be avoided for parallel applications with large number of processors.

In summary, in this paper we have presented data about the invalidation patterns of five applications using 4, 8 and 16 processor traces. By classifying data objects, we are able to predict invalidation behavior beyond the number of processors currently traced. Such extrapolation suggests that directory-based cache schemes with just two or three pointers per entry can work in scalable multiprocessors, if the applications are well-designed. In particular, effort has to be put into limiting contention over synchronization objects and eliminating frequently read/written data objects.

8 Acknowledgments

We would like to thank Roberto Bisiani for letting us use his VAX-8350 at CMU and David Black, Robert Baron, and Mary Thompson for helping us with the inner details of the MACH operating system. We wish to thank Larry Soule, Jeff McDonald, Jonathan Rose, Mike Smith and Francisco Carrasco for letting us trace their applications, and for patiently explaining the details of the data structures used by them. We would like to thank Richard Sites of Digital Equipment Corporation, Hudson MA, for providing the VAX-8350 used for tracing at Stanford and for supporting Wolf-Dietrich Weber. Anoop Gupta is supported by DARPA contract N00014-87-K-0828 and by a faculty award from Digital Equipment Corporation.

References

[1] Anant Agarwal and Anoop Gupta.
Memory Reference Characteristics of Multiprocessor Applications under MACH.
In *ACM SIGMETRICS*. 1988.

[2] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz.
An Evaluation of Directory Schemes for Cache Coherence.
In *15th International Symposium on Computer Architecture*. 1988.

[3] Francisco Javier Carrasco.
A Parallel Maxflow Implementation.
March 1988.
CS411 - Final Project Report. Stanford University.

[4] M. Censier and P. Feautier.
A New Solution to Coherence Problems in Multicache Systems.
IEEE Transactions on Computers, C-27(12):1112–1118, December 1978.

[5] K. M. Chandy and J. Misra.
Asynchronous Distributed Simulation via a Sequence of Parallel Computations.
In *Communications of the ACM*, April 1981.

[6] David Cheriton.
Workform Processing: A Model and Language for Parallel Computation.
Stanford University, Computer Science Technical Report, 1986.

[7] Stephen R. Goldschmidt.
Simulating Multiprocessor Memory Traces.
December 1987.
EE390 Report. Stanford University.

[8] J.R. Goodman.
Using Cache Memory to Reduce Processor-Memory Traffic.
In *Proc. Tenth International Symposium on Computer Architecture*, pages 124–131, June 1983.

[9] Anoop Gupta, Charles Forgy, and Robert Wedig.
Parallel Algorithms and Architectures for Rule-Based Sytems.
In *Proc. 13th Int. Symp. of Computer Architecture*, June 1986.

[10] S. Kirkpatrick, C.D. Gelatt, and M. P. Vecchi.
Optimization by Simulated Annealing.
Science. 220(4580):671–680. May 1983.

[11] Lusk, Overbeek, et al.
Portable Programs for Parallel Processors.
Holt, Rinehart, and Winston Inc., 1987.

[12] Lusk, Stevens, and Overbeek.
A Tutorial on the Use of Monitors in C: Writing Portable Code for Multiprocessors.
Argonne National Laboratory, Argonne, Illinois 60439. 1986.

[13] Jeffrey D. McDonald.
A Direct Particle Simulation Method for Hypersonic Rarified Flow on a Shared Memory Multiprocessor.
March 1988.
CS411 - Final Project Report. Stanford University.

[14] Jeffrey D. McDonald and Donald Baganoff.
Vectorization of a Particle Simulation Method for Hypersonic Rarified Flow.
In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[15] Louis Monier and Pradeep Sindhu.
The Architecture of the Dragon.
In *Proc. Thirtieth IEEE Int. Conference*, pages 118–121. IEEE, Februrary 1985.

[16] R. Katz, S. Eggers, D. Wood, C. Perkins, and R. Sheldon.
Implementing a Cache Consistency Protocol.
In *12th International Symposium on Computer Architecture*, 1985.

[17] Jonathan Rose.
LocusRoute: A Parallel Global Router for Standard Cells.
In *Design Automation Conference*, pages 189–195, June 1988.

[18] Larry Rudolph and Zary Segall.
Dynamic Decentralized Cache Consistency Schemes for MIMD Parallel Processors.
In *Proc. 12th Int. Symp. on Computer Architecture*, pages 355–362. ACM SIGARCH, June 1985.
also SIGARCH Newsletter, Volume 13, Issue 3, 1985.

[19] C. Sechen and A. Sangiovanni-Vincentelli.
The Timberwolf Placement and Routing Package.
IEEE JSSC, SC-20(2):510–522, April 1985.

[20] Richard L. Sites and Anant Agarwal.
Multiprocessor Cache Analysis using ATUM.
In *Proc. 15th Annual International Symposium on Computer Architecture*, May 1988.

[21] Michael Smith and Wolf-Dietrich Weber.
Parallel Simulated Annealing.
March 1988.
CS411 - Final Project Report, Stanford University.

[22] Larry Soule and Tom Blank.
Parallel Logic Simulation on General Purpose Machines.
In *Design Automation Conference*, pages 166–171, June 1988.

[23] C. Thacker and L. Stewart.
Firefly: A Multiprocessor Workstation.
In *2nd Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, ACM, October 1987.

SPIM: A Pipelined 64 X 64 bit Iterative Multiplier

Mark R. Santoro
Mark A. Horowitz
Center for Integrated Systems
Stanford University
Stanford, CA. 94305
(415)725-3707

Abstract

A 64 by 64 bit iterating multiplier, SPIM (Stanford Pipelined Iterative Multiplier) is presented. The pipelined array consists of a small tree of 4:2 adders. The 4:2 tree is better suited than a Wallace tree for a VLSI implementation because it is a more regular structure. A 4:2 carry save accumulator at the bottom of the array is used to iteratively accumulate partial products, allowing a partial array to be used, which reduces area. SPIM was fabricated in a 1.6µm CMOS process. It has a core size of 3.8 X 6.5mm and contains 41 thousand transistors. The on chip clock generator runs at an internal clock frequency of 85MHz. The latency for a 64 X 64 bit fractional multiply is under 120ns, with a pipeline rate of one multiply every 47ns.

SPIM: A Pipelined 64 X 64 bit Iterative Multiplier

Mark R. Santoro
Mark A. Horowitz
Center for Integrated Systems
Stanford University
Stanford, CA. 94305

I. Introduction

The demand for high performance floating point coprocessors has created a need for high-speed, small-area multipliers. Applications such as DSP, graphics, and on chip multipliers for processors require fast area efficient multipliers. Conventional array multipliers achieve high performance but require large amounts of silicon, while shift and add multipliers require less hardware but have low performance. Tree structures achieve even higher performance than conventional arrays but require still more area.

The goal of this project was to develop a multiplier architecture which was faster and more area efficient than a conventional array. As a test vehicle for the new architecture a structure capable of performing the mantissa portion of a double extended precision (80 bit) floating point multiply was chosen. The multiplier core should be small enough such that an entire floating point co-processor, including a floating point multiplier, divider, ALU, and register file, could be fabricated on a single chip. A core size of less than $25mm^2$ was determined to be acceptable. This paper presents a 64 by 64 bit pipelined array iteratively accumulating multiplier, SPIM (Stanford Pipelined Iterative Multiplier), which

can provide over twice the performance of a comparable conventional full array at 1/4 of the silicon area.

II. Architectural Overview

Conventional array multipliers consist of rows of carry save adders (CSA) where each row of carry save adders sums up one additional partial product (see Figure 1).[1] Since intermediate partial products are kept in carry save form there is no carry propagate, so the delay is only dependent upon the depth of the array and is independent of the partial product width. Although arrays are fast, they require large amounts of hardware which is used inefficiently. As the sum is propagated down through the array, each row of carry save adders is used only once. Most of the hardware is doing no useful work at any given time. Pipelining can be used to increase hardware utilization by overlapping several calculations. Pipelining greatly increases throughput, but the added latches increase both the required hardware, and the latency.

Since full arrays tend to be quite large when multiplying double or extended precision numbers, chip designers have used partial arrays and iterated using the system clock. This structure has the benefit of reducing the hardware by increasing utilization. At the limit, an iterative structure would have one row of carry save adders and a latch. Figure 2 shows a minimal iterative structure. Clearly, this structure requires the least amount of hardware and has the highest utilization since each CSA is used every cycle. An important observation is that iterative structures can be made fast if the latch delays are small, and the clock is matched to the combinational delay of the carry save

[1] Carry save adders are also often referred to as full adders or 3:2 adders.

adders. If both of these conditions are met the iterative structure approaches the same throughput and latency as the full array. This structure does, however, require very fast clocks. For a 2µm process clocks may be in the 100MHz range. A few companies use iterative structures in their new high performance floating point processors [5].

In an attempt to increase performance of the minimal iterative structure additional rows of carry save adders could be added, resulting in a bigger array. For example, addition of a row of CSA cells to the minimal structure would yield a partial array with two rows of Carry Save Adders. This structure provides two advantages over the single row of CSA cells: it reduces the required clock frequency, and requires only half as many latch delays.[2] One should note, however, that although we doubled the number of carry save adders, the latency was only reduced by halving the number of latch delays. The number of CSA delays remains the same. Increasing the depth of the partial array by simply adding additional rows of carry save adders in a conventional structure yields only a slight performance increase. This small reduction in latency is the result of reducing the number of latches.

To increase the performance of this iterative structure we must make the CSA cells fast and, more importantly, decrease the number of series adds required to generate the product. Two well known methods for the latter are Booth encoding and tree structures [2][9]. Modified Booth encoding, which halves the number of series adds required, is used on most modern floating point chips,

[2]In fact one rarely finds a multiplier array that consists of only a single row of carry save adders. The latch overhead in this structure is extremely high.

including SPIM [7][8]. Tree structures reduce partial products much faster than conventional methods, requiring only order logN CSA delays to reduce N partial products (see Figure 3). Though trees are faster than conventional arrays, like conventional arrays they still require one row of CSA cells for each partial product to be retired. Unfortunately, tree structures are notoriously hard to lay out, and require large wiring channels. The additional wiring makes full trees even larger than full arrays. This has caused designers to look at permutations of the basic tree structure [1][11]. Unbalanced or modified trees make a compromise between conventional full arrays and full tree structures. They reduce the routing required of full trees but still require one row of carry save adders for each partial product. Ideally one would want the speed benefits of the tree in a smaller and more regular structure. Since high performance was a prerequisite for SPIM a tree structure was used. This left two problems. The first, was the irregularity of commonly used tree structures. The second problem was the large size of the trees.

Wallace [9], Dadda [4], and most other multiplier trees use a carry save adder as the basic building block. The carry save adder takes 3 inputs of the same weight and produces 2 outputs. This 3:2 nature makes it impossible to build a completely regular tree structure using the CSA as the basic building block. A binary tree has a symmetric and regular structure. In fact, any basic building block which reduces products by a factor of two will yield a more regular tree than a 3:2 tree. Since a more regular tree structure was needed the solution was to introduce a new building block: the 4:2 adder, which reduces 4 partial products of the same weight to 2 bits. Figure 4 is a block diagram of the 4:2 adder. The truth table for the 4:2 adder is shown in Table 1. Notice that the 4:2

adder actually has 5 inputs and 3 outputs. It is different from a 5:3 counter which takes in 5 inputs of the same weight and produces 3 outputs of different weights. The sum output of the 4:2 has weight 1 while the Carry and Cout both have the same weight of 2. In addition, the 4:2 is not a simple counter as the Cout output must NOT be a function of the Cin input or a ripple carry could occur. As for the name, 4:2 refers to the number of inputs from one level of a tree and the number of outputs produced at the next lower level. That is, for every 4 inputs taken in at one level, two outputs are produced at the next lower level. This is analogous to the binary tree in which for every 2 inputs 1 output is produced at the next lower level. The 4:2 adder can be implemented directly from the truth table, or with two carry save add (CSA) cells as in Figure 5.[3]

A 4:2 tree will reduce partial products at a rate of $\log_2(N/2)$ whereas a Wallace tree requires $\log_{1.5}(N/2)$; where N is the number of inputs to be reduced. Though the 4:2 tree might appear faster than the Wallace tree, the basic 4:2 cell is more complex so the speed is comparable. The 4:2 structure does however yield a tree which is much more regular. In addition the 4:2 adder has the advantage that two Carry Save Adders are in each pipe in place of one. This reduces both the required clock frequency and the latch overhead.

To overcome the size problem SPIM uses a partial 4:2 tree, and then iteratively accumulates partial products in a carry save accumulator to complete the computation. The carry save accumulator is simply a 4:2 adder with two of the

[3]SPIM implemented the 4:2 adder with two CSA cells because it permits a straight forward comparison with other architectures on the basis of CSA delays. By knowing the size and speed of the CSA cells in any technology a designer can predict the size and speed advantages of this method over that currently used.

inputs used to accumulate the previous outputs. The carry save accumulator is much faster than a carry propagate accumulator and requires only one additional pipe stage.

Figure 6 compares a single 4:2 adder with carry save accumulator, to a conventional partial piped array.[4] Both structures reduce 4 partial products per cycle. Notice, however, that the tree structure is clocked at almost twice the frequency of the partial piped array. It has only 2 CSA cells per pipe stage, whereas the partial piped array has 4. Consequently, the partial array would require 32 CSA delays to reduce 32 partial products where the tree structure would need only 18 CSA delays. Using the 4:2 adder with carry save accumulator is almost twice as fast as the partial piped array, while using roughly the same amount of hardware.

The 4:2 adder structure can be used to construct larger trees, further increasing performance. In Figure 7 we use the same 4:2 adder structure to form an 8 input tree. This allows us to reduce 8 partial products per cycle. Notice that we still pipeline the tree after every 2 carry save adds (each 4:2 adder). In contrast, if we clocked the tree every 4 carry save adds it would double the cycle time and only decrease the required number of cycles by one. The overall effect would be a much slower multiply.

[4]In figures 6, 7, and 9 the detailed routing has not been shown. Providing the exact detailed routing, as was done in figure 5, would provide more information; however, it would significantly complicate the figures and would tend to obscure their purpose, which is to show the data flow in terms of pipe stages and carry save add delays.

Figure 8 shows the size and speed advantages of different sized 4:2 trees with carry save accumulators vs. conventional partial arrays. This plot is a price/performance plot where the price is size and the performance is speed (latency = 1/speed). The plot assumes we are doing a 64 X 64 bit multiply. Booth encoding is used, thus we must retire 32 partial products. Size has been normalized such that 32 rows of CSA cells (a full array) has a size of 1 unit.[5] In the upper left corner is the structure using only 2 rows of CSA cells. In this case the tree and conventional structures are one and the same and can be seen as a partial array 2 rows deep, or as a 2 input partial tree. We can see that adding hardware to form larger partial arrays provides very little performance improvement. A full array is only 15% faster than the iterative structure using 2 rows of carry save adders. Adding hardware in a tree type structure however, dramatically improves performance. For example, using a 4 input tree, which uses 4 rows of carry save adders, is almost twice as fast as the 2 input tree. Using an 8 input tree is almost 3 times as fast as a 2 input tree and only 1/4 the size of the full array.

The latency of the multiplier is determined by the depth of the partial 4:2 tree and the fraction of the partial products compressed each cycle. The latency is equal to $\log_2(K/2) + (N/K)$ where N is the operand size and K is the partial tree size. If Booth encoding is used N would be one half the operand size since Booth encoding has already provided a factor of 2 compression. Startup times and pipe stages before the tree must also be taken into account when determining latency. We choose the 8 input piped tree with Booth encoding for

[5]Latency is in terms of CSA delays. We have assumed a latch is equivalent to 1/3 of a CSA delay in an attempt to take the latch delays into account. Size is the number of CSA cells used. It does not include the latch or wiring area.

SPIM, as we felt this provided best area speed tradeoff for our purpose. The number of cycles required to reduce 64 bits using Booth encoding and an 8 bit tree is:

$$\log_2(8/2) + (32/8) + \text{one cycle overhead} = 7 \text{ cycles}$$[6]

III. SPIM Implementation

Figure 9 is a block diagram of the SPIM data path. The Booth encoders, which encode 16 bits per cycle, are to the left of the data path. The Booth encoded bits drive the Booth select muxes in the A and B block. The A and B block Booth select mux outputs drive an 8 input tree structure constructed of 4:2 adders which are found in the A, B, and C blocks. Each pipe stage uses one 4:2 adder which consists of two carry save adders. The D block is a carry save accumulator. It also contains a 16 bit hard wired right shift to align the partial sum from the previous cycle to the current partial sum to be accumulated.

Figure 10 is a die photograph of SPIM. The A block inputs are pre-shifted allowing the A block to be placed on top of B block. Using 4:2 adders in a partial tree allows the array to be efficiently routed, and laid out as a bit slice, thus making the SPIM array a very regular structure. Interestingly, the CSA cells occupy only 27% of the core area. The Booth select muxes used in the A and B blocks make these blocks three times as large as the C block. Each Booth mux with it's corresponding latch is larger than a single carry save adder. Also, due to the routing required for the 16 bit shift, the D block is twice as large as the C block. The array area can be split into four main components; routing,

[6]The one cycle overhead is used for the Booth select muxes.

CSA cells, muxes, and latches. The routing required 20% of the area, while the other 75% was equally split between the CSA cells, muxes, and latches.

The critical path in the SPIM data path is through the D block. The D block contains the slowest path because of the added routing at the output, and the additional control mux at its input. The input mux is needed to reset the carry save accumulator. It selects "0" to reset, or the previous shifted output when accumulating. The final critical path through the D block includes 2 CSA cells, a master slave latch, a control mux, and the drive across 16 bits (128μm) of routing.

IV. Clocking

The architecture of SPIM yields a very fast multiply; however, the speed at which the structure runs demands careful attention to clocking issues. Only two carry save adders (one 4:2 adder) are found in each pipe stage, yielding clock rates on the order of 100MHz. The typical system clock is not fast enough to be useful for this type of structure. To produce a clock of the desired frequency SPIM uses a controllable on chip clock generator. The clock is generated by a stoppable ring oscillator. The clock is started when a multiply is initiated, and stopped when the array portion of the multiply has been completed. The use of a stoppable clock provides two benefits. It prevents synchronization errors from occurring and it saves power as the entire array is powered down upon completing a multiply. The actual clock generator used on SPIM is shown in Figure 11. It has a digitally selectable feedback path which provides a programmable delay element for test purposes. This allows the clock frequency

to be tuned to the critical path delay. In addition, the clock generator has the ability to use an external test clock in place of the fast internally generated clock.

When a multiply signal has been received a small delay occurs while starting up the clocks. This delay comes from two sources. The first is the logic which decodes the run signal and starts up the ring oscillator. The second source of delay is from the long control and clock lines running across the array. They have large capacitive loads and require large buffer chains to drive them. The simulated delay of the buffer chain and associated logic is 6ns, almost half a clock cycle. Since the inputs are latched before the multiply is started, SPIM does the first Booth encode before the array clocks become active (cycle 0). Thus, the startup time is not wasted. After the clocks have been started SPIM requires seven clock cycles (cycles 1-7) to complete the array portion of a multiply.

The detailed timing is shown in Table 2. In the time before the clocks are started (cycle 0) the first 16 bits are Booth encoded. During cycle 1, the first 16 Booth-coded partial products from cycle 0 are latched at the input of the array. The next four cycles are needed to enter all 32 Booth-coded partial products into the array. Two additional cycles are needed to get the output through the C and D blocks. If a subsequent multiply were to follow it would have been started on cycle 4, giving a pipelined rate of 4 cycles per multiply. When the array portion of the multiply is complete the carry save result is latched, and the run signal is turned off. Since the final partial sum from the D block is latched into the carry propagate adder only every fourth cycle, several cycles are available to stop the clock without corrupting the result.

The clock generator is located in the lower left hand side of the die (see Figure 10). The clock signal runs up a set of matched buffers, along the side of the array, which are carefully tuned to minimize skew across the array. Wider than minimum metal lines are used on the master clock line to reduce the resistance of the clock line relative to the resistance of the driver. The clock and control lines driven from the matched buffers then run across the entire width of the array in metal.

V. Test Results

To accurately measure the internal clock frequency the clock was made available at an output allowing an oscilloscope to be attached. SPIM was then placed in continuous (loop) mode where the clock is kept running and multiplies are piped through at a rate of one multiply every 4 cycles. Since the clock is continuously running its frequency can be accurately determined.

Three components determine the actual performance of SPIM. The startup time, when the clocks are started and the first Booth encode takes place (cycle 0), the array time, which includes the time through the partial array plus the accumulation cycles (cycles 1-7), and tha carry propagate addition time, when the final carry propagate addition converts the carry save form of the result from the accumulator to a simple binary representation. Due to limitations in our testing equipment only the array time could be accurately measured. Since the array time requires 7 cycles, and the array clock frequency was 85MHz the array time is simply 7 * (1/85MHz) = 82.4ns. The startup and cpadd times,

based upon simulations, were 6ns and 30ns respectively. In flowthrough mode the total latency is simply the sum of the startup time (6ns), the array time (82.4ns), and the cpadd time (30ns), for a total of 118.4ns. Thus SPIM has a total latency under 120ns. SPIM has a throughput of one multiply every 4 cycles or 4 * (1/85MHz) = 47ns, for a maximum pipelined rate in excess of 20 million 80 bit floating point multiplies per second.

The performance range of the parts tested was from 85.4MHz to 88.6MHz at a room temperature of 24.5 ^{o}C and a supply voltage of 4.9 volts. One of the parts was tested over a temperature range of 5 to 100 ^{o}C. At 5 ^{o}C it ran at 93.3MHz with speeds of 88.6MHz and 74.5MHz at 25 and 100 ^{o}C. The average power consumed at 85MHz was 72mA while an average of only 10mA was consumed in standby mode.

VI. Future Improvements

The Booth select muxes with their corresponding latches account for 38% of the array area. This was larger than expected. Though Booth encoding reduces the number of partial products by a factor of two, the same result could be achieved by adding one more level of 4:2 adders to the tree. Since much of the routing already exists for the Booth muxes, adding another level to the tree requires replacing each two Booth select muxes with a 4:2 adder and 4 AND gates (see Figure 12). Since the CSA cells are slightly larger than the Booth select muxes the array size will grow slightly, (by about 7%). However, if we take the whole picture into account, the core would remain about the same size, as we would no longer need the Booth encoders. Replacing the Booth

encoders and Booth select muxes with an additional level to the tree would also reduce the latency by one cycle from 7 cycles to 6. This occurs because the cycle required to Booth encode is now no longer needed. There are other advantages in addition to the increase in speed. Perhaps the greatest gain is the reduction in complexity. Both the Booth encoders and Booth select muxes are now unnecessary, thus the number of cells has been reduced. In addition, Booth encoding generates negative partial products. An increase in complexity results in the need to handle the negative partial products correctly. Replacing the Booth encoders with an additional level of 4:2 adders would remove the negative partial products. Our observation is that an increase in speed and reduction in complexity can be obtained with little or no increase in area.[7]

SPIM uses full static master slave latches for testing purposes. These latches are quite large, accounting for 27% of the array size. In addition they are slow, requiring 25% of the cycle time. Since the SPIM architecture has been proven, these latches are not required on future versions. One obvious choice is simply to replace the full static master slave version with dynamic latches. Another option is to split the master slave latches into two separate half latches and incorporate them into the CSA cells. This would reduce area and increase speed. A still more efficient structure, is the use of single phase dynamic latches. The balanced pipe nature of the multiplier makes the use of single phase latches possible. Since only half as many latches are required in the

[7]Replacing the Booth encoders and select muxes with an additional level of 4:2 compressors is a viable alternative on more conventional, i.e. non-piped and non iterative, trees as well. The non-pipelined speed gain depends upon the relative speed of the Booth encode plus Booth select mux vs. the delay through one 4:2 compressor and a NAND gate.

pipe, single phase dynamic latches would reduce the cycle time and decrease latch area.

Research on piped 4:2 trees and accumulators has continued. A test circuit consisting of a new clock generator and an improved 4:2 adder has been fabricated in a 0.8µm CMOS technology. Preliminary test results have demonstrated performance in the range of 400MHz.

VII. Conclusion

SPIM was fabricated in a 1.6µm CMOS process through the DARPA MOSIS fabrication service. It ran at an internal clock speed of 85MHz at room temperature. The latency for a 64 X 64 bit fractional multiply is under 120ns. In piped mode SPIM can initiate a multiply every 4 cycles (47ns), for a throughput in excess of 20 million multiplies per second. SPIM required an average of 72mA at 85MHz, and only 10mA in standby mode. SPIM contains 41 thousand transistors with a core size of 3.8 X 6.5mm, and an array size of 2.9 X 5.3mm.

The 4:2 adder yields a tree structure which is as efficient and far more regular than a Wallace type tree and is therefore better suited for a VLSI implementation. By using a partial 4:2 tree with a carry save accumulator a multiplier can be built which is both faster and smaller than a comparable conventional array. Future designs implemented in a 0.8µm CMOS technology should be capable of clock speeds approaching 400MHz.

Acknowledgements

The development of SPIM was partially supported by the Defense Advanced Project Research Agency (DARPA) under contracts MDA903-83-C-0335 and N00014-87-K-0828. Fabrication support through MOSIS is also gratefully acknowledged.

References

[1] S. F. Anderson, J. G. Earle, et al., "The IBM system/360 Model 91: Floating-Point Execution Unit", IBM Journal, VOL. 11, NO. 1, pp. 34-53, January 1967.

[2] A. D. Booth, "A Signed Binary Multiplication Technique", Qt. J. Mech. Appl. Math., Vol. 4, Part 2, 1951.

[3] J. F. Cavanagh, "Digital Computer Arithmetic Design and Implementation", McGraw-Hill, 1984.

[4] L. Dadda, "Some Schemes for Parallel Multipliers," Alta Frequenza, Vol. 34, No. 5, pp. 349-356, March 1965.

[5] B. Elkind, J. Lessert, J. Peterson, and G. Taylor, "A Sub 10 ns Bipolar 64 Bit Integer/Floating Point Processor Implemented on Two Circuits", IEEE Bipolar Circuits and Technology Meeting, pp. 101-104, September 1987.

[6] K. Hwang, "Computer Arithmetic: Principles, Architecture, and Design", New York, John Wiley & Sons, 1979.

[7] P. Y. Lu, et al., "A 30-MFLOP 32b CMOS Floating-Point Processor", IEEE Solid-State Circuits Conference proceedings Vol. XXXI, pp. 28-29, February 1988.

[8] W. McAllister and D. Zuras, "An nMOS 64b Floating Point Chip Set", IEEE Int. Solid-State Circuits conf., pp. 34-35, February 1986.

[9] C. S. Wallace, "A Suggestion for Fast Multipliers", IEEE Trans. Electronic Computers, Vol. EC-13, pp. 14-17, February 1964.

[10] S. Waser, and M. J. Flynn, "Introduction to Arithmetic for Digital Systems Designers", New York, CBS Publishing, 1982.

[11] D. Zuras, and W. McAllister, "Balanced Delay Trees and Combinatorial Division in VLSI", IEEE Journal of Solid-State Circuits, VOL. sc-21, no. 5, October 1986.

11/10/88
SPIM
Santoro and Horowitz

Captions for Tables

Table 1: Truth table for the 4:2 adder.

where:
n is number of inputs (from In1, In2, In3, In4) which = 1
Cin is the input carry from the Cout of the adjacent bit slice
Cout and Carry both have weight 2
Sum has weight 1

NOTES:

* Either Cout or Carry may be "1" for 2 or 3 inputs equal to 1 but NOT both.

  Cout may NOT be a function of the Cin from the adjacent block or a ripple carry may occur.

Table 2: SPIM pipe timing. Numbers indicate which partial products are being reduced. 0 is the least significant bit.

n	Cin	Cout	Carry	Sum
0	0	0	0	0
1	0	0	0	1
2	0	*	*	0
3	0	*	*	1
4	0	1	1	0
0	1	0	0	1
1	1	0	1	0
2	1	*	*	1
3	1	1	1	0
4	1	1	1	1

Table 1
Santoro and Horowitz

Action \ Cycle	0	1	2	3	4	5	6	7
Booth Encode	startup 0-15	16-31	32-47	48-63				
A and B block Booth Muxs		0-15	16-31	32-47	48-63			
A Block CSA's			0-7	16-23	32-39	48-55		
B Block CSA's			8-15	24-31	40-47	56-63		
C Block				0-15	16-31	32-47	48-63	
D Block					clear 0-15	16-31	32-47	48-63

Table 2
Santoro and Horowitz

11/10/88
SPIM
Santoro and Horowitz

Figure Captions

Figure 1. Conventional Array Multiplier. Shaded areas represent intermediate partial product flowing down array.

Figure 2. Minimal Iterative Structure using a single row of carry save adders. Black bars represent latches.

Figure 3. A conventional structure (a) has depth proportional to N, while a tree structure (b) has depth proportional to logN.

Figure 4. Block diagram of a 4:2 adder.

Figure 5. A 4:2 adder implemented with two carry save adders.

Figure 6. With the same 4 CSA cells a 4 input partial tree structure with a carry save accumulator (a) will attain almost twice the throughput of a partial piped array (b). In (a) the carry save accumulator is placed under the 4:2 adder.

Figure 7. An 8 input tree constructed from 4:2 adders can reduce 8 partial products per cycle.

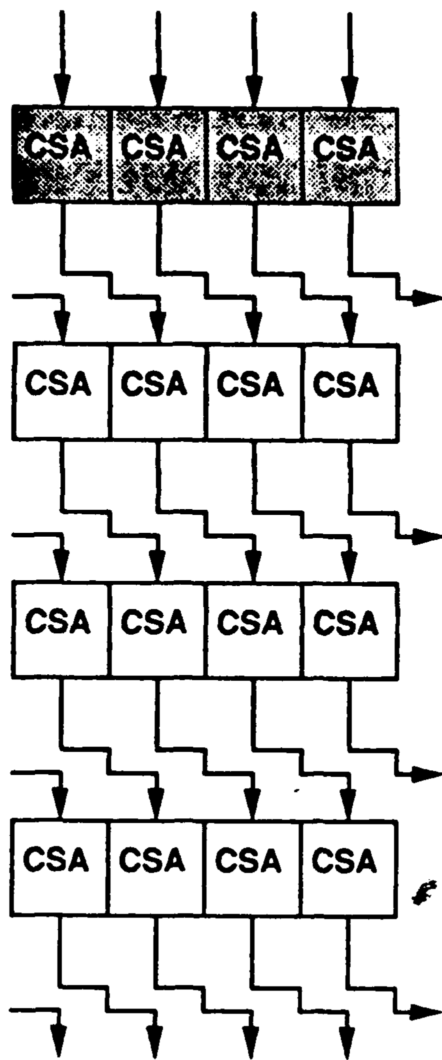
Figure 8. Architectural comparison of piped partial tree structure with carry save accumulator vs. conventional partial array.

Figure 9. The SPIM Data Path.

Figure 10. Microphotograph of SPIM.

Figure 11. SPIM clock generator circuit.

Figure 12. Booth encoding vs. additional tree level. The Booth encoders and Booth select muxes (a) can be replaced with an additional level of 4:2 adders and AND gates (b).



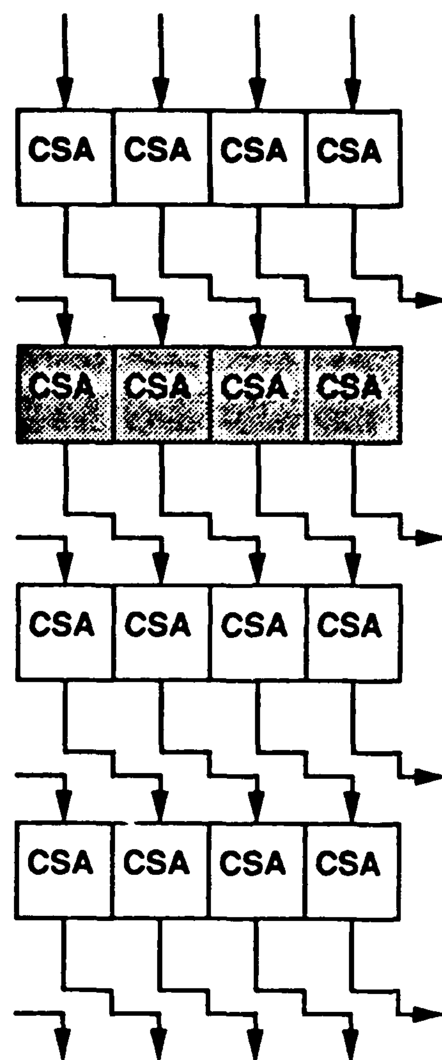


Figure 1
Santoro and Horowitz

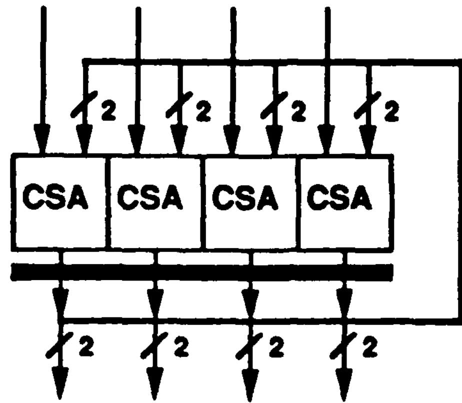


Figure 2
Santoro and Horowitz

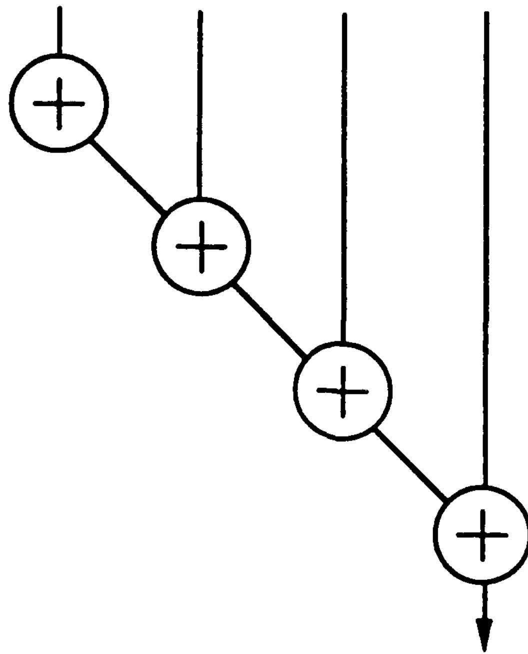


Figure 3a
Santoro and Horowitz

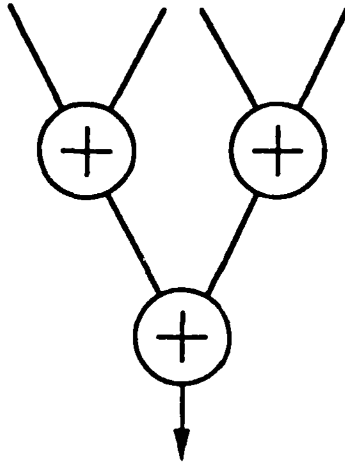


Figure 3b
Santoro and Horowitz

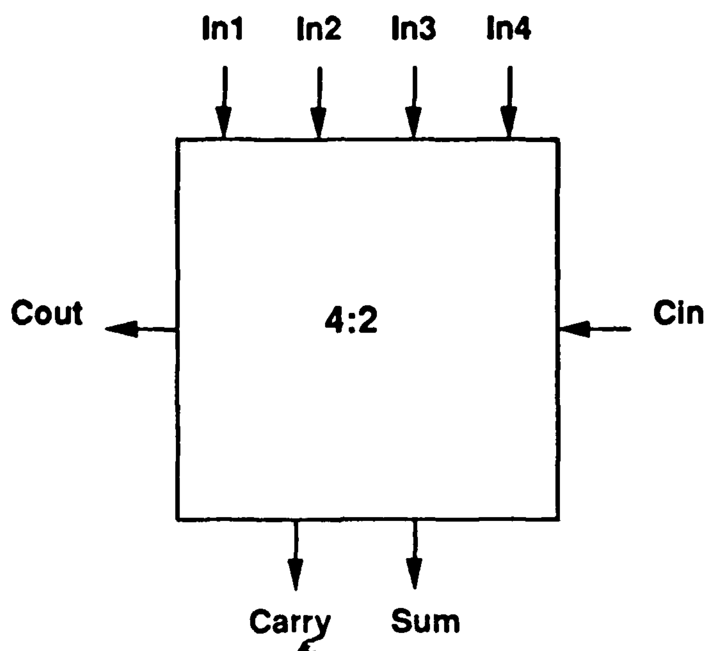


Figure 4
Santoro and Horowitz

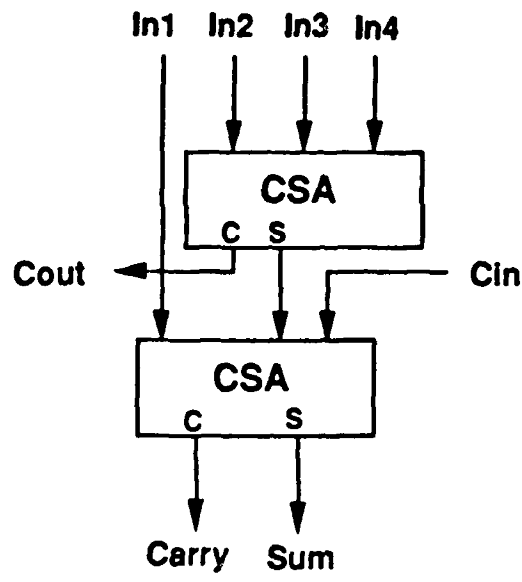


Figure 5
Santoro and Horowitz

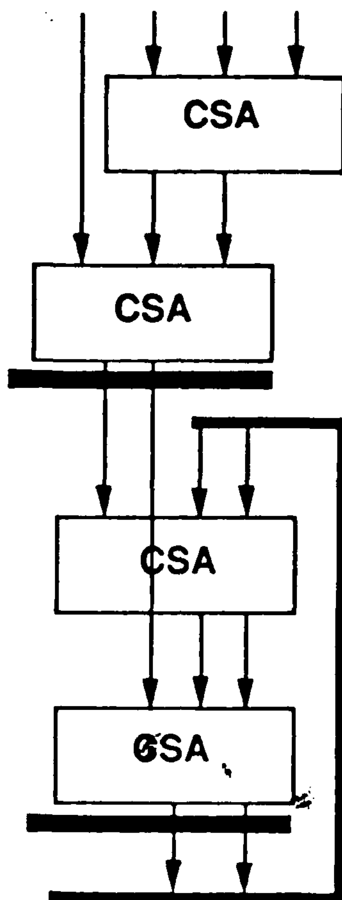


Figure 6a
Santoro and Horowitz

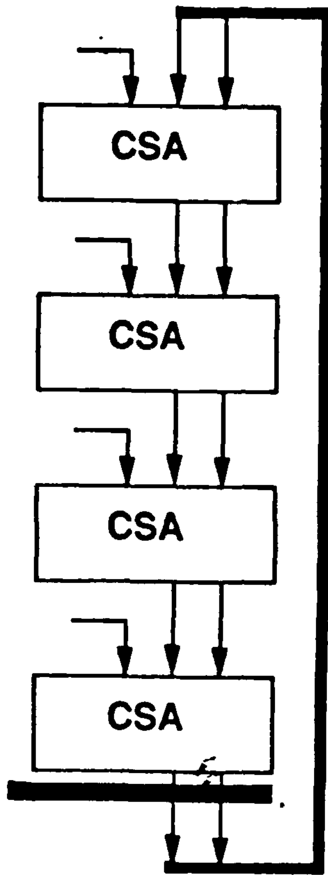


Figure 6b
Santoro and Horowitz

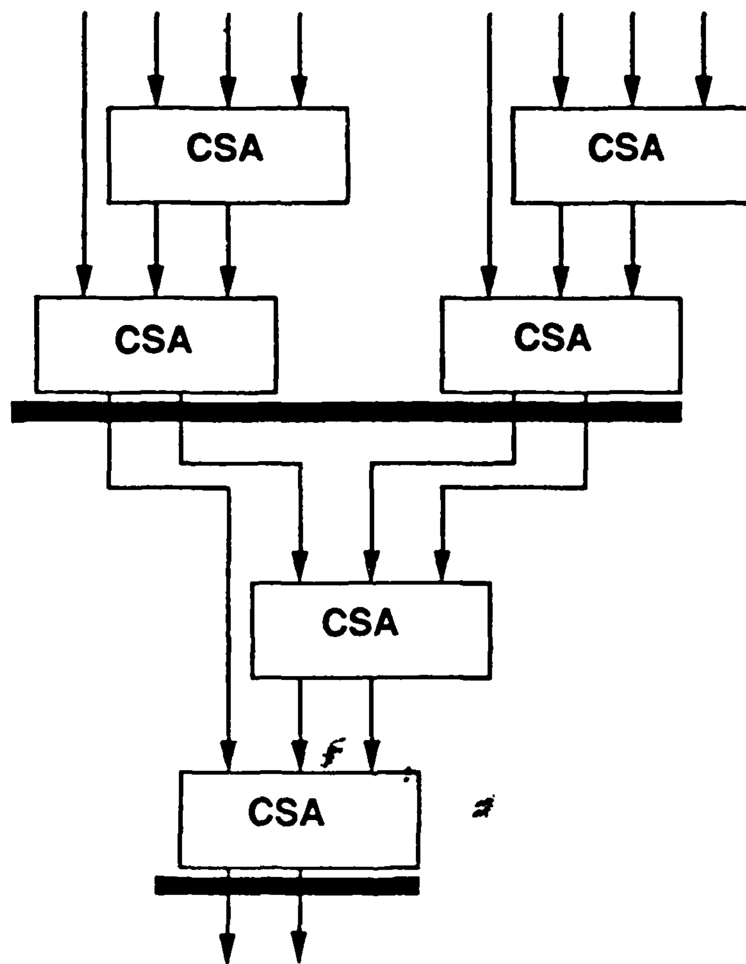


Figure 7
Santoro and Horowitz

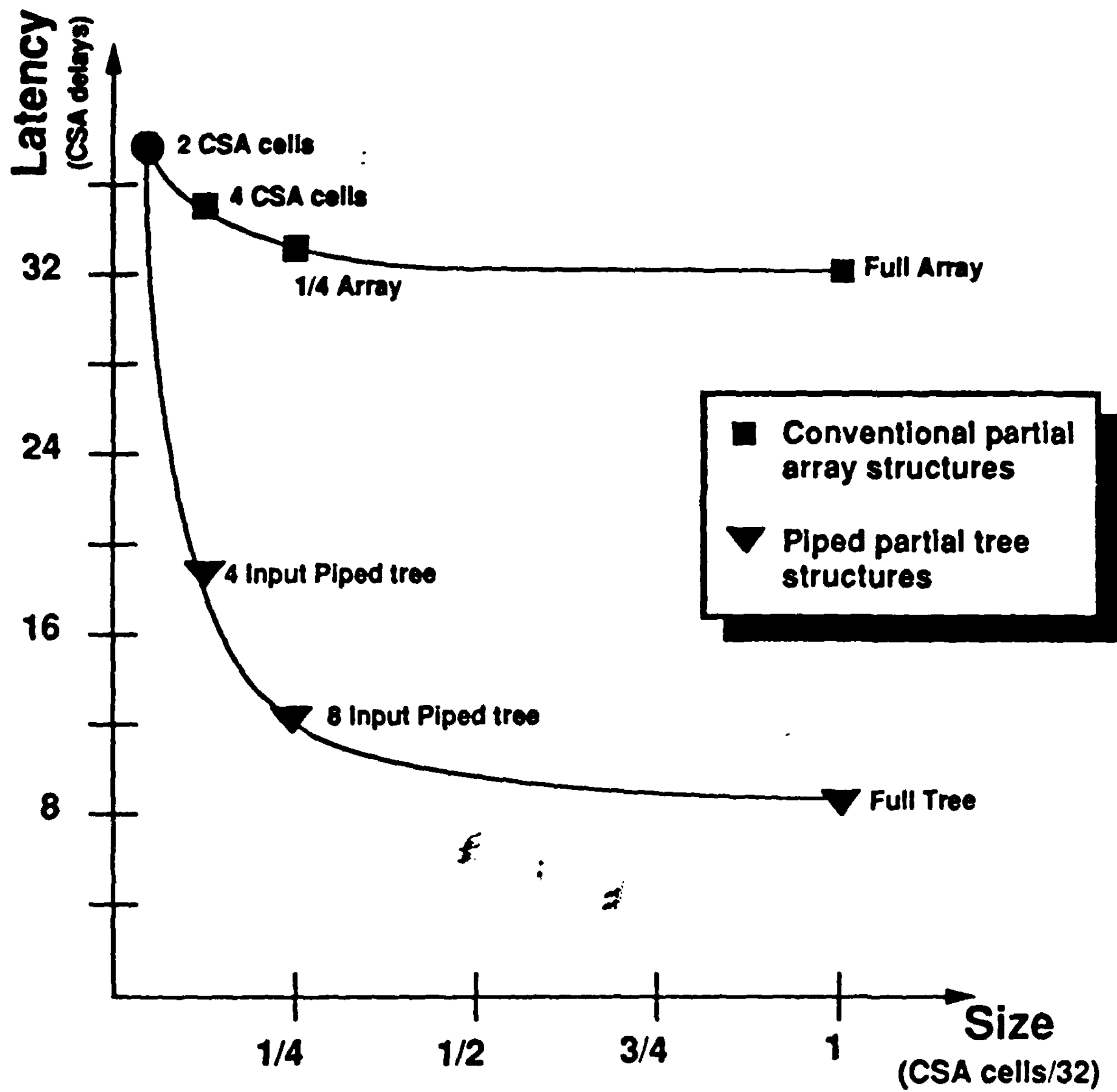


Figure 8
Santoro and Horowitz

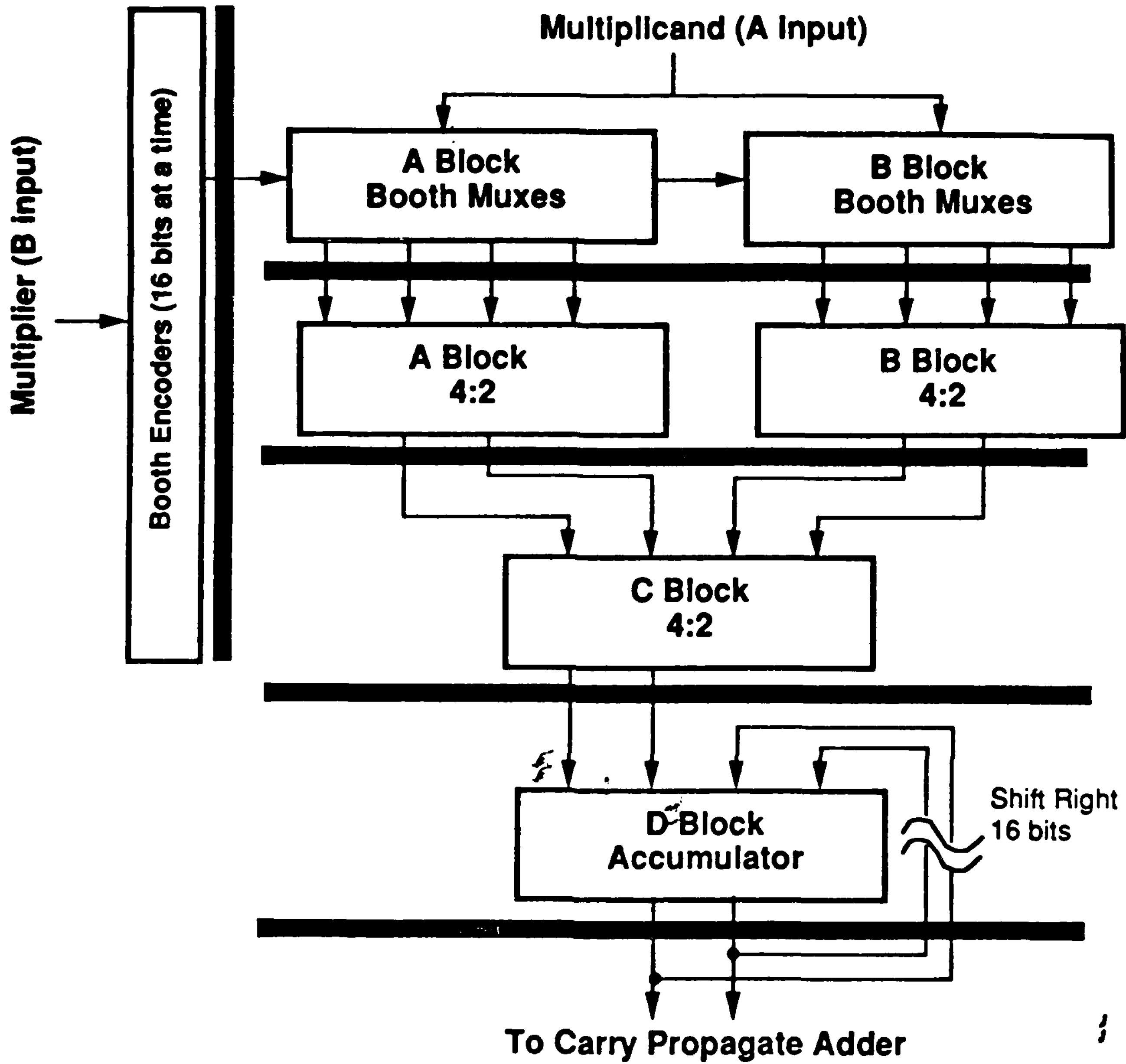


Figure 9
Santoro and Horowitz

B input latches
Booth Encoders
Control and Clock Buffers
Control
Clocking
A input latches
A Block
B Block
C Block
D Block
CPAdder and Output bus drivers

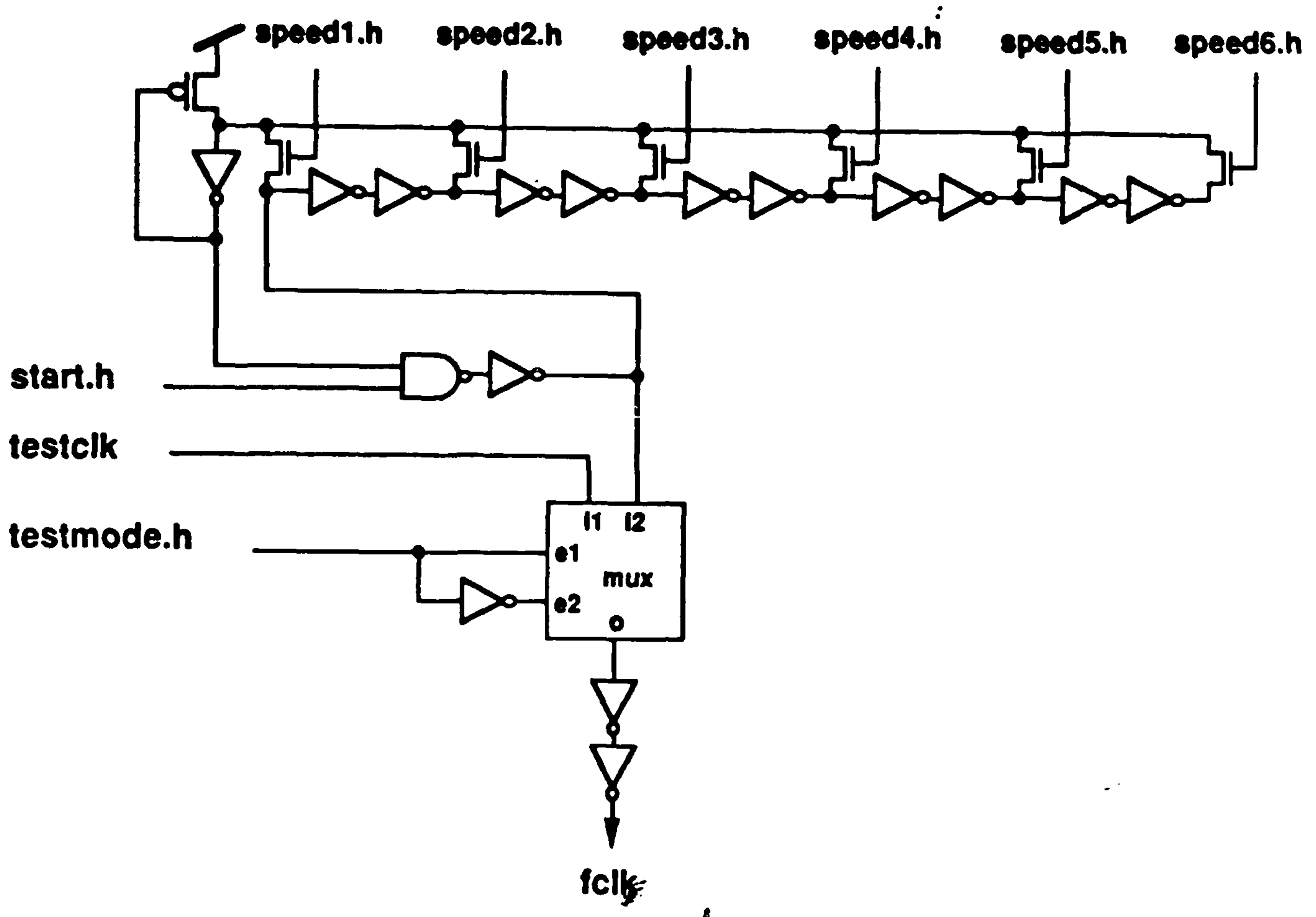


Figure 11
Santoro and Horowitz

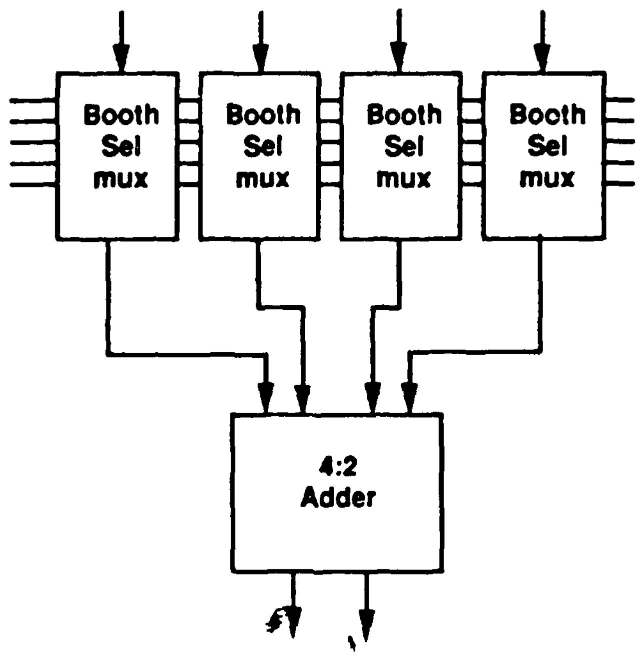


Figure 12a
Santoro and Horowitz

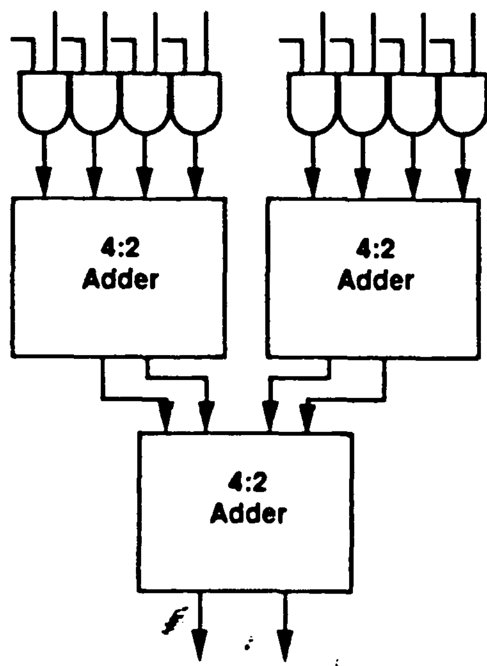


Figure 12b
Santoro and Horowitz

Characterization of Parallelism and Deadlocks in Distributed Digital Logic Simulation

Larry Soule and Anoop Gupta
Computer Systems Laboratory
Stanford University, CA 94305

Draft of November 3. 1988

Abstract

This paper explores the suitability of the Chandy-Misra algorithm for digital logic simulation. We use four realistic circuits as benchmarks for our analysis, with one of them being the vector-unit controller for the Titan supercomputer from Ardent. Our results show that the average number of logic elements available for concurrent execution ranges from 6.2 to 92 for the four circuits, with an overall average of 50. Although this is twice as much parallelism as that obtained by traditional event-driven algorithms, we feel it is still too low. One major factor limiting concurrency is the large number of global synchronization points — "deadlocks" in the Chandy-Misra terminology — that occur during execution. Towards the goal of reducing the number of deadlocks, the paper presents a classification of the types of deadlocks that occur during digital logic simulation. Four different types are identified and described both intuitively in terms of circuit structure and formally with equations. Using domain specific knowledge, the paper proposes methods for reducing these deadlock occurrences. For one of the benchmark circuits, the use of the proposed techniques eliminated *all* deadlocks and increased the average parallelism from 40 to 160. We believe that the use of such domain knowledge will make the Chandy-Misra algorithm significantly more effective than it would be in its generic form.

1 Introduction

Logic simulation is a very common and effective technique for verifying the behavior of digital designs before they are physically built. A thorough verification can reduce the number of expensive prototypes that are constructed and save vast amounts of debugging time. However, logic simulation is extremely time consuming for large designs where verification is needed the most. The result is that for large digital systems only partial simulation is done, and even then the CPU time required may be days or weeks. The use of parallel computers to run these logic simulations offers one promising solution to the problem.

Traditionally, the two commonly used parallel simulation algorithms for digital logic have been (i) compiled-mode simulations and (ii) centralized time event-driven simulations. In compiled-mode simulations, each logic element in the circuit is evaluated on each clock tick. The main advantage of this algorithm is its simplicity, the main disadvantage being that the processors do a lot of avoidable work, since typically only a small fraction of logic elements change state on any clock tick. The algorithm's simplicity makes it suitable for direct implementation in hardware [3,6], but such implementations make it difficult to incorporate user-defined models or represent the circuit elements at different levels of abstraction. In the second approach of centralized time event-driven algorithms, only those logic elements whose inputs have changed are evaluated on a clock tick. This avoids the redundant work done in the previous algorithm, however the notion of the global clock and synchronized advance of time for all elements in the circuit limits the amount of concurrency [2,14,17]. These centralized time approachs work efficiently on multiprocessors with 10 nodes or so [12,13,16], but for larger machines we need alternative approaches that move away from this centralized advance of the simulation clock.

The approach generating the most interest recently is the Bryant/Chandy-Misra distributed time discrete-event simulation algorithm [1,4,5,8,10,11,15]. It allows each logic element to have a local clock, and the elements communicate with each other using time-stamped messages. In this paper, we explore the suitability of the Chandy-Misra algorithm for parallel digital logic simulation. We use four realistic circuits as benchmarks for our analysis. In fact, one of the circuits is the vector-unit controller for the Titan supercomputer from Ardent. Our results show that

Submitted to DAC 1989.

the basic unoptimized Chandy-Misra algorithm results in an average *concurrency*[1] of 50 for the four circuits while being just as efficient as the event-driven algorithm. For two of the benchmark circuits, which were also studied in an earlier paper [14], the *unoptimized* Chandy-Misra algorithm extracted 40% and 107% more parallelism than the centralized time event-driven simulation algorithm.

The 50-fold average concurrency observed in the four benchmark circuits, however, is still too low. Once all the overheads are taken into account, the 50-fold concurrency may not result in much more than 10-20 fold speed-up. One major factor limiting concurrency is the large number of global synchronization points — "deadlocks" in the Chandy-Misra terminology — that occur during execution. We believe that understanding the nature of the deadlocks, why they occur and how their number can be reduced, is the key to getting increased concurrency from the Chandy-Misra algorithm. To this end, the paper presents a classification of the types of deadlocks that occur during digital logic simulation. Four different types are identified and described both intuitively in terms of circuit structure and formally with equations. Using domain specific knowledge, we then propose methods for reducing these deadlock occurrences. For one benchmark circuit, we show how using information about logic gates can eliminate all of the deadlocks. We believe that the use of such domain knowledge will make the Chandy-Misra algorithm significantly more effective than it would be in its generic form.

The organization of the rest of the paper is as follows. The next section describes the basic Chandy-Misra algorithm and some notation used in the paper. Next we describe the four benchmark circuits that were simulated to get the measurements. Section 4 presents measurements of the parallelism extracted by the algorithm and Section 5 presents the classification of the deadlocks and ways for resolving them. Finally, Section 6 presents a summary of the results and discusses directions for future research.

2 Background and Notation

2.1 Basic Chandy-Misra Algorithm, Deadlocks, and NULL Messages

We begin with a brief description of the basic Chandy-Misra algorithm [5] as applied to the domain of digital logic simulation. The simulated circuit consists of several circuit elements (transistors, gates, latches, etc) called *physical processes* (*PP*). One or more of these *PP*s can be combined into a *logical process* (*LP*), and it is with these *LP*s that the simulator works.[2] Each different type of *LP* has a corresponding section of code that simulates the underlying physical processes (note that the mapping between *PP*s and *LP*s is often trivial in gate-level circuits, with each gate represented as a simulation primitive). Each of these *LP*s has associated with it a *local* time that indicates how far the element has advanced in the simulation. Different *LP*s in the circuit can have different local times associated with them, and thus the name distributed time simulation algorithm. Each *LP* receives time-stamped event messages on its inputs and consumes the messages whenever all of the inputs are ready. As a result of consuming the messages, the logic element advances its local time and possibly sends out one or more time-stamped event messages on its outputs.

As an example, consider a two-input AND-gate with local-time 10, an event waiting on input-1 at time 20 (thus the value of input-1 is known between times 10 and 20), and no events pending on input-2. In this state, the AND-gate process is suspended and it waits for an event message on input-2. Now suppose that it gets an event on input-2 with a time-stamp of 15. The AND-gate now becomes active, consumes the event on input-2, advances its local time to 15, and possibly sends an output message with time stamp 15 plus AND-gate delay.

We now introduce the concepts of *deadlocks*. In the basic Chandy-Misra algorithm, even when input events are consumed and the local time of an *LP* is advanced, no messages are sent on an output line unless the value of that output changes. This optimization is similar to that used in normal sequential event-driven simulators where only elements whose inputs have changed are evaluated and it makes the basic Chandy-Misra algorithm just as efficient. However, this optimization also causes *deadlocks* in the Chandy-Misra algorithm. In a deadlock situation, no element can advance its local time, because each element has at least one input with no pending events. We reemphasize that this deadlock has nothing to do with a deadlock in the physical circuit, but it is purely a result of the optimization

[1] Note that, by concurrency we refer to the number of logic elements that could be evaluated in parallel if there were infinite processors.

[2] In this paper the terms *LP* and element are used interchangeably.

discussed above. The deadlock is resolved by scanning all the unprocessed events in the system, finding the minimum time-stamp associated with these events, and updating the input-time of all inputs with no events to this time (note that this deadlock resolution can also be done in parallel). Consequently, the basic Chandy-Misra algorithm cycles between two phases: the *compute* phase when elements are advancing their local time, and the *deadlock resolution* phase when all elements are stuck.

One way to totally bypass the deadlock problem is to not use the optimization discussed above. Thus elements would send output messages whenever input events are consumed and the local time of an element is advanced. This would be done even if the value on the output does not change. Such messages are called NULL messages in the Chandy-Misra terminology, as they carry only time information and no value information. Unfortunately, always sending NULL messages makes the Chandy-Misra algorithm so inefficient that it is not a good alternative to avoiding deadlocks. However, in this paper we show how *selective* use of NULL messages can significantly reduce the number of deadlocks that need to be processed.

Regarding parallel implementation of the Chandy-Misra algorithm, since each element is able to advance its local time *independently* of other elements, all elements can potentially execute concurrently. However, only when all inputs to an element become ready (have a pending event), is the element marked as available for execution, and placed on a distributed work queue. The processors take these elements off the distributed queue, execute them, update their outputs, and possibly activate other elements connected to the outputs. This happens until a deadlock is reached, when the deadlock resolution procedure is invoked.

2.2 Notation

As pointed out in the introduction, understanding the nature of deadlocks is key to increasing the parallel simulation performance. To help describe and understand the deadlocks, we now introduce some formal notation. Recall that each logical process has input and output event queues with time-stamped messages associated with it. For a particular LP_i, we have:

$\mathbf{E_{ij}}$ - the time of the earliest *unprocessed* event on input j of LP_i.

$\mathbf{E_i^{min}}$ - the minimum time of all the current input events of LP_i (short for $min_j E_{ij}$).

$\mathbf{V_i}$ - the maximum simulation time LP_i has progressed to.

$\mathbf{V_{ij}}$ - the simulation time the jth input of LP_i is valid until.

$\mathbf{D_{ij}}$ - the propagation delay from any change in an input value to a change in the jth output of LP_i.

$\mathbf{V_{ij}^O}$ - the simulation time the jth output LP_i is valid until (usually $V_{ij}^O = V_i + D_{ij}$).

$\mathbf{O_{ij}}$ - the *node* connected to the jth output of LP_i.

$\mathbf{I_{ij}}$ - the *node* connected to the jth input of LP_i.

$\mathbf{C_{ij}}$ - Directed circuit connectivity: $\begin{cases} \text{True} & \text{if there is a link from } LP_i \text{ to } LP_j \\ \text{False} & \text{otherwise} \end{cases}$

In addition to the variables above, most circuits have some notion of a system clock and an associated cycle time, so let this cycle time be denoted as $\mathbf{T_{cycle}}$.

3 Benchmark Circuits

In this section, we first provide a brief description of the benchmark circuits used in our study and then some general statistics characterizing these circuits. The four circuits that we use are:

1. **Ardent-1:** This circuit is that of the vector control unit (VCU) for the Ardent Titan graphics supercomputer[7]. The VCU is implemented in a 1.5μ CMOS gate array technology and it provides the interface among the integer processing unit, the register file, and the memory. It also allows multiple scalar instructions to be executed concurrently by scoreboarding. It consists of approximately 45,000 two-input gates.

2. **H-FRISC:** A small RISC generated by the HERCULES [9] high-level synthesis system from the 1988 High Level Synthesis Workshop. The RISC instruction set is stack based and fairly simple. This circuit consists of approximately 11,000 two-input gates.

3. **Multiplier:** This circuit represents the inner core of a custom 3μ CMOS combinational 16x16 bit integer multiplier. Multiplies are pipelined and have a latency time of 70ns. The approximate complexity is 7,000 two-input gates.

4. **8080:** This circuit corresponds to a TTL board design that implements the 8080 instruction set. The design is pipelined, runs 8080 code at a speed of 3-5 MIPS, and provides an interface that is "pin-for-pin" compatible with the 8080. The approximate complexity is 3,000 two-input gates.

We note that the benchmark circuits cover a wide range of design styles and complexity — we have a large mixed-level synchronous gate array; a medium gate-level synthesized circuit; a medium gate-level combinational chip; and a small synchronous board-level design. The fact that we have both synchronous pipelined circuits and totally combinational circuits is also important, because they exhibit very different deadlock behavior during simulation.

We now present some general statistics for these benchmark circuits in Table 1. The statistics consist of:

- Element count: The number of primitive elements (*LPs*) in the circuit. One expects the amount of concurrency in the circuit to be positively correlated with this number (it is indeed so, as can be seen in Table 2).

- Element complexity: This is defined as the number of equivalent two input gates per primitive element. The number of primitive elements multiplied by the element complexity gives a more uniform measure for the circuit complexity. The element complexity also gives an indication of the compute time required to evaluate a primitive element, and thus specifies the grain of computation.

- Element fan-in/fan-out: The average number of inputs/outputs of an element. These numbers are also correlated to the element complexity. If the average number of inputs is high, one would expect a higher probability of deadlock as there are more ways in which one of the inputs may have no event.

- Percent logic and synchronous elements: The percentage of elements that are purely combinational logic and the percentage that have internal state. Pipelined designs like the Ardent and 8080, tend to have a higher percentage of synchronous elements.

- Net count: The number of wires in the circuit.

- Net fan-out: The average number of elements a wire is attached to. The Ardent and 8080 circuits have some global buses that affect many components. This fact is reflected in their high net fan-out numbers.

- Representation: The level of representation of the simulation primitives. A circuit made up of only logic gates and one-bit registers is at the gate-level while a design made up of TTL-like components is at the RTL-level.

Another important performance related aspect that we can infer from these numbers is the relative cost of resolving a deadlock. This cost of resolving a deadlock depends on the execution time of the models (related to element complexity) and the number of elements that must be checked and possibly activated. Thus we would expect deadlock resolution to be fairly cheap for the 8080 design with 281 elements, since there are so few elements to be checked and because each evaluation of an RTL element is much longer than a trivial logic operation. However, we would expect the relative cost of resolving a deadlock in the larger gate-level circuits (for example, H-FRISC) to be high due to the large number of components and the low execution time of the models.

Table 1: Basic Circuit Statistics

Statistic	Ardent-1	H-FRISC	Mult-16	8080
Element Count	13.349	8.076	4.990	281
Element Complexity	3.4	1.40	1.42	12
Element Fan-in	2.72	2.14	2.14	5.78
Element Fan-out	1.2	1.0	1.0	2.63
% Logic Elements	88.8	97.2	100	83.3
% Synchronous Elements	11.2	2.8	0.0	16.7
Net Count	13.873	8.093	5.077	748
Net Fan-out	2.66	2.14	2.14	5.48
Representation	gate/RTL	gate	gate	RTL
Basic Unit of Delay	0.5ns	unit	1ns	1ns

4 Parallelism Measurements

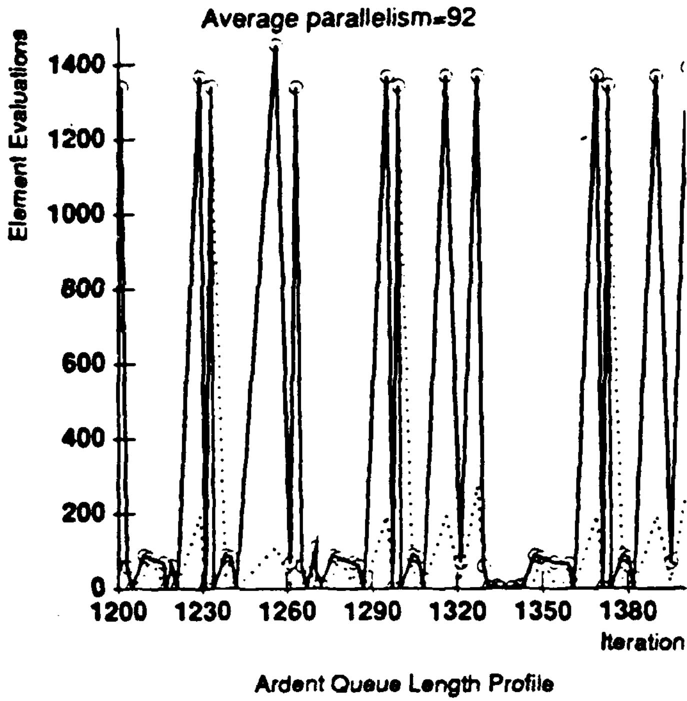
In this section. we discuss how parallelism is exploited by the Chandy-Misra algorithm and present data regarding the amount of concurrency available in the four benchmark circuits. We also present data regarding the granularity of computation. the number of deadlocks per clock cycle, and the amount of time spent in deadlock resolution. These numbers were gathered from our parallel implementation of the Chandy-Misra algorithm running on an Encore Multimax. a shared-memory multiprocessor with sixteen NS32032 processors. each processor delivering approximately 0.75 MIPS.

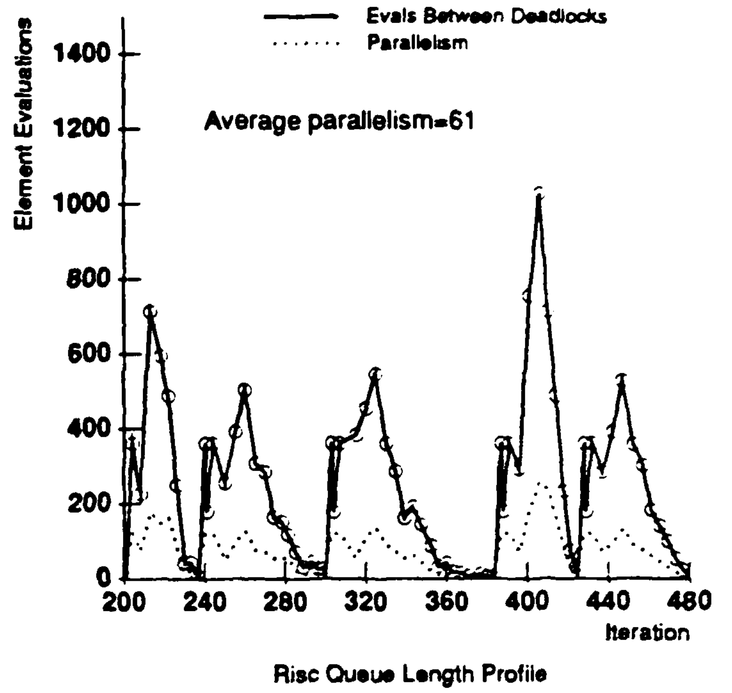
Since we are interested in the parallel implementations of the Chandy-Misra algorithm. the first question that arises is how much speed-up can be obtained if there were arbitrarily many processors, and if there were no synchronization or scheduling overheads. We call this measure the *concurrency* or intrinsic parallelism of the circuits under Chandy-Misra algorithm. For our concurrency data, we further assume that all element evaluations take exactly one unit of time. Thus, the simulation proceeds as follows. After a deadlock and the ensuing deadlock resolution phase, all elements that are activated (i.e., have at least one event on each of their inputs) are processed. This happens in exactly one unit-cost cycle as we assume arbitrarily many processors. The number of elements that are evaluated constitutes the concurrency for this *iteration*. The evaluation of the elements, of course, results in the activation of a whole new set of elements, and these are evaluated in one cycle in the next iteration. The computation proceeds on this way until a deadlock is reached, and we start all over again.

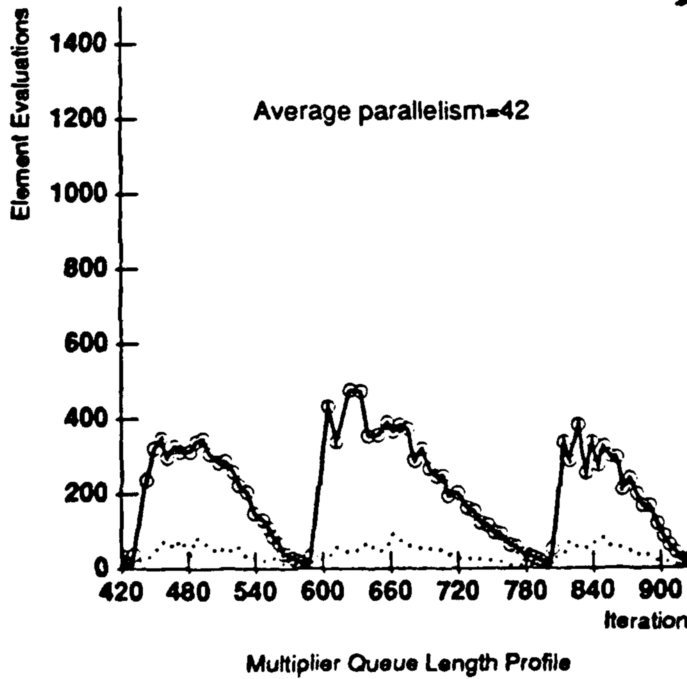
Figure 1 shows the concurrency data (shown using the dashed line) and event profiles (shown using the solid line) for the four benchmark circuits. The event profiles show a plot of the total number of logic elements evaluated between deadlocks. The profiles are generated over three to five simulated clock cycles in the middle of the simulation. We would like to reemphasize that the profiles in Figure 1 are not algorithm independent, but are specific to the basic Chandy-Misra algorithm. In fact, our research suggests enhancements to the basic Chandy-Misra algorithm, so that much more concurrency may be observed.

The profiles clearly show cyclical patterns with the highest peaks corresponding to the system clock(s) of the simulated circuits, and the portions between the peaks corresponding to the events propagating through the combinational logic between the sets of registers. The Ardent profile shows that the circuit quickly stabilizes after the clock with only a few deadlocks while the multiplier, with many levels of combinational logic, takes quite a while to stabilize with many deadlocks. This close correspondence between the event profiles and the circuit being simulated shows the importance of exploiting domain specific information: any circuit characteristic we change or exploit will be directly reflected in the event profiles. Understanding how these changes affect the profiles and being able to predict them is important in obtaining better performance. A summary of the concurrency information is also presented in Table 2. The top line of the table shows the concurrency as averaged over all iterations in the simulation.

In addition to knowing how many concurrent element evaluations or tasks that are available, we also need to know







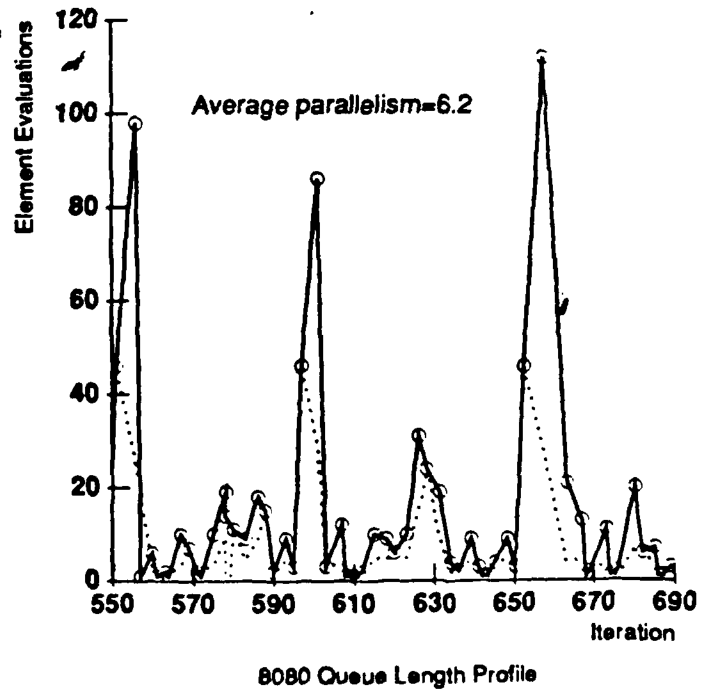


Figure 1: Event Profiles

Table 2: Simulation Statistics

Statistic	Ardent-1	H-FRISC	Mult-16	8080
Unit-cost Parallelism	92	67	42	6.2
Granularity (ms)	0.74	0.66	0.75	2.61
Deadlock Ratio	308	245	248	15
Cycle Ratio	1.644	1.982	6.712	132
Deadlocks Per Cycle	5.3	8.1	27.1	8.9
Avg Deadlock Resolution Time (ms)	520	230	206	11
% Time in Deadlock Resolution	58	46	41	19

the task granularity and how often deadlocks (global processor synchronizations) occur. The granularity or basic task size for our application (a model evaluation) includes checking the input channel times, executing the model code, calculating the least next event and possibly activating the elements in its fan-out. The numbers discussing task granularity and frequency of deadlocks are summarized in Table 2. The table also presents the following ratios that help characterize the performance of the Chandy-Misra algorithm:

- **Deadlock ratio (DR):** Number of element evaluations divided by the number of deadlocks.
- **Cycle ratio (CR):** Number of element evaluations divided by the number of simulated clock cycles.
- **Deadlocks per cycle:** Number of deadlocks divided by the number of simulated clock cycles.

Since increased parallelism was the main motivation for using the Chandy-Misra algorithm, we now compare the concurrency it obtains to that obtained using a traditional event-based algorithm. For our comparison, we use the concurrency data presented for the 8080 and multiplier circuits in a parallel event-driven environment in [13,14]. These papers showed that the available concurrency was about 3 for the 8080 and 30 for the multiplier. From Table 2, the corresponding numbers for the Chandy-Misra algorithm are 6.2 for the 8080 and 42 for the multiplier. The fact that the concurrency increases only by a factor of 1.5-2 is somewhat disappointing, since Chandy-Misra algorithm is more complex to implement. However, we believe that using the techniques proposed in the next section, the Chandy-Misra algorithm can be suitably enhanced to show much higher concurrency.

The last two lines of Table 2 give data about the average time taken by each call to deadlock resolution and the total fraction of time spent in deadlock resolution. The cost of resolving a deadlock for the three larger circuits is indeed high, especially when compared to the cost of evaluating a logic element (see the granularity line). For example, in the time it takes to resolve a deadlock in Ardent, 700 logic element activations could have been processed. In H-FRISC, 350 elements could have been evaluated, and in the multiplier, 275 elements could have been evaluated. In our research, we are also exploring techniques to reduce the deadlock resolution time significantly by caching information from previous simulation runs of same circuit, but results are not available yet.

5 Characterizing Deadlocks

Even though there is reasonable parallelism available in the execution phase of the Chandy-Misra algorithm, deadlock resolution is so expensive in the larger circuits that it consumes 40-60% of the total execution time. Clearly we have to reduce this percentage in order to get good overall parallel performance. The first step towards this reduction is understanding why deadlocks occur and how they can be avoided. The types of deadlock that occur in logic simulation are characterized in this section and this characterization gives us insight into what aspects of logic simulation can be effectively exploited to achieve good overall performance.

In the logic simulations that were studied, the elements that became deadlocked can be put into two categories: (i) those deadlocked due to some aspect of the circuit structure (e.g topology, nature of registers, feed-back) and (ii)

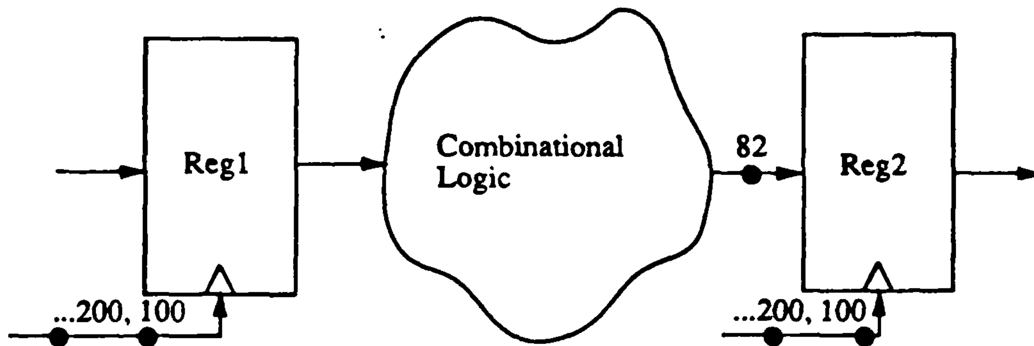


Figure 2: Deadlock Caused by a Clocked Register

those deadlocked due to low activity levels (e.g. typically only 0.1% of elements need to be evaluated on each time step in event-driven simulators[14]). In the following subsections, descriptions and examples of each of the types of deadlock are given, along with measurements that show how much each type contributes to the whole.

5.1 Registers and Generator Nodes

In a typical circuit, enough time is allowed for the changes in the output of one set of registers to propagate all the way to the next set of registers in the datapath and stabilize before the registers are clocked again. For example, in Figure 2, the critical path of the combinational part of the circuit is 82ns, and the clock node changes every 100ns to allow everything to stabilize. *Reg1* is clocked at the start of the simulation, and the events propagate through the combinational logic, generating an event at time 82. This event at time 82 is consumed by *Reg2* since the clock node is defined for all time in this example. However, the next event at time 100 is *not* consumed since the input to the latch is only defined up to time 82, not 100. This causes *Reg2* to block and the deadlock resolution phase is entered. This is a large source of deadlocks since most circuits have many registers, latches and generator nodes (e.g. clock(s), reset, inputs, etc.),

In Table 3 we see that for the Ardent, register-clock deadlocks account for 92% of all the elements activated in the deadlock resolution phase even though registers only make up 11% of the elements. This is mainly due to the pipelined nature of the Ardent design where there is only a small amount of combinational logic between register stages. In the case of the RISC design, there are more combinational logic between the registers than the Ardent and more logic gates connected to the input stimulus generators. Thus register-clock and generator deadlocks both cause around 20% of the deadlock activations for a total of 40%. In the multiplier design, there are many levels of logic between the inputs and outputs and does not have any registers. Thus there can not be any register-clock deadlocks and very few generator deadlocks. The 8080 design, like the Ardent, is pipelines and hence register-clock deadlocks are the main source of deadlock Here 55% of the activations are caused by register-clock deadlocks while only 17% of the elements are registers.

5.1.1 Detection

In order to measure how much any particular deadlock type affects the overall simulation, there must be some way of concretely identifying that type. A *register-clock deadlock* is said to occur whenever a clocked element LP_i that is activated during deadlock resolution has the earliest unprocessed event on its clock input. A *generator deadlock* is said to occur whenever the earliest unprocessed event was received directly from a generator element. In terms of the notation introduced earlier, this can be expressed as when $E_i^{min} \bmod T_{cycle} = 0$

Table 3: Register-Clock and Generator Deadlocks

Circuit	Total Deadlock Activations	Register-clock Activations	% of Total	Generator Activations	% of Total
Ardent-1	316.0k	290.0k	92	583	0.2
H-FRISC	45.6k	8.9k	20	8,800	19.0
Mult-16	27.2k	0.0k	0	40	0.1
8080	8.3k	4.6k	55	53	0.6

5.1.2 Proposed Solutions

Taking advantage of behavior: In general, an input event may arrive at any time in an element's future causing it to change its output. Thus, an element can only be sure of its outputs up to the minimum time its inputs are valid plus the output delay ($V_i + D_{ij}$). In the case of registers and latches, however, we know that the output will not change until the next event occurs on the clock input regardless of the other inputs. This knowledge of input *sensitization* is easy to use and potentially very effective since the outputs can be advanced up to the next clock cycle. In registers and latches with asynchronous inputs (like set, clear, etc.), those inputs must be taken into account as well as the clock node when determining the valid time of an output.

Fan-out Globbing: This technique reduces the overhead and the time needed to perform deadlock resolution. Recall that a particular LP is composed of many PPs. These PPs can be combined in different ways to form larger units. Combining many registers that share the same clock node will reduce the overhead of activating each register separately. Typically hundreds of one-bit registers and gates are connected to the clock node(s) and *often* times during deadlock resolution, the minimum event is on the clock node (as in the example above). If we combine these registers and gates in groups of n, we call this grouping *fan-out globbing with a clumping factor of* n since we are combining the fan-out elements of the clock. This reduces the overhead of inserting and deleting the elements in the evaluation queue. However, since it combines elements, it also reduces the parallelism available. We are currently looking into just how much reduction in overhead and parallelism this causes.

5.2 Multiple Input Paths with Different Delays

Whenever there are multiple paths with different delays from a node to an element, there is a chance of that element deadlocking. An example of this is the MUX shown in Figure 3. There are two paths from the Select line to the OR-gate at the output. If the Data and ScanData lines are valid, an event on the Select node could propagate through the two paths and generate events at times 11 and 12. The event at time 12 will not be consumed by the OR-gate since its other input is only defined up to time 11 causing the OR-gate to deadlock. Thus, multiple paths from a node to an element can result in an unconsumed event on the path with the larger delay.

5.2.1 Detection

Let LP_i be the deadlocked element and j be the index of the input with the unprocessed event (i.e. j such that $E_{ij} = E_i^{min}$). Then if there are two different paths from some element, LP_k, to the deadlocked element, LP_i, with the longer path ending at input j, then a *multiple path* deadlock has occurred.

5.2.2 Proposed Solutions

Since this type of deadlock is due to the local topology of the circuit, there is no easy way of avoiding it. However, there are a couple of options.

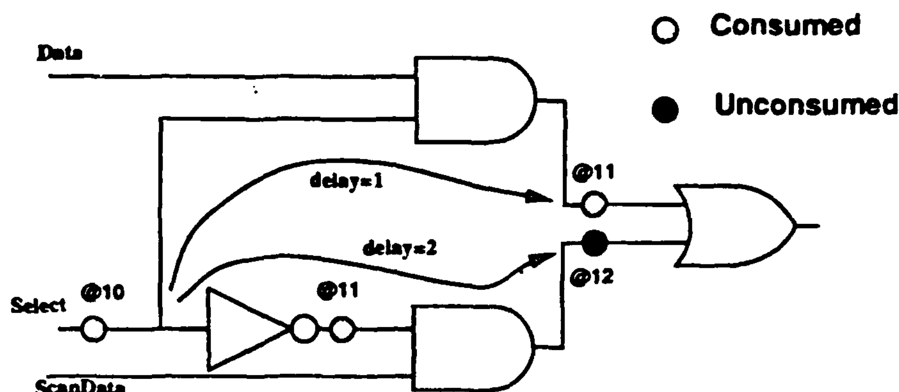


Figure 3: Deadlock due to Multiple Paths of Different Delays

Demand-driven: The elements that are affected by multiple paths could be marked either while compiling the netlist or from previous simulation runs. When these elements are executed, a demand driven technique could be used. With a demand-driven technique, whenever an element can not consume an input event, requests are made to its fan-in elements (the ones driving its input pins) asking "Can I proceed to this time?". These requests propagate backwards until a yes or no answer can be ensured. Propagating these requests can be expensive especially if there are long feedback chains in the circuit. Thus we must be very selective in the elements we choose to use this technique with.

Structure globbing: If there are not too many elements involved in the multiple paths, we may be able to *hide* the multiple paths by globbing those elements into one larger *LP*. However, the composite behavior of the gates must be generated and the detailed timing information must be preserved. Preserving the exact timing information is non-trivial. In essence a state variable must be made for each of the internal nodes and the element may have to schedule itself to make the outputs change at the correct times. This self-scheduling may cause the element to deadlock because, by requesting itself to be evaluated at some time, it must wait until the inputs are valid up to that time just as before. If the detailed timing information does not need to be preserved, the composite behavior is easy to generate (compiled-code simulation techniques can be used on the small portion of the circuit that is being globbed together) and this deadlock type will be avoided.

Taking advantage of behavior: If we know the behavior of an element, it may be possible to advance that element even though some of its inputs are not known. For example in Figure 3, if the event at time 11 going into the OR-gate has a value of 1, the output is known to be 1 regardless of the value of the other input and the OR-gate need not deadlock. In a gate-level simulation, the behavior of most of the elements is very simple and can be readily exploited.

5.3 Order Of Node Updates

The *activation criteria* for the basic Chandy-Misra algorithm is: activate an element only when an event is received on one of its inputs. Sometimes this activation criteria can cause a consumable input event to be stranded due to the order in which the node updates are performed. This stranded event will cause the element to deadlock. In Figure 4, element e1 consumes the event at time 10, produces an event at time 11, and activates element e3. If e3 is now executed, e3 will not be able to consume the event at time 11 because the input from e2 will not be valid at time 11. If an event at time 10 now arrives at e2 and e2 is evaluated, it will update the valid-time of the input to e3 but it will not activate e3 because no *event* was generated. If e3 had waited for e2 to be evaluated, the inputs to e3 would have both been valid at time 12 and the event at time 11 could have been consumed.

In Table 4, we see that the order of node updates type of deadlock is uncommon in the Ardent simulation which is

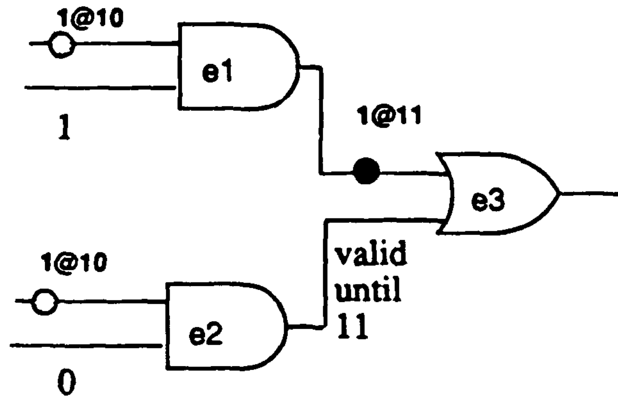


Figure 4: Deadlock Caused by Order Of Node Updates

Table 4: Deadlock Activations Caused by the Order of Node Updates

Circuit	Deadlock Activations	Order of Node Updates	% of Total
Ardent-1	316.0k	1.4k	0.4
H-FRISC	45.6k	1.0k	2.2
Mult-16	27.2k	1.7k	6.2
8080	8.3k	0.2k	2.2

dominated by the register-clock deadlocks. The order of node updates is, however, important in the combinational multiplier with its many levels of logic

5.3.1 Detection

Suppose LP_i is activated during deadlock resolution because it was waiting to consume an event at time t. If *all* of the input nodes are found to have advanced up to time t — that is if the element can safely consume the event without *any* input times being updated, an *order of node updates* deadlock has occurred. In the notation introduced earlier. this happens if $\min_j V_{ij} \geq E_i^{min}$

5.3.2 Proposed Solutions

New activation criteria: The problem is that the activation criteria does not activate an element when the valid times of its input node are updated. The problem can be eliminated if an element checks its fan-out elements when it updates the time of its output nodes. Any of those fan-out elements that have a real event at a time less than or equal to the new valid-time, should be activated. In the example, e2 would activate e3 when it updated the valid-time of its output to 11 since e3 has a real event at time 11. Note that this only works for the case where the updated node is directly connected to the element with the unconsumed event. If there are any intermediate

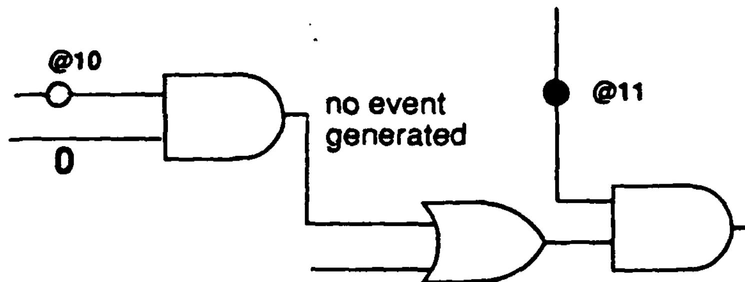


Figure 5: Deadlock Caused by Unevaluated Path

elements the deadlock is considered to be caused by an unevaluated path which is explained in the next subsection. If e3 had a third input, it still may not be able to consume the event at time 11 even after e2 is evaluated. This extra activation creates needless work and the effectiveness of this solution depends on the relative cost of performing a deadlock resolution on the particular circuit being simulated.

We can describe this new activation criteria formally by doing the following after each LP_i is evaluated:

```
For each output j
    For each LP_k connected to output j
        If V_ij^O >= E_k^min
          then Activate LP_k
```

Rank ordering: The *rank* of an element is the maximum number of levels of logic between the element and any registers. It can be computed by assigning all registers and generator elements a rank of 0 and then iterating through the combinational elements assigning them a rank of one plus the maximum rank of its input elements. If the elements in the evaluation queue are ordered by their rank, the node updates will proceed in a more ordered fashion (i.e. elements farther away from the registers and external inputs that affect it will be evaluated later possibly letting their inputs become defined). In the example, e2 would be inserted before e3 since the inputs to e3 depend on the outputs of e2.

Since the rank information is easy to compute while compiling the netlist, the *run-time* cost is very little. Also, this technique does not generate any extra activations so the overall cost is cheap.

5.4 Unevaluated Path

The elements in the fan-out of a wire are activated only when a real event is produced on that wire. Thus, if element LP_i consumes an event but does not produce a new event (i.e. the activation does not result in a change in value of output signals), all paths from LP_i to the other elements will not be evaluated or updated. Figure 5 shows the case where one event is consumed and, since no new event is produced, the OR-gate is not activated and the second AND-gate can not consume the event at time 11 since the valid-time of the input from the OR-gate was not updated.

In Table 5 we see that unevaluated paths are very important in three of the four circuits. This is especially true for the RISC and multiplier designs which consist of many levels of combinational elements. For the RISC, the number of deadlocks caused by unevaluated paths is around 60% and that for the multiplier around 90%. In contrast, unevaluated paths are relatively unimportant in simulations of the pipelined Ardent design.

Table 5: Deadlock Activations Caused by Unevaluated Paths

Circuit	Deadlock Activations	One Level NULL	% of Total	Two Level NULL	% of Total	Combined %
Ardent-1	316.0k	3.0k	1.0	21.0k	6.6	8
H-FRISC	45.6k	4.3k	9.4	22.6k	49.6	59
Mult-16	27.2k	1.5k	5.5	23.8k	87.5	93
8080	8.3k	0.5k	5.7	2.9k	34.9	41

5.4.1 Detection

If NULL messages are *always* sent, the simulation will never deadlock (see Section 2.1). Unfortunately, this is highly inefficient since typical activity levels in event-driven simulators are around 0.1% in each time step. To find out how many deadlocks we could avoid by only *selectively* sending NULL messages, we did the following. We measured how many deadlock activations would have been avoided if every deadlocked element had received NULL messages from its immediate fan-in — corresponding to what we call "one level" of NULL messages — and how many activations would have been avoided by two levels of NULL messages.

To define this more concretely, let the *distance* between LP_i and LP_j be the minimum number of elements, $(e_1, e_2, \ldots e_k)$ such that $C_{ie_1}, C_{e_1e_2}, \ldots C_{e_kj}$ are all true. Let this distance be denoted by δ_{ij} and the minimum delay between LP_i and LP_j by τ_{ij}. Using these definitions we get: LP_i was deadlocked by an unevaluated path of n levels.

If For each input j where $V_{ij} < E_i^{min}$
  and each LP_k where $\delta_{ki}=n$ and the path ends at input j
    $(V_k + \tau_{ki}) \geq E_i^{min}$ holds

5.4.2 Proposed Solutions

Caching: Since the activity levels are so low, we need to be very selective about which elements should send NULL messages. The proposed selection process follows the concept of *caching*. By caching information from previous runs, we can identify the elements that repeatedly deadlock due to an unevaluated path as the simulation progresses. When these elements get activated, they will send out NULL messages whenever their outputs times advance. In order to be effective, the caching algorithm must be quick and effective.

Taking advantage of behavior: If we know the behavior of an element, it may be possible to advance that element even though some of its inputs are not known. For example, if the event at time 11 going into the AND-gate of Figure 5 has a value of 0, the output is known to be 0 regardless of the value of the other input. In a gate-level simulation, the behavior of most of the elements is very simple and can be readily exploited. As it turns out, this technique works very well for the combinational multiplier circuit. It eliminates *all* deadlocks and increases the parallelism from 40 to 160.

5.5 Summary of the Contributions from each Deadlock Type

A summary of the composition of an average deadlock for the benchmark circuits is given in Table 6. In all but the Ardent circuit, the main deadlock type is the two-level NULL caused by unevaluated paths which are, in turn, caused by the very low activity levels in digital logic simulations. The Ardent and 8080 deadlocks are made up predominantly of register-clock deadlocks. They account for 92% and 55% of the deadlock activations even though synchronous elements comprise only 11 to 17% of the total elements. This is mainly due to the heavily pipelined

Table 6: Deadlock Activations Classified by Type

Circuit	Total Deadlock Activations	Register-clock Activations	Generator Activations	Order of Node Updates	One Level NULL	Two Level NULL
Ardent-1	316.0k	290.0k	583	1.4k	3.0k	21.0k
RISC	45.6k	8.9k	8.800	1.0k	4.3k	22.0k
Mult	27.2k	0.0k	40	1.7k	1.5k	23.8k
8080	8.3k	4.6k	53	0.2k	0.5k	2.9k

nature of the two circuits — lots of latches with only a few levels of logic in between. Thus most of the deadlocks occur when the registers and latches are waiting for their inputs to become valid.

The main contributors to deadlock in the RISC circuit (after the two-level NULL deadlocks), are generator and register-clock deadlocks. This is due to the consistent control style used by the synthesis system. The system clocks are generated externally and first pass through a level of logic that controls which parts of the design are active. These qualified clocks are then distributed to their corresponding circuit sections — the result being that most registers are waiting on their inputs and the elements connected to the generator nodes are waiting on their other inputs.

The multiplier design is highly interconnected with many levels of logic. Almost all of the deadlock activations are caused by the unevaluated paths in the circuit as shown by the two-level NULL column. This is caused by a few paths that are active all the way from the inputs to the outputs while most of the paths do not have any activity at all after the first couple of levels.

6 Conclusions

In characterizing the parallelism in distributed-time simulations of real circuits, we have shown that the Chandy-Misra algorithm extracts an average parallelism of 50 for the four benchmark circuits used. While this is 1.5-2 times better than traditional parallel event-driven algorithms, it is still too low to be used effectively in large parallel processing systems. Since deadlocks are the major factor limiting parallelism and the overall performance, the paper focused on understanding the nature of deadlocks. We classify the deadlocks that occur in logic simulation into four types: register clocks and generator nodes, multiple paths, unevaluated paths and the order of node updates. These four types are able to cover almost all of the deadlocks that occur. Concentrating on each type, we presented specific solutions to avoid or resolve the deadlocks caused by that type. Preliminary results show that we can eliminate all of the deadlocks in the multiplier simulation raising the parallelism from 40 to 160. These solutions exploit several different aspects of circuit behavior and we feel that it is with this domain specific knowledge that significantly better parallel performance can be achieved.

7 Acknowledgements

The authors are supported by DARPA contract N00014-87-K-0828. In addition, Anoop Gupta is supported by a DEC faculty award, and Larry Soule is supported by DEC, and by a National Science Foundation Graduate Fellowship.

References

[1] Marc Abrams. The Object Library for Parallel Simulation (OLPS). In *Winter Simulation Conference, 1988*, December 1988.

[2] M. Bailey and L. Snyder. An Empirical Study of On-Chip Parallelism. In *25th Design Automation Conference*, pages 160–165. University of Washington, June 1988.

[3] Tom Blank. A survey of hardware accelerators used in computer-aided design. *IEEE Transactions on Design and Test*, 21–39, August 1984.

[4] R. E. Bryant. *Simulation of Packet Communication Architecture Computer Systems*. Technical Report MIT.LCS.TR-188, MIT, July 1977.

[5] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. *Comm of the ACM*, 24(11):198–206, April 1981.

[6] Monty Denneau. The Yorktown Simulation Engine. In *19th Design Automation Conference*, page 7.2. ACM/IEEE, 1982.

[7] Tom Diede, Carl Hagenmaier, Glen Miranker, Johnathan Rubinstein, and William Worley. The Titan Graphics Supercomputer Architecture. *Computer*, 21(9):13–30, September 1988.

[8] Richard Fujimoto. Lookahead in Parallel Discrete Event Simulation. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 34–41, University of Utah, 1988.

[9] David Ku and Giovanni DeMicheli. HERCULES - A System for High-Level Synthesis. In *25th Design Automation Conference*, pages 483–486, ACM/IEEE, June 1988.

[10] David Nicol. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. In *PPEALS 88*, pages 124–137, ACM, 1988.

[11] D. Reed, A. Maloney, and B. McCredie. Parallel Discrete Event Simulation: A Shared Memory Approach. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[12] J. Smith, K. Smith, and R. Smith. Faster Architectural Simulation Through Parallelism. In *24th Design Automation Conference*, pages 189–194, ACM/IEEE, June 1987.

[13] Larry Soule and Tom Blank. Parallel Logic Simulation on General Purpose Machines. In *Proceedings of the 25th Design Automation Conference*, pages 166–171, Stanford University, 1988.

[14] Larry Soule and Tom Blank. Statistics for Parallelism and Abstraction Level in Digital Simulation. In *Proceedings of the 24th Design Automation Conference*, pages 588–591, Stanford University, 1987.

[15] David Wagner, Edward Lazowska, and Brian Bershad. *Techniques for Efficient Shared-Memory Parallel Simulation*. Technical Report 88-04-05, University of Washington, Department of Computer Science, August 1988.

[16] Andrew Wilson. *Parallelization of an Event Driven Simulator on the Encore Multimax*. Technical Report ETR 86-005, Encore Computer, 1986.

[17] Franklin Wong. Statistics on Logic Simulation. In *23rd Design Automation Conference*, pages 13–19, ACM/IEEE, July 1986.

Temperature Measurement and Equilibrium Dynamics of Simulated Annealing Placements

Jonathan Rose
Computer Systems Laboratory,
The Center For Integrated Systems, Stanford University

Wolfgang Klebsch and Juergen Wolf
Siemens AG, Munich, Federal Republic of Germany

Abstract

One way to reduce the computational requirements of Simulated Annealing placement algorithms is to use a faster heuristic to replace the early phase of Simulated Annealing. Such systems need to know a starting temperature for the annealing phase that makes the best use of the structure provided by the heuristic, yet does an appropriate amount of improvement. This paper presents a method for *measuring* the temperature of an existing placement. It is based on a view of Simulated Annealing state that differs from previous work - the probability distribution of the *change* in cost function, as opposed to the *absolute* cost function. Using this view a new definition of equilibrium is given and the *equilibrium temperature* of a placement is defined. This also gives rise to an new view of the *equilibrium dynamics* of Simulated Annealing. A measure is developed that quantifies the nearness of a Simulated Annealing placement to equilibrium, and experimental evidence of its ability to detect equilibrium is given. Based on the measure a method is presented for determining the equilibrium temperature of a placement, and it is applied to placements of a real circuit produced both by a Simulated Annealing and a Min-Cut placement algorithm. For the latter an experimental relationship between the Min-Cut *cut area* and the measured temperature is demonstrated.

1 Introduction

The success of the Simulated Annealing algorithm for automatic placement [Sech85] has been hindered by its excessive computational requirements. Recent work on standard cell placement algorithms [Rose86a, Grov87, Rose88a] has suggested alleviating this by using a two-stage approach: begin with a good, reasonably successful heuristic such as the Min-Cut algorithm [Breu77,Dunl85] and then follow it with a Simulated Annealing-based approach for more fine optimization. Replacement of the early phase of Simulated Annealing with a faster but potentially worse algorithm allows a tradeoff between execution time and quality. A critical issue in this approach is to decide the starting temperature of the Simulated Annealing phase. If the temperature is too high, then some of the structure created by the first phase will be destroyed and unnecessary extra work will have be to be done in the Simulated Annealing phase. If the temperature is too low then solution quality is lost, similar to the case of a quenching cooling schedule [Whit84].

This paper presents a technique for *measuring* the temperature of a placement for use in such two-stage systems. The problem is to determine the starting temperature for a Simulated Annealing process

so that the "best" use of the original structure is made, yet an "appropriate" amount of optimization is done to improve it. To give meaning to the concept of a placement's temperature, a framework is needed in which the notions of "best" and "appropriate" are defined.

Accordingly, we present a new view of Simulated Annealing state different from those articulated in [Laar87, Aart85,Rome84, Whit84]. The principal difference is that we look at probability distributions of the *change* in cost function of a Simulated Annealing state, rather than the absolute cost function. Using this view we give a definition of equilibrium from which follows the notion of the *equilibrium temperature* of a placement. The way in which the probability distribution changes as equilibrium is reached, known as the *equilibrium dynamics* [Laar87], is demonstrated with measurements on a real circuit.

We develop a measure that quantifies the nearness of a Simulated Annealing placement to equilibrium, called the Cost Force Ratio (CFR), and give experimental evidence of the CFR's ability to detect equilibrium. Based on the CFR measure, we present a method for measuring the equilibrium temperature of a placement, and show that it works both for placements produced by a Simulated Annealing and a Min-Cut placement algorithm. For the latter we show an experimental relationship between the Min-Cut *cut area* and the measured temperature.

The determination of starting temperature for Simulated Annealing in two-stage systems has not been seriously addressed before. Both [Rose86a,Rose88a] and [Grov87] introduce the question but avoid answering it by choosing a starting temperature based simply on previous experience. A shorter version of this paper is to appear in [Rose88b].

2 Definition of Equilibrium and Temperature

In previous discussions of cooling schedules and convergence [Laar87, Aart85,Rome84, Whit84], the Simulated Annealing *state* has been represented either as the probability distribution of the absolute cost $P(C)$, or the set of transition probabilities from every state i to every other state j, T_{ij}. We suggest a different view that gives more information about equilibrium dynamics: the probability distribution of the *change* in cost function *from* the current state. $P(\Delta C)$ is the probability that a given state under a Simulated Annealing process with a particular generation function [Rome84] will generate a move with a change in cost function of ΔC. $P(\Delta C)$ varies with temperature (T) and as moves are made.

We can use this view to give a different perspective on the equilibrium of a Simulated Annealing process. Since at equilibrium the absolute cost function no longer changes, this implies that the expected value of the *change* in cost function is zero:

$$E(\Delta C) = 0 \quad (1)$$

An expression for $E(\Delta C)$ can be formed assuming that $P(\Delta C)$ is known:

$$E(\Delta C) = \int_{-\infty}^{\infty} \Delta C \;\; P(\Delta C)\, P_{Accept}(\Delta C)\, d\,\Delta C \tag{2}$$

$P_{Accept}(\Delta C)$ is the probability that the acceptance function will accept a move with cost ΔC [Rome84]. It commonly has the value 1 for $\Delta C \leq 0$ and $e^{\frac{-\Delta C}{T}}$ for $\Delta C > 0$ [Sech85]. We note here that $P(\Delta C)$ in equation (2) must be the distribution measured on a *running* Simulated Annealing process *at* the equilibrium temperature. This distribution is difficult to measure, as will be discussed further in Section 3.1.

Using this $P_{Accept}(\Delta C)$ we can split equation (2) into two parts and, and at equilibrium from equation (1) we can equate it to zero:

$$\int_{-\infty}^{0} \Delta C \;\; P(\Delta C)\, d\,\Delta C \;\; + \int_{0}^{\infty} \Delta C \;\; P(\Delta C)\, e^{\frac{-\Delta C}{T_{eq}}}\, d\,\Delta C \; = \; 0 \tag{3}$$

Thus *equilibrium* can now be defined as the state where, at a given $T = T_{eq}$, the distribution $P(\Delta C)$ satisfies equation (3). Conversely, the *equilibrium temperature* of a placement with a distribution $P(\Delta C)$ is the temperature, T_{eq}, for which equation (3) is satisfied.

We note here that $P(\Delta C)$ in equation (2) must be the distribution measured on a running Simulated Annealing process *at* the equilibrium temperature. This distribution is difficult to measure, as will be discussed further in Section 3.1.

2.1 Equilibrium Dynamics

The way in which the probability distributions change throughout the process, or the *equilibrium dynamics*, can be explained by observing how $P(\Delta C)$ changes when moving from non-equilibrium to equilibrium. Suppose a system is in equilibrium at temperature T_1, and its temperature is then lowered to T_2. Figure 1 is a plot of $P(\Delta C)$ and P_{Accept} versus ΔC for a fictitious system in equilibrium at temperature T_1. When the temperature is lowered to T_2 the only change is that the positive portion of the accept function becomes uniformly lower because $e^{\frac{-\Delta C}{T_2}} < e^{\frac{-\Delta C}{T_1}}$ for all $\Delta C > 0$.

For this system to *regain* equilibrium after the temperature change, $P(\Delta C)$ must change to again satisfy equation (3). This means that one or both of the following must happen:

1. The positive portion of $P(\Delta C)$ must either shift right (greater bad moves) or up (more bad moves), increasing the expected positive component of ΔC (E_+) or,

2. The negative portion of $P(\Delta C)$ must either shift right (smaller good moves) or down (fewer good moves), reducing the magnitude of expected negative component of ΔC (E_-).

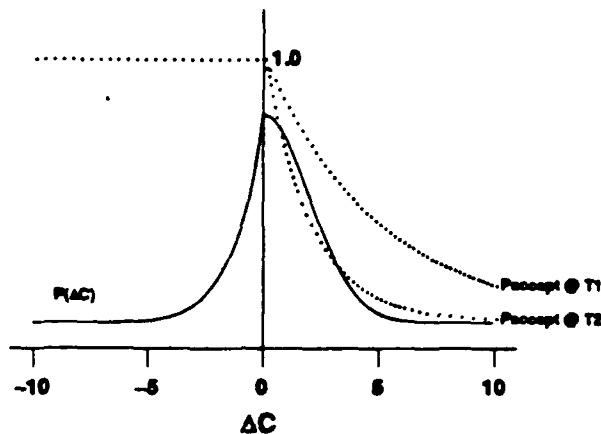


Figure 1 - *Fictitious Probability Distribution and Acceptance Function at Temperature Change*

Experimentally, both these effects are observed. Figure 2 is a plot of $P(\Delta C)$ versus ΔC for the 833 standard cell Primary1 circuit from the Preas-Roberts standard cell benchmark suite [Prea87]. It was produced by the SALTOR Simulated Annealing placement program [Rose86b,Rose88a], which is based on the ideas of the Timberwolf standard cell placement program [Sech85]. $P(\Delta C)$ is measured by generating 200,000 moves on a placement without actually accepting those moves (these are called *virtual* moves). In this way the placement is not changed, and a "point" measurement of the distribution in time is obtained. As discussed in Section 3.1, this *static* measurement is very close to the *dynamic* one, where the measurement is made on the Simulated Annealing process running in equilibrium.

Figure 2 gives $P(\Delta C)$ for three temperatures: very high (T = 5000), medium (T = 300) and low (T = 9). As the temperature decreases, the negative portion of $P(\Delta C)$ undergoes a dramatic shift to the right, and is much smaller than the positive portion of $P(\Delta C)$. This relates to the placement process in that all of the large good moves are used up, and only a few relatively small improvements are possible.

As temperature decreases, the positive portion of $P(\Delta C)$ in Figure 2 undergoes a right and upward shift. This occurs because as the placement gets better, there are more moves that will have a greater bad effect on the placement.

2.2 An Equilibrium-Nearness Measure

Using equation (3) we can invent a measure of the nearness of a given Simulated Annealing state to equilibrium. Define E_- to be the absolute value of the first term in the equation, that is

$$E_- = \left| \int_{-\infty}^{0} \Delta C \; P(\Delta C) d\Delta C \right|$$

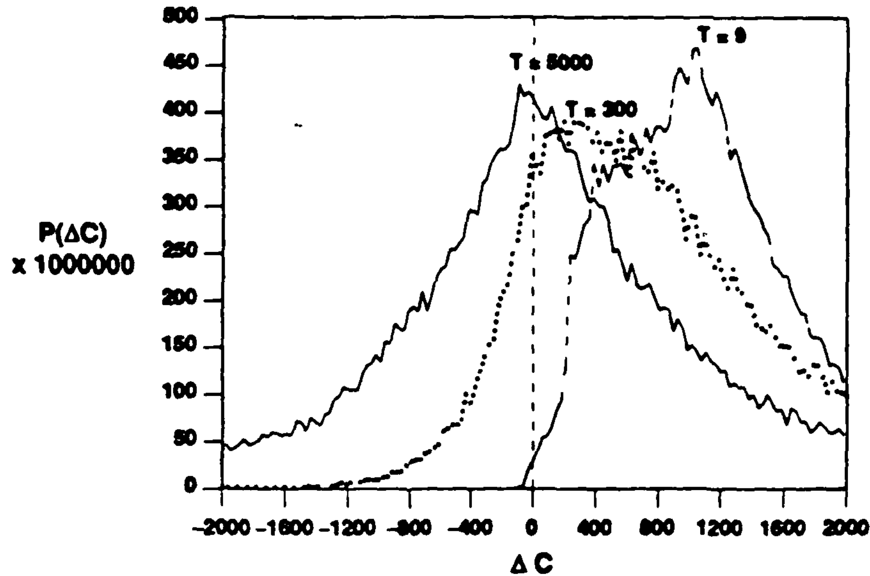


Figure 2 - *P(ΔC) versus ΔC on Primary1 for Temperatures 5000,300 and 9*

Similarly let E_+ be the second term of equation (3):

$$E_+ = \int_0^{\infty} \Delta C \; P(\Delta C) \, e^{\frac{-\Delta C}{T_m}} \, d\Delta C$$

Where T_m is the temperature of the Simulated Annealing process. We can now define the *Cost Force Ratio*, (CFR) as:

$$CFR = \frac{E_-}{E_+ + E_-} \times 100 \tag{4}$$

The closer CFR is to 50% (the expected value of the good moves being equal to the expected values of the bad moves, $E_- = E_+$) the closer the system is to equilibrium.

Figure 3 is a plot of CFR versus generated move number for a Simulated Annealing process running on circuit Primary1, as it goes from non-equilibrium to equilibrium at temperature 400 changing to 300. CFR is determined by keeping a window of ΔC values multiplied by the P_{Accept} function and using this to calculate E_+ and E_-. In this figure the CFR comes down from an initial value of 55% and hovers around 50%. This shows that the CFR indicates when equilibrium has been achieved. It varies about the 50% point due to the stochastic nature of the algorithm and the approximation of measuring the CFR in a finite window.

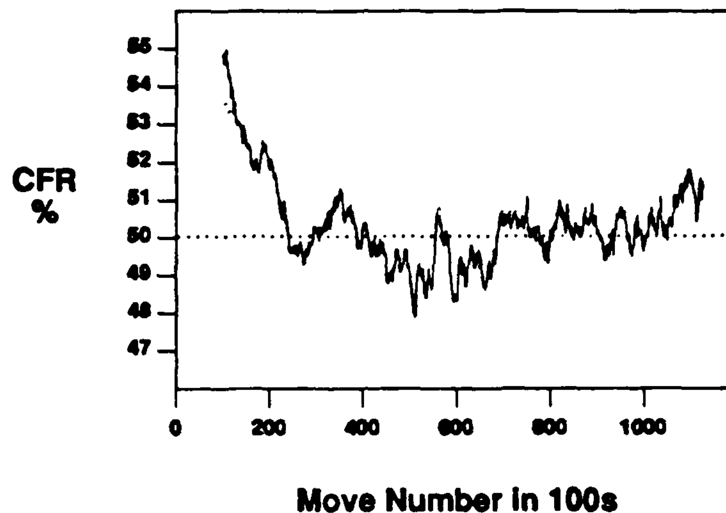


Figure 3 - *CFR versus Move Number as Process Achieves Equilibrium*

3 Measuring Temperature

As defined in Section 2, the temperature of a placement is the temperature at which the Simulated Annealing process running on the placement is in equilibrium. In this section we present a method for measuring the temperature of an arbitrary placement.

The method is called the CFR Binary Search and has the following outline:

1. *Measure $P(\Delta C)$ for the given circuit under the Simulated Annealing process.* This is discussed in detail in Section 3.1.

2. Set the starting search temperature, T_m, arbitrarily.

3. Based on the current T_m, calculate:

$$P_{Accept}(\Delta C) = e^{\frac{-\Delta C}{T_m}} \qquad \Delta C > 0$$

$$= 1 \qquad \Delta C \leq 0$$

4. Calculate the *effective* probability distribution: $P_{eff}(\Delta C) = P(\Delta C) \times P_{Accept}(\Delta C)$. $P_{eff}(\Delta C)$ is the probability that a move with cost ΔC will be both generated and accepted.

5. Calculate the Cost Force Ratio, CFR, using $P_{eff}(\Delta C)$ and

$$E_- = \left| \int_{-\infty}^{0} \Delta C \; P_{eff}(\Delta C)\, d\Delta C \right|$$

$$E_+ = \int_{0}^{\infty} \Delta C \; P_{eff}(\Delta C)\, d\Delta C$$

and equation (4).

6. If CFR < 50, reduce T_m according to a binary search and go to step 3;
   If CFR > 50, increase T_m according to a binary search and go to step 3.

7. When CFR = 50, T_m is the equilibrium temperature, T_{eq}. Finish.

Each iteration of the CFR Binary Search requires only the recalculation of the positive portion of the acceptance function probability, $P_{Accept}(\Delta C)$, and subsequently E_+ and CFR since E_- does not change with T_m. Note also that $P(\Delta C)$ need only be generated once. This is important since it takes many moves (10^4 to 10^5) to get an accurate picture of the probability distribution.

3.1 Measurement of the Probability Distribution

A key and difficult step in the CFR Binary Search temperature measurement procedure is the measurement of the distribution $P(\Delta C)$. There are two potential methods:

1. Static Measurement. $P(\Delta C)$ is measured by generating *virtual* moves in the Simulated Annealing process on the placement, and recording the frequency with which each cost occurs. That is, moves are generated in the usual manner, but none are accepted, and so the placement does not change.

2. Dynamic Measurement. $P(\Delta C)$ is measured by generating *and* accepting moves on the placement. Here the placement does change as the measurement is made.

For the general case of any Simulated Annealing application a static measurement will not give the correct distribution. This is because a static measurement of $P(\Delta C)$ could be taken when the system was at a local (but not global) optimum. In this case there would be no good (negative) moves generated and since E_- would appear to be 0, the temperature would also appear to be 0, which is incorrect in the case of a local optimum. This is an example of an extreme case, but similar problems can occur when the state is at or near discontinuities in the energy landscape.

It is not possible, however, to measure dynamically the distribution while running a Simulated Annealing process *at* the placement's equilibrium temperature because that temperature is what we are seeking. If $P(\Delta C)$ is measured at the wrong temperature, then the placement's temperature will actually

change, and the $P(\Delta C)$ will reflect the temperature of the measuring process rather than the true temperature. This is not unlike the Heisenberg uncertainty principle - the act of measuring the temperature can cause the temperature to change.

An alternative is to measure $P(\Delta C)$ using the static method, and to determine how accurate this is as an approximation. The accuracy of this approach, with respect to dynamic measurement is entirely problem dependent - it depends on the energy landscape of the underlying Simulated Annealing formulation. We have experimented to determine the accuracy for the standard cell placement problem and have found that the static measurment of $P(\Delta C)$ is almost exactly the same as the dynamic measurement. Figure 4 shows a plot of a static distribution and a dynamic distribution measured on circuit Primary1 at temperature 300. Measurements and numerical comparisons on this and several other circuits at various temperatures have shown very small differences between the static and dynamic measurements. Thus we will use the static measurement of $P(\Delta C)$ in the temperature measurement algorithm.

Note that this can only be done because of the nature of the placement problem and the specific Simulated Annealing formulation - it is not a general result for all Simulated Annealing problems.

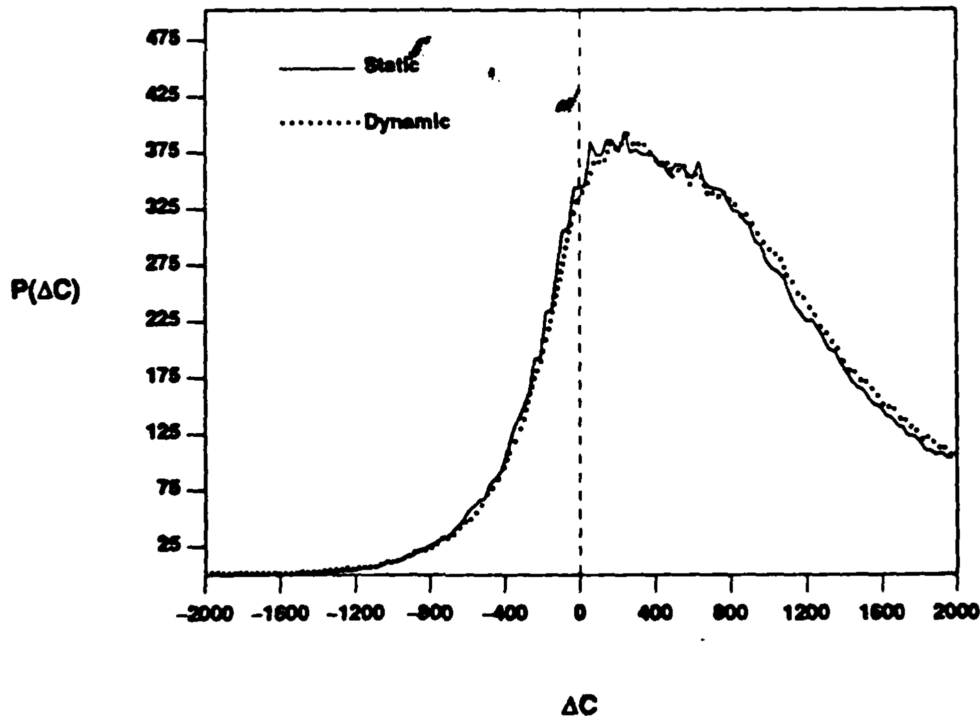


Figure 4 - *Comparison of Static and Dynamic Measurement of $P(\Delta C)$*

3.2 Temperature Measurements of Simulated Annealing Placements

The CFR Binary Search was used to measure the temperature of a set of Primary1 placements produced by the SALTOR Simulated Annealing placement program [Rose86,Rose88a]. Each placement was measured by using N = 100,000 virtual moves to experimentally determine $P(\Delta C)$. Table 1 gives the temperature at which each placement's Simulated Annealing process was terminated (while in equilibrium), and the measured temperature using the CFR Binary Search.

SA Produced Temperature	CFR Binary Search Measured Temp	Difference
500	496	-4
405	420	+15
294	285	-11
213	215	+2
153	164	+11
99	97	-2
57	60	+3
28	28	0
9	15	+6
2	4	+2

Table 1 - *Temperature Measurement of Simulated Annealing Placements*

The measured temperature is quite accurate at the higher temperature, usually less than 7% error. The lower temperature measurements are *proportionately* less accurate, but since their absolute values are small this is not surprising. The error is due to three effects:

1. The cooling schedule used to produce the placement is not perfect, and so the placement is probably not quite in equilibrium.

2. The slight difference, as discussed above, between the static and the (more correct) dynamic measurement of $P(\Delta C)$.

3. At lower temperatures, there are fewer negative moves, and so the accuracy of E_- decreases, decreasing the accuracy of CFR and hence the temperature measurement.

This last point can be seen experimentally: figure 5 is a plot of the percentage standard deviation of the measured temperature as a function of the number of virtual moves, N, for temperatures 28, 153 and 405. The standard deviation was calculated from fives runs at each number of virtual moves. The variation is a

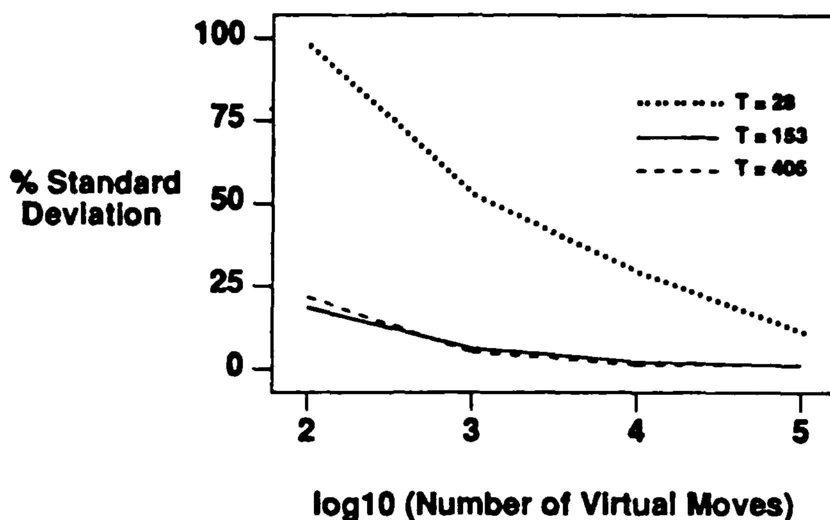


Figure 5 - *Variation of Temperature with* $\log_{10}$ *(Number Virtual Moves)*

decreasing function of N, as would be expected. The increase in percentage variation at lower temperatures is illustrated as described above.

4 Temperature Measurement of Min-Cut Placements

The reason for measuring the temperature of a placement is to be able to switch from a non-annealing algorithm to an annealing-based one, and to begin at the correct temperature. In this section we first define a few relevant terms, then discuss the feasability of measuring non-annealing placements, and finally measure a set of placements produced by the Min-Cut placement algorithm [Breu77,Dunl85].

4.1 Definition of Terms

Several terms first need to be defined for Min-Cut placements, as shown in Figure 6. A Min-Cut placement algorithm is characterized by, among other things, the order and spacing of the cut lines applied. In Figure 6, the rectangle represents the entire placement, over which is laid a set of vertical and horizontal cut lines. If the spacing of the vertical cut lines is V and of the horizontal cut lines is H, then the *cut area*, A, is given by $A = V \times H$.

4.2 Feasability and Matching of Algorithms

One difficulty with measuring the temperature of non-annealing produced placements is that the definition of temperature presented in Section 2 depends on the associated Simulated Annealing process being in equilibrium. It is clear, however, that a placement produced by the non-annealing algorithm is not

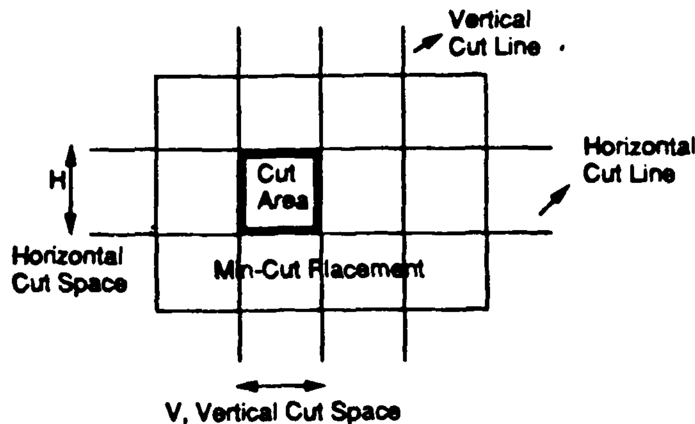


Figure 6 - *Definition of Cut-Area*

in equilibrium. Thus we must make an approximation and assume that a min-cut placement can be thought of as being in equilibrium at some temperature. The effect of this approximation is measured in the next section where we compare the CFR Binary Search method with a more direct method.

Next we must take into account a mismatch between Min-Cut placement and the Simulated Annealing *move set* used in Timberwolf [Sech85] and SALTOR [Rose88a]. This move set allows cells to overlap and penalizes that overlap. The Min-Cut placement, however, has no overlap. Thus the first moves made on the Min-Cut placement during a Simulated Annealing process are more likely to be bad until a basic amount of overlap occurs, since almost every move will create some overlap where there was none before. This will shift the $P(\Delta C)$ distribution to the right and give erroneous results for a measured temperature. On the other hand, some Simulated Annealing algorithms, such as [Grov86], do not use overlap and would not have this problem. To avoid it here, we used a simplified circuit in which all cells were set to be of equal size and only exchange moves are made in the Simulated Annealing process. This prevents any overlap from occurring. Experimentally, we have seen that reasonable results are still obtained if overlap is allowed to occur, since the wire length portion of the cost function dominates the overlap.

4.3 Measurements

Using the CFR Binary Search method we measured the temperature of several Min-Cut placements with different cut areas. These placements were produced by the ALTOR standard-cell placement program [Rose85]. Table 2 gives the measured temperature for each placement and its cut area.

To check if the temperature measurements were correct, we measured the temperatures of the placements in a different way, called the *Delta Method*. The Delta Method finds the temperature of a placement by running an annealing process on the placement at a range of temperatures. It is run for 100 move generations per cell, for each temperature, and the percentage difference in absolute cost function is measured, called the delta. The temperature at which the absolute value of the delta is less than 2% is

the equilibrium temperature of the placement. This is a direct way of experimentally finding the temperature at which the change in cost function is near 0. The Delta Method requires much more computation than the CFR Binary Search method, and thus is of no practical use. Table 2 shows the temperatures determined by the delta method, and the difference between the the binary search method and the Delta method. The binary search temperature measurement of Min-Cut placements is not as accurate as those for Simulated-Annealing produced placements, yet it does track the temperature reasonably well.

Cut Area $\mu m^2 \times 10^4$	Temperature Measured Binary Search	Delta Method	Difference
2021	398	374	+24
1011	234	200	+34
505.3	162	132	+30
252.6	124	96	+28
126.3	91	67	+24
63.22	73	50	+23
31.58	49	40	+9
25.24	40	32	+8
12.60	34	30	+4
7.697	29	27	+2
3.139	28	26	+2

Table 2 - *Temperature Measurement of Min-Cut Placements*

The CFR Binary Search method consistently overestimates the equilibrium temperature. due to the fact that a min-cut placement is not in equilibrium, as discussed in section 4.2. A more specific reason for this is that the Min-Cut placement leaves several particularly good moves possible, because of its lesser hill-climbing ability (we used [Fidu82] as the partitioning algorithm). A Simulated Annealing process would quickly correct these, but they result in an overestimation of E_- and hence a temperature that is too high.

4.4 Comments

Intuitively, one would expect the measured temperature of a Min-Cut placement to be an increasing function of the cut area, and this is observed in Table 2. This intuition comes from the notion that at higher temperatures, Simulated Annealing moves cells over large distances which determines a coarse placement. The first few cuts of Min-Cut placement, corresponding to a large cut area, also determines a coarse placement. At lower temperatures, Simulated Annealing makes moves that are much smaller in scope [Whit84] corresponding to the much smaller cut area of Min-Cut placement. The results of the

measurements shown in Table 2 bear out this intuition, as it is clear that the measured temperature is an increasing function of the cut area.

It is interesting to note the relationship between cut area and measured binary search temperature. We have found that the measured temperature is close to a linear function of the square root of the cut area ($\sqrt{A}$), as shown in Figure 7. The square root of the cut area is roughly equivalent to either the H or the V shown in Figure 6. This makes sense under the following line of reasoning: bad moves will move a distance from proportional to $\sqrt{A}$ since Min-Cut only places cells to an "accuracy" of $\sqrt{A}$. Assume that the cost of those moves is proportional to the move distance, as an approximation. The temperature that is likely to accept bad moves of cost $k \times \sqrt{A}$ is also proportional to $\sqrt{A}$ because moves are accepted with probability $e^{\frac{k \times \sqrt{A}}{T}}$. Hence the temperature is an approximate linear function of the distance $\sqrt{A}$.

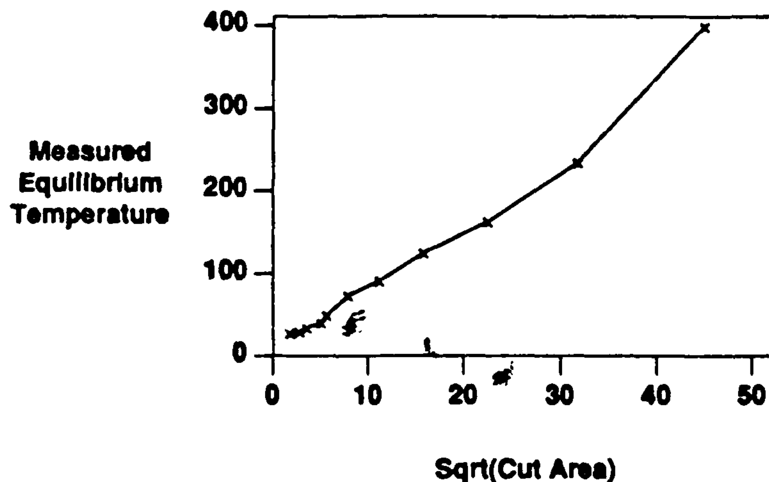


Figure 7 - *Plot of Measured Temperature vs. Sqrt Cut Area*

5 Conclusions

We have presented a method for determining the *temperature*, in the Simulated Annealing sense, of an arbitrary placement. It uses a new view of Simulated Annealing state that is based on the probability distribution of the change in cost function. This view provides a new definition of equilibrium, a measure of the nearness of a Simulated Annealing state to equilibrium, and an interesting perpsective on equilibrium dynamics.

The temperature of several Simulated Annealing placements have been measured with good accuracy. The temperature of a set of Min-Cut placements has been measured with reasonable accuracy, and we have demonstrated an experimental relationship between cut area and temperature. These measurements are useful for determining the starting temperature when switching from a non-annealing

based placement strategy to an annealing-based one.

6 References

Aart85
E.H.L Aarts, P.J.M. van Laarhoven, "A New Polynomial-Time Cooling Schedule," Proc. ICCAD 85, November 1985, pp. 206-208.

Breu77
M.A. Breuer, "Min-Cut Placement," Journal of Design Automation and Fault-Tolerant Computing, Oct 1977, pp 343-362.

Dunl85
A.E. Dunlop, B.W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," IEEE Transactions on CAD, Vol. CAD-4, No. 1, January 1985, pp 92-98.

Fidu82
C.M. Fiduccia, R.M. Matheyses, "A Linear Time Heuristic for Improving Network Partitions," Proc. 19th Design Automation Conference, June 1982, pp 175-181.

Grov86
L.K. Grover, "A New Simulated Annealing Algorithm for Standard Cell Placement," Proc. ICCAD 86, November 1986, pp. 378-380.

Grov87
L.K. Grover, "Standard Cell Placement Using Simulated Sintering," Proc. 24th DAC, June 1987, pp. 56 - 59.

Laar87
P.J.M. van Laarhoven and E.H.L Aarts, **Simulated Annealing: Theory and Applications,**" D Reidel Publishing Co., Dordrecht, Holland, 1987.

Prea87
B.T. Preas, "Benchmarks for Cell-Based Layout Systems," Proc. 24rd Design Automation Conference, June 1987, pp. 319-320.

Rome84
F. Romeo, A. Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications," Memorandum No. UCB/ERL M84/34, March 1984, Electronics Research Laboratory, University of California, Berkeley.

Rose85
J.S. Rose, W.M. Snelgrove, Z.G. Vranesic, "ALTOR: An Automatic Standard Cell Layout Program," Proc. Canadian Conference on VLSI, November 1985, pp. 168-173.

Rose86a
J.S. Rose, "Fast, High Quality VLSI Placement on an MIMD Multiprocessor," Ph.D. Thesis, Department of Electrical Engineering, University of Toronto 1986; also Computer Systems Research Institute Technical Report # 189.

Rose86b
J.S. Rose, D.R. Blythe, W.M. Snelgrove, Z.G. Vranesic, "Fast, High Quality VLSI Placement on an MIMD Multiprocessor," Proc. ICCAD 86, November 1986, pp. 42-45.

Rose88a
J.S. Rose, W.M. Snelgrove, Z.G. Vranesic, "Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing," IEEE Transactions on CAD, Vol. 7, No.3, March 1988, pp. 387-396.

Rose88b
J.S. Rose, W. Klebsch, J. Wolf, "Temperature Measurement of Simulated Annealing Placements," to appear in Proc. ICCAD 1988.

Sech85
C. Sechen, A. Sangiovanni-Vincentelli, "The Timberwolf Placement and Routing Package," IEEE JSSC, Vol. SC-20, No. 2, April 1985, pp 510-522.

Whit84
S.R. White, "Concepts of Scale in Simulated Annealing," Proc. Int. Conf. on Computer Design, October 1984, pp. 646-651.

The Parallel Decomposition and Implementation of an Integrated Circuit Global Router

Jonathan Rose
Computer Systems Laboratory,
The Center for Integrated Systems,
Stanford University, Stanford, CA 94305

Abstract

Better quality automatic layout of integrated circuits can be obtained by combining the placement and routing phases so that routing is used as the *cost function* for placement optimization. Conventional routers are too slow to make this feasible, and so this paper presents a parallel decomposition and implementation of an integrated circuit global router. The *LocusRoute* router is divided into three orthogonal "axes" of parallelism: routing several wires at once, routing segments of a wire in parallel, and dividing up the potential routes of a segment among different processors to be evaluated. The implementation of two of these approaches achieve significant speedup - wire-by-wire parallelism attains speedups from 6.9 to 13.6 using sixteen processors, and route-by-route achieves up to 4.6 using eight processors. When combined, these approaches can potentially provide speedups of as much as 55 times.

1 Introduction

The task of automatic layout of integrated circuits has traditionally consisted of two parts: automatic *placement* where the circuit modules are positioned and automatic *routing* in which the paths of the connecting wires are determined. The objective of both tasks is to result in a layout with as little area as possible. The best way to evaluate the "goodness" of a placement is to route it and determine its final area. Up to now this has not been feasible because routing itself is a difficult combinatorial optimization problem and common heuristics have been too slow to be used in this way.

The advent of usable commercial multiprocessors, with potentially enormous aggregate computation power may change this view if automatic routing can be decomposed into tasks that can be efficiently run in parallel. The aim of the *Locus Project* at Stanford University is to combine placement and routing into one optimization process, and to do this t using multiprocessing to increase the speed of the routing.

This paper presents the parallel decomposition and implementation of the *LocusRoute* global router for integrated circuits. The goal of the router is to make the average routing time for one wire close to the time that it takes to recalculate more conventional cost functions. This means that the routing time must be on the order of one to five milliseconds per wire on a VAX 11/780-class machine [Sech85]. The intention is for the global router to be invoked to rip-up and re-route wires whose end points have changed when one or more cells are moved in an iterative improvement placement scheme.

Prior work on parallel routing (see [Blan84] for a survey) has been done in isolation from the placement problem and has generally focused on the Lee routing algorithm [Lee61]. In most cases the algorithm has been fixed in hardware and as such lacks the flexibility that is always required in practical CAD software such as the global router described in [Yama85]. A far more versatile approach is to use general purpose parallel processors, which allow an application to be tuned in a manner similar to uniprocessors. Using the flexibility of a general purpose multiprocessor, several "axes" of parallelism can be exploited. If these axes are *orthogonal* to each other then when used together they can provide significant speedup. Two approaches to parallelizing an algorithm are said to be orthogonal if, when used together, the resulting speedup is the product of the speedup of the individual methods.

The basic idea of the LocusRoute algorithm is to investigate a subset of the two-bend routes between pairs of pins to be routed. The uniprocessor LocusRoute program can route wires in average times from 45 ms to

935 ms on a DEC Micro Vax II depending on the size of the circuit. The routing speed is increased by parallelizing the algorithm in three ways: routing several wires at once, routing several two-point segments simultaneously, and evaluating possible two-bend routes in parallel. The wire-by-wire parallel approach achieves speedups ranging from 6.9 to 13.6 using sixteen processors. The route-by-route approach achieves speedups of up to 4.6 using eight processors. These two axes of parallelism are orthogonal to each other.

This paper is organized as follows: Section 2 describes the standard cell layout methodology and defines the associated global routing problem. Section 3 describes the uniprocessor LocusRoute algorithm. Section 4 presents three approaches for speeding up the router using parallel processing, and gives performance results.

2 Standard Cell Layout

The standard cell-style layout is a common circuit design methodology in which all circuit modules are of equal height and are "butted" together to form rows as shown in Figure 1.

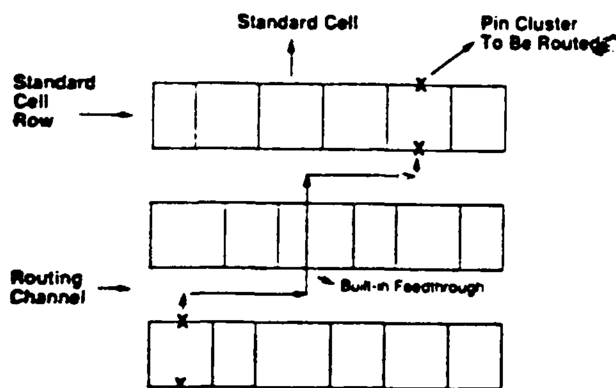


Figure 1 - *Standard Cell Layout*

Power and ground wires run horizontally through the cells and are connected by abutment. Cells have connection points on their top and bottom and typically one logical pin has two physical pins on each. This group of pins is called a *pin cluster*. Connections between adjacent rows are made by routing wires in the horizontal *routing channels* as shown in Figure 1. If a connection is required between two non-adjacent rows then either feedthrough cells are inserted in the intervening rows to make room for vertical connections or an uncommitted path in an existing cell (called a "built-in feedthrough") is used.

2.1 Problem Definition

Global routing for standard cells decides the following for each wire: First, for each pin cluster it decides which of the physical pins are actually to be connected. Second, if there is no path between channels when one is required, it must decide either which built-in feedthrough to use or where to insert a feedthrough cell. Lastly, it must decide which channel to use in the route from a pad into the core cells. The objective is to minimize the sum of the maximum widths of each routing channel (hereafter called the *total density*), and in so doing minimize the final area.

In this discussion of global routing there will be no differentiation between feedthrough cells and built-in feedthroughs - they are referred to jointly as *vertical hops*. The decision to insert a feedthrough cell or use a built-in feedthrough is deferred to a post-processing step [Rose88b].

3 A Standard Cell Global Router

This section gives a brief description of the LocusRoute global router. A more complete discussion can be found in [Rose88b].

3.1 Routing Model

The LocusRoute algorithm uses the following routing model: Each possible routing position in a channel (also called *routing grid* of that channel) is represented as one element of an array as shown in Figure 2. The array, called the *Cost Array*, has a vertical dimension of the number of rows plus one, and a horizontal dimension of the width of the placement in routing grids. Each element of the Cost Array contains two values: H_{ij} and V_{ij}. H_{ij} contains the number of of wire routes that pass horizontally through the grid at channel i in position j. V_{ij} is the cost, assigned by parameter, of traversing a row in travelling from channel i to channel $i+1$ at grid position j. The routing problem for a wire is represented as a list of (i, j) pairs of locations in the Cost Array, corresponding to the locations of pins to be joined.

Under this model, the objective is to find a minimum-cost path for each wire. The wire's cost is given by the sum of all of the H_{ij} and V_{ij} that it traverses. After a wire is routed through location (i, j)

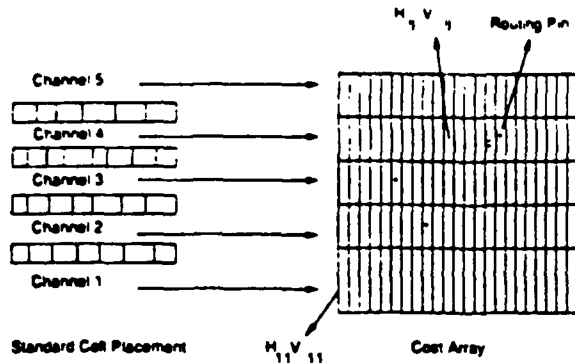


Figure 2 - *Routing Model*

its presence is recorded in the Cost Array (i.e. H_{ij} is incremented, as is V_{ij} if the direction is vertical) so that subsequent wires can take it into account. Thus the more wires going through a particular location in a channel, the less likely it is that area will be used.

3.2 The Global Routing Algorithm

There are five main steps in the LocusRoute global routing algorithm for standard cells. They are:

1. A multi-point wire is decomposed into two-point *segments*, by finding its minimum spanning tree using Kruskal's algorithm [Krus56].

2. The segments are further decomposed, if necessary, into *permutations*, which are the set of possible routes between each pin in a pin cluster. There are four possible routes, one between each of the two physical pins in each pin cluster. It has been experimentally determined that only when the clusters are greater than a certain horizontal distance apart (about 300 routing grids) is it necessary to evaluate all four permutations. Less than this distance, only the closest pin pair need be evaluated.

3. A low-cost path in the Cost Array is found for each permutation by evaluating a subset of the two-bend routes between each pin pair. The permutation with the best cost is selected as the route for that segment. This step is described in further detail below, in Section 3.3.

4. Traceback. This is a cleanup step that provides enough information for later detailed routing.

5. Wire *lay down*. The presence of the newly routed wire is put into the Cost Array by incrementing the array elements where the new wire resides. Once there, other wires can take it into account.

3.3 Route Evaluation

The LocusRoute algorithm searches for a low-cost path for a permutation by evaluating a number of different routes. The idea is to determine the cost of a subset of all two-bend routes between the two pins, and then choose the one with the lowest cost. Figure 3 illustrates three possible two-bend (or less) routes inside a representation of the Cost Array as a small example.

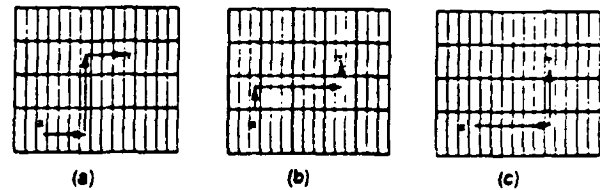


Figure 3 - *Sample Two-Bend Routes*

If the horizontal distance between the two pins is H routing grids, and the vertical difference in channels between the pins is C, then the total number of two-bend routes is $C+H$. A parameter, called the *two bend percent* (TBP) dictates the percentage of the total number possible two-bend routes to be evaluated. Thus the total number of routes evaluated is given by $\frac{TBP}{100}\times(C+H)$. When TBP is less than 100, then the routes are evaluated in a priority order [Rose88b]. Experimentally, it was determined that a TBP of 20% would result in a path as good as that found by an exhaustive maze router, as compared on the basis of total density for the entire circuit.

The LocusRoute algorithm makes use of a general iterative technique in the manner described in [Nair87]. Briefly, this means that after the first time all wires are routed, each is sequentially ripped up from the Cost Array, and then re-routed. By routing each wire several times (typically four is sufficient), the wire order-dependency is reduced and the final answer is improved by five to ten percent.

The uniprocessor LocusRoute algorithm compares favorably with a widely used placement and global routing package [Sech85], and with a good quality industrial global router [Rose88b].

4 Parallel Decomposition & Implementation

In this section several ways of parallelizing the LocusRoute router are proposed and implemented. Figure 4 illustrates several such axes of parallelism:

1. Wire-based Parallelism. Each processor is given an entire multi-point wire to route.

2. Segment-based Parallelism. Each two-point segment produced by the Kruskal decomposition can be routed in parallel.

3. Permutation-based Parallelism. Each of the four possible permutations, as discussed in Section 3.2, can be evaluated in parallel.

4. Route-based Parallelism. Each of the possible two-bend routes for every permutation can be evaluated in parallel.

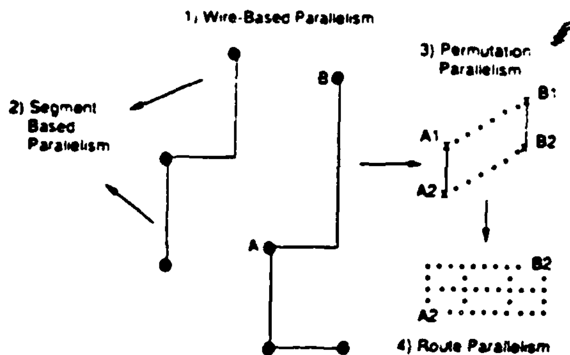


Figure 4 - *Parallel Decomposition of LocusRoute*

Note that these are only *potential* axes of parallelism. It is possible to eliminate some of them as uneconomical by using statistical run-time measurements of the sequential router. For example, the number of two-point segments that actually need to have all four permutations evaluated is quite small with respect to the total. Thus, permutation-based parallelism is not going to provide significant speedup and isn't worth the time it requires to develop. On the other hand, other measurements show that the time spent evaluating the cost of two-bend routes ranges from 50 to 90 percent of the total routing time, so that some amount of speedup from route-based parallelism can be expected.

To date, we have not considered pipelining as an axis of parallelism. A pipeline implementation would have the same stages as the basic algorithm described in Section 3.2. To some extent, pipelining uses the same axis of parallelism as wire-based parallelism since it also routes several wires at once. The best use of pipelining would be to execute the first two stages, segment and permutation decomposition, for all wires in parallel since these stages have no data dependencies on the routing of other wires. In the context of iterative improvement placement, however, the wire positions will not be known in advance as they are when considering the routing problem in isolation.

Each of the following sections discusses the details of the axes of parallelism that have been implemented. In the case where the quality of the answer of the parallel program is worse than the sequential program, a quantitative measure of the amount of degradation is given. This section is concluded by a discussion of the combination of two of the axes of parallelism. All decompositions assume a shared-memory multiprocessor.

4.1 Wire-Based Parallelism

In Wire-Based parallelism, each multi-point wire is given to a separate processor, which runs the LocusRoute routing algorithm as described in Section 3: prior to decomposition, if the iteration technique is used, the wire must be "ripped up" out of the Cost Array. Next, each wire is decomposed into two-point wires, and possibly further into permutations. A subset of the potential two-bend routes is generated, and then evaluated by traversing the Cost Array. When a final route is chosen, the Cost Array is updated to reflect the new presence of that route.

The Cost Array is a shared data structure to which all processors have read and write access. Other than a task queue, the cost array is the only shared piece of data. This is an excellent axis of parallelism: if the sharing of the Cost Array does not cause performance degradation due to memory contention, the speedup should simply be the number of wires that are routed in parallel. The resulting parallel answer, however, will not necessarily be the same as the sequential answer. The problem is the sequential router has complete knowledge of all wires that have already been routed, by virtue of their presence

in the cost array. The parallel router has less information because it doesn't see the wires that are being routed simultaneously. The more wires routed in parallel, the less information each processor has to choose good routes that avoid congestion and hence the total density increases. Thus the total density will increase as the number of processors increases. The measured effect on total density is discussed below, in Section 4.1.1.

4.1.1 Wire-Based Parallel Results

Figure 5 is a plot of the speedup versus number of processors for a 3029-wire example running on an sixteen-processor shared-memory Encore MULTIMAX. The Encore uses National 32032 chip sets which, in our benchmarks, timed out slightly faster than a DEC Micro Vax II. The speedup for p processors, S_p is calculated as $\frac{T_1}{T_p}$, where T_1 is the execution time on one processor and T_p is the execution time using p processors. The execution time measured *does not* include the time for input of the circuit, only the actual routing computation time. For this circuit the increase in total density due to the missing "knowledge" effect described in Section 4.1 from 1 to 16 processors is 6%, and the number of vertical hops increases 2%.

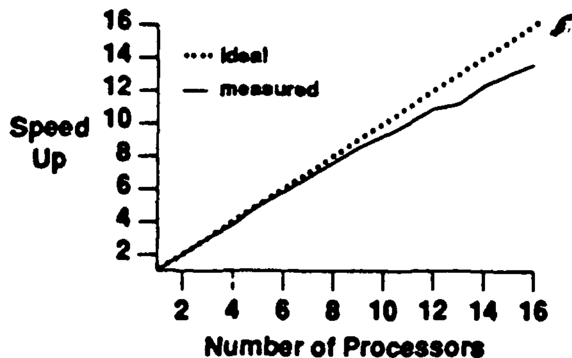


Figure 5 - *Wire-Based Speedup for 3029-Wire Circuit*

The program was run on several other circuits, which are from several sources: The standard cell benchmark suite (Primary1, Primary2, Test06 [Prea87]), Bell-Northern Research Ltd. (BNRA-BNRE), and the University of Toronto Microelectronic Development Centre (MDC). The placement for all of the circuits was done by the ALTOR standard cell placement program [Rose85,Rose88a]. Table 2 gives the execution time and speedup using 1, 8 and 15 processors, for all the test circuits. The execution time is for four iterations over all the wires. The speedup ranges from 5.4 for a smaller circuit to 7.6 for the largest, using 8 processors. It ranges from 6.9 to 13.6 using 15 processors. The speedup is less for smaller circuits because they are done in such a short time, and the startup overhead becomes a factor.

Circuit Name	# Wires	1 Pr T (s)	8 Pr T (s)	15 Pr T (s)	8 Pr Spdup	15 Pr Spdup
BNRE	420	70.0	13.0	10.2	5.4	6.9
MDC	575	79.3	14.3	11.2	5.5	7.1
BNRD	774	139	21.0	16.0	6.6	8.7
Primary1	904	279	43.4	33.6	6.4	8.3
BNRC	937	198	30.5	21.4	6.5	9.3
BNRB	1364	565	84.6	63.4	6.7	8.9
BNRA	1634	725	112	77.8	6.5	9.3
Test06	1673	5684	797	465	7.1	12.2
Primary2	3029	3950	517	290	7.6	13.6

Table 2 - *Performance of Wire-Based Parallelism*

Table 3 gives the total density and vertical hop counts using 1, 8 and 15 processors. The increase in total density ranges between 1% to 7% for 15 processors. The increase in vertical hops is ranges from 1% to 9% but is generally less than 4%. In the placement context this level of degradation is tolerable. In the future, however, on machines with more processors, it will likely become more of a problem. We have considered three ways of reducing the effect of the missing knowledge due to simultaneous routing of wires. The first is to try to ensure that the different processors only deal with wires that are in distinct physical areas, so that the wires routed simultaneously do not interact. The second way to reduce processor interference is not to rip up a route until the new route is determined. In this way there is a much shorter period of time in which the cost array does not contain the presence of the wire. This severely degrades the new route of the wire itself, however, since it sees the old copy of itself while evaluating potential routes. Experimentally, the degradation was sufficient to nullify any gain from the approach. A third method not yet implemented is to route the wires in a different order for each iteration, (iteration is described in Section 3.3) so that the knowledge missing in one iteration is different from that in another.

Circuit Name	Total Density 1 Pr	Total Density 15 Pr	Total Density % More	Vertical Hops 1 Pr	Vertical Hops 15 Pr	Vertical Hops % More
BNRE	130	134	3%	449	490	9%
MDC	134	142	6%	241	243	1%
BNRD	176	181	3%	530	572	8%
Primary1	262	269	3%	940	966	3%
BNRC	191	193	1%	739	772	4%
BNRB	307	325	6%	1904	1974	4%
BNRA	298	320	7%	2106	2197	4%
Test06	318	338	6%	3221	3286	2%
Primary2	560	592	6%	3053	3126	2%

Table 3 - *Quality of Wire-Based Parallelism*

4.1.2 Gain Due to Removal of Locks

An interesting issue is whether or not each processor should lock the Cost Array as it both rips up and re-routes wires in the Cost Array. The act of ripping up a route is essentially a decrement, and re-routing is an increment on a set of cells in the Cost Array. Locking the Cost Array during these operations ensures that two simultaneous operations on the same element does not prevent one of the operations from being lost. It does, however, cause a significant performance degradation. For example, for the Primary1 circuit the speedup decreased from 8.3 to 6.4 using 15 processors when Cost Array locking was used. For the Primary2 circuit the speedup for 15 processors was reduced to 12.1 from 13.0 due to locking.

The final routing quality, however, does not decrease when locking is omitted. The reason for this is that the probability of two processors accessing the *same* Cost Array element (of which there are on the order of 10000) at the *same* instant is very low. Even if very few increment or decrement operations are lost, the effect on final quality is negligible since only a few elements would be wrong by a small amount. This was shown experimentally by performing ten runs with 15 processors on each of the above circuits, for both the locking and non-locking cases. Table 1 gives for the two circuits the average running time, and the average and standard deviation of the total density and number of vertical hops. From this table it can be seen that the quality in both cases is very nearly the same. Note that in a placement context in which many more wires will be ripped up and re-routed, the effect of these small errors would be cumulative and so an occasional correction step may be necessary if locks are not used.

Circuit & Lock Type	Avg T (s)	Density Avg.	Density SD	Vertical Hops Avg	Vertical Hops SD
Primary1 Locks	43.8	269	2.0	962	4.9
Primary1 NO Locks	33.7	272	3.0	964	3.4
Primary2 Locks	325	591	1.9	3126	7.5
Primary2 NO Locks	303	591	4.9	3122	4.0

Table 1 - *Speed & Quality Using and Not Using Locks*

4.2 Segment-Based Parallelism

In segment-based parallelism, each two-point segment of a wire is given to a different processor to route. This is the stage following the Kruskal decomposition, but prior to the evaluation of different two-bend routes. Measurements of the sequential router showed that about 60% of the routing time was spent on wires with more than one segment. On the surface this implies that a speedup of about two could be achieved using three processors. Unfortunately, this is not the case. Even though there are many wires that provide two or three-way parallel tasks, the size of those tasks are not necessarily equal. The amount of time taken by LocusRoute to route two points is proportional to the manhattan distance between the two points. If, in a three-point wire, two of the points are close together and the third is far away, it will then take much longer to route one segment than the other. Thus the processor assigned to the short segment will be idle while the longer one is being routed. This unequal load prevents a reasonable speedup. On the test circuits a speedup of about 1.1 using two processors was measured.

It is fairly clear, however, that an extra processor could be assigned to a *number* of processors that are routing different wires. It is likely that at any given time, one of them will be able to use the extra processor to route multiple segments. Though every processor won't be able to use a second processor all the time, some number of processors can be used in this way. This technique would become essential if many processors were used in wire-based parallelism, at the point where the number of processors was close to the

number of wires. In that case the load balance would become a problem in wire-based parallelism because wires with many segments take much longer than wires with few segments. Hence segment-based parallelism could be used to speed up the routing of the larger wires.

4.3 Route-Based Parallelism

In route-based parallelism all of the two-bend routes to be evaluated are divided among separate processors. Each finds the lowest-cost path among the set of two-bend routes that it is assigned. When all processors finish, the route with the best overall cost is selected. In this case the processor loads will be well-balanced because the routes are all of the same length, and the number of routes is evenly divided among the processors.

Figure 6 is a plot of the speedup versus number of processors for the circuit Test06, a large circuit. It achieves a speedup of 4.6 using 8 processors.

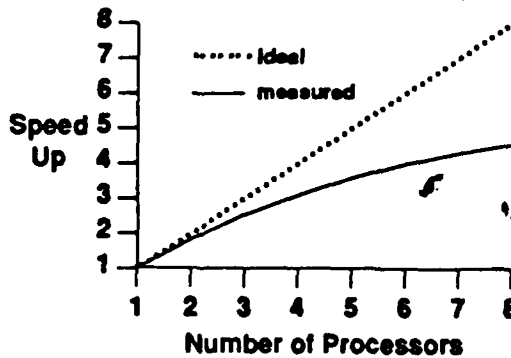


Figure 6 - *Route-Based Speedup for Circuit Test06*

Table 4 gives the best speedup achieved for all of the test circuits, ranging from 1.2 using 2 processors to 4.6 using 8 processors. The number of processors given for each circuit in the table are chosen by eye as to which number gives reasonable efficiency. It is clear that only the larger circuits benefit from more processors. The principal reason for the limitation in speedup is the sequential portion of the routing: the wire decomposition and the post-route processing that places the presence of the route into the Cost Array. On the small circuits that have lesser speedup, the sequential portion is about 50% of the total routing time, while on the larger circuits which have better speedup the sequential portion ranges from 10-15%. Other minor effects which degrade performance are the imbalance of processor task sizes due to integral numbers of routes and the fact that some segments have only a few potential routes.

Circuit Name	Best Route Speedup Speedup/#Proc
BNRE	1.2/2
MDC	1.3/2
BNRD	1.4/2
Primary1	1.8/3
BNRC	1.6/3
BNRB	2.1/4
BNRA	2.0/4
Test06	4.6/8
Primary2	3.3/5

Table 4 - *Performance of Route-Based Parallelism*

4.4 Combining Two Axes of Parallelism

The wire-based parallel and route-based parallel approaches are perfectly orthogonal; hence their speedups should multiply. Assume, for a given circuit that a speedup of S_w is achieved using wire-based parallelism on W processors, and a speedup of S_r is achieved using route-based parallelism on R processors. Then, because the two approaches are orthogonal, the resulting speedup when they are used together should be $S_w \times S_r$ using $W \times R$ processors. This model neglects the effect of memory contention that may occur when the number of processors is increased dramatically. Table 5 shows the best predicted speedup for the test circuits. Combined speedup ranges from 8.3 using 30 processors to 55 using 120 processors. The smaller circuits are routed very quickly and so it is difficult to get speedups greater than 10 due to the startup overhead. The larger circuits benefit greatly from the combination of the approaches.

Table 5 also contains the average routing time per wire on one processor, A_1, and what the the average routing time per wire would be under the maximum speedup, A_{RW}. That is, $A_{RW} = \frac{A_1}{S_w \times S_r}$. The average routing times for all circuits, under the various speedups range from 4.0ms to 17ms, and approaches our goal of one to five milliseconds per wire. It is interesting to note that even though the uniprocessor times are widely

Circuit Name	Total Density			Vertical Hops		
	1 Pr	15 Pr	% More	1 Pr	15 Pr	% More
BNRE	130	134	3%	449	490	9%
MDC	134	142	6%	241	243	1%
BNRD	176	181	3%	530	572	8%
Primary1	262	269	3%	940	966	3%
BNRC	191	193	1%	739	772	4%
BNRB	307	325	6%	1904	1974	4%
BNRA	298	320	7%	2106	2197	4%
Test06	318	338	6%	3221	3286	2%
Primary2	560	592	6%	3053	3126	2%

Table 3 - *Quality of Wire-Based Parallelism*

4.1.2 Gain Due to Removal of Locks

An interesting issue is whether or not each processor should lock the Cost Array as it both rips up and re-routes wires in the Cost Array. The act of ripping up a route is essentially a decrement, and re-routing is an increment on a set of cells in the Cost Array. Locking the Cost Array during these operations ensures that two simultaneous operations on the same element does not prevent one of the operations from being lost. It does, however, cause a significant performance degradation. For example, for the Primary1 circuit the speedup decreased from 8.3 to 6.4 using 15 processors when Cost Array locking was used. For the Primary2 circuit the speedup for 15 processors was reduced to 12.1 from 13.0 due to locking.

The final routing quality, however, does not decrease when locking is omitted. The reason for this is that the probability of two processors accessing the *same* Cost Array element (of which there are on the order of 10000) at the *same* instant is very low. Even if very few increment or decrement operations are lost, the effect on final quality is negligible since only a few elements would be wrong by a small amount. This was shown experimentally by performing ten runs with 15 processors on each of the above circuits, for both the locking and non-locking cases. Table 1 gives for the two circuits the average running time, and the average and standard deviation of the total density and number of vertical hops. From this table it can be seen that the quality in both cases is very nearly the same. Note that in a placement context in which many more wires will be ripped up and re-routed, the effect of these small errors would be cumulative and so an occasional correction step may be necessary if locks are not used.

Circuit & Lock Type	Avg	Density		Vertical Hops	
	T (s)	Avg.	SD	Avg	SD
Primary1 Locks	43.8	269	2.0	962	4.9
Primary1 NO Locks	33.7	272	3.0	964	3.4
Primary2 Locks	325	591	1.9	3126	7.5
Primary2 NO Locks	303	591	4.9	3122	4.0

Table 1 - *Speed & Quality Using and Not Using Locks*

4.2 Segment-Based Parallelism

In segment-based parallelism, each two-point segment of a wire is given to a different processor to route. This is the stage following the Kruskal decomposition, but prior to the evaluation of different two-bend routes. Measurements of the sequential router showed that about 60% of the routing time was spent on wires with more than one segment. On the surface this implies that a speedup of about two could be achieved using three processors. Unfortunately, this is not the case. Even though there are many wires that provide two or three-way parallel tasks, the size of those tasks are not necessarily equal. The amount of time taken by LocusRoute to route two points is proportional to the manhattan distance between the two points. If, in a three-point wire, two of the points are close together and the third is far away, it will then take much longer to route one segment than the other. Thus the processor assigned to the short segment will be idle while the longer one is being routed. This unequal load prevents a reasonable speedup. On the test circuits a speedup of about 1.1 using two processors was measured.

It is fairly clear, however, that an extra processor could be assigned to a *number* of processors that are routing different wires. It is likely that at any given time, one of them will be able to use the extra processor to route multiple segments. Though every processor won't be able to use a second processor all the time, some number of processors can be used in this way. This technique would become essential if many processors were used in wire-based parallelism, at the point where the number of processors was close to the

General Compiled Electrical Simulation

Daniel Weise
Stanford University
Computer Systems Laboratory
Center for Integrated Systems 207
Stanford, California 94305
(415) 725-3711

Scott Seligman
Stanford University
Computer Science Department
Margaret Jacks Hall
Stanford, California 94305
(415) 723-3088

Abstract: This report describes the initial results of our research into *General Compiled Electrical Simulation.* We use advanced compiler techniques to speed up electrical level simulation such as the type performed by SPICE. Our system creates a *simulation program* by compiling together a simulator and the circuit to be simulated. Our approach results not only in substantial speedups, but also in simpler simulators. An added benefit is that simulation programs can be parallelized much more effectively than a simulator can be parallelized.

Submitted to the 1989 Design Automation Conference

This research was supported by a CIS Seed Grant and Darpa Contract N00014-87-K-0828.

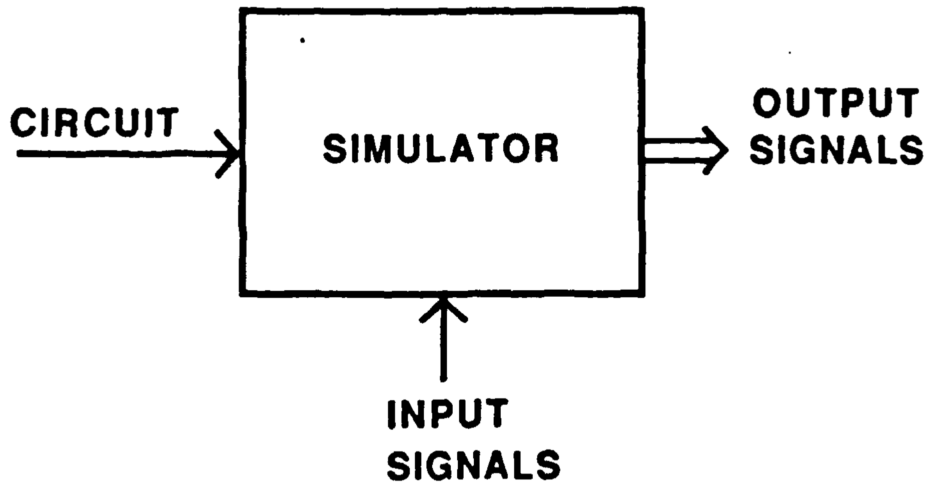
1 Introduction

Circuit simulation is vital to creating correctly functioning VLSI circuits. The advent of SPICE [Nagel] and other circuit simulators was a boon for circuit designers. Unfortunately, accurate electrical level simulation requires prohibitive amounts of computation for moderate or large circuits. Even when a given circuit takes an acceptable amount of time to simulate, it must be simulated many times before its design is finished, and the total simulation time can be very large. Most importantly, as circuit sizes increase, the size of circuits we wish to simulate grows as well. [White] recounts that at one major IC house more than 70% of an IBM 3090 is devoted to circuit simulation, and that at another house SPICE is run more than 10,000 times a month. Faster simulators will reduce the cost of designing functional silicon and improve the designs.

We are designing and building an interactive system for high speed electrical simulation of digital and analog circuits that is simple to use, programmable, much faster than any highly optimized simulator, capable of hierarchical and mixed mode simulation, and able to produce efficient code for parallel processors. We are employing four important ideas in the design of the system: general compiled simulation, embedded operation, object orientedness, and standard interfaces. Our ultimate goal is a software/hardware system costing around $15,000 that can sustain 100 MFLOPS on simulation problems.

This paper discusses the first of these ideas, *general compiled electrical simulation*. Standard circuit simulators accept a circuit and its input waveforms, and return the circuit's output waveforms. In general compiled electrical simulation (Figure 1), a simulator and a circuit are together compiled into a *simulation program* that, when run, has exactly the same behavior as running the simulator over the circuit, but runs many times faster. The simulation program's increased speed comes from compile-time unfolding of all constant data structures (such as the structure of the circuit itself) and from optimization of the unfolded code. Component values can be left symbolic during compilation so that recompilation isn't necessary when component values change. This feature increases the speedups for *What-If?* simulation and Monte Carlo analysis.

A simulation program consists of mostly straight line arithmetic code which uses few, if any, structured values. All opportunities for parallelism are explicit. A compiler can plan the spatial and temporal use of nearly every value, i.e., where the value is stored and when the value is computed. Therefore a simulation program will make very efficient use of heavily pipelined and parallel machines. A compiler will also be able to optimize register and data cache usage. Because the dynamic behavior of the simulation program is statically determined and all parallelism is explicit, economical special purpose hardware for executing the simulation program is feasible.



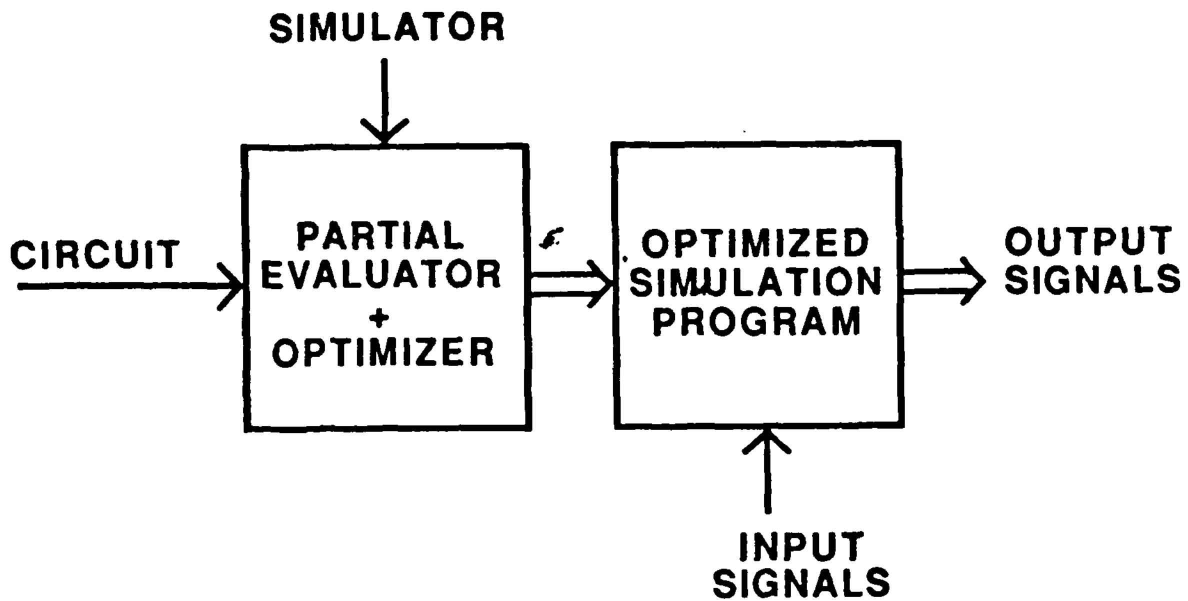


Figure 1: Normal Simulation and General Compiled Simulation. Normal simulation is depicted in the top figure. In normal simulation a simulator accepts a circuit and its input signals, and returns the circuit's output signals. In general compiled simulation, shown in the bottom figure, a *partial evaluator* accepts a simulator and a circuit, and produces a *simulation program* that, when applied to input signals, produces output signals.

A prototype of our system is implemented for linear circuits[1]. The simulation programs we produce run at least five times faster than [Spice3].

This paper has six sections. Background and related work are presented next, in Section 2. Section 3 describes simulation programs and the benefits of compiled simulation. Section 4 discusses general compiled simulation and presents our partial evaluator. Our results are given in Section 5. The last section presents a summary and outlines our future research.

2 Background and Related Work

The creation of simulation programs for faster simulation has been successfully implemented at the switch level, logic level, and behavioral level. Some work has begun at the analog level.

[Bryant] has written COSMOS, a program for compiled switch-level simulation. Not counting compilation time, COSMOS runs about ten times faster than its cousin MOSSIM. The speedups come from processing structure only once and from optimizing the resulting simulation program (actually, a system of boolean equations). At the logic level [Barzilai] describes a program called HSS that compiles logic expressions into System/370 assembler code. HSS can simulate around 240 million gate-patterns per second, a fairly respectable speed. [Hansen] has implemented a compiled simulation system called *Terse* for behavioral/logic simulation. *Terse* employs a rich set of symbolic input types to provide as much compile-time optimization as possible. Hansen doesn't quote specific speedups, but claims that *Terse* is instrumental in daily design work at MIPS.

Some compiled simulation was performed in early versions of SPICE [Nagel], where some of the matrix LU decomposition routines were turned into assembly code for each circuit. Considering that model evaluation was the CPU intensive part of his system, Nagel should have investigated compiled simulation for model evaluation. Unfortunately, he didn't have the technology for doing so. Another program that produced assembly code for matrix operations was [ASTAP]. More recently [Vladimirescu] reported on a more modern system that also compiles away the control portion of matrix solution.

[Lewis] has proposed compiled simulation and special purpose hardware for electrical simulation. He intends to build special purpose hardware that can perform simulations 500 times faster than uniprocessor systems. Our research differs from his in three major respects. First, we are interested in automatic general methods for compiled simulation whereas Lewis writes his simulation compilers by hand. Second, we are interested in designing

[1] Results and timings for non-linear circuits will be ready by the time final versions of papers are due for the inclusion in the proceedings.

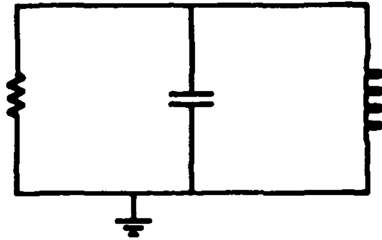


Figure 2: A simple RLC circuit

the cheapest, simplest hardware possible. Our goal is to sustain 100 MFLOPS for around $15,000. Third, we also want our compiler to produce efficient code for commercially available parallel processors. This paper addresses only the first difference.

3 Simulation Programs

Compiled simulation produces *simulation programs* that run much faster than an equivalent simulator simulating a circuit. We expect a tenfold speedup over a simply written simulator and a fivefold speedup over an optimized simulator. Two factors contribute to the speed of the simulation program: constant folding of the topology of the circuit, so that the topology is processed only at compile time, and optimizing the resulting program using techniques such as common subexpression elimination and dead code elimination. Constant folding the topology at compile time unfolds the loops that tranverse the structure of the circuit so that the circuit structure is embedded in the resulting program. This process yields virtually straight line code. For example, for a simple transient analysis simulator only two loops remain: the Newton-Raphson iteration for nonlinear analysis and the outer integration loop.

As an example of the power of compiled simulation we present a fragment of our transient analysis simulator (Figure 3) and the code produced for that fragment (Figure 4) when compiled against an RLC circuit (Figure 2). The simulator uses nodal analysis and trapezoidal integration. To generate the state at time $t+h$ from the state at time t, it creates a matrix M to be solved for node voltages at time $t+h$. From the node voltages at time $t+h$ and the state at time t it computes the branch currents at time $t+h$. The simulator computes matrix M by summing into it the current and conductance contributions of each component. The simulator is object oriented: each time-varying and reactive component carries its own function for computing its contribution to the matrix M. The methods must be retrieved and invoked at each time step.

Figure 4 depicts the simulation program that results from partially evaluating the func-

```
(define (next-state state)
  (let* ((matrix (+matrices
                   matrix
                   (create-integration-matrix circuit state h parameters)))
         (new-voltages (solve-matrix (trim-ground matrix)))
         (new-currents (compute-b-currents
                         circuit new-voltages state h parameters)))
    (create-new-state new-voltages new-currents (+ h (state-time state)))))

(define (create-integration-matrix circuit state h parameters)
  (let ((voltages (state-voltages state))
        (currents (state-currents state)))
    (let loop ((components (circuit-components circuit))
               (matrix (create-nxn+1-matrix (circuit-number-of-nodes circuit))))
      (if (null? components)
          matrix
          (loop (cdr components)
                ((2t-component-integration-method (car components))
                 matrix voltages currents h parameters))))))
```

Figure 3: Scheme Code Fragments for Transient Analysis. These two routines constitute the inner loop of transient analysis for linear circuits. The first function accepts a state at time t and returns the state at time $t + h$. It first calculates the node voltages by solving the matrix which is the sum of the "DC matrix" (the current and conductance contributions from resistors and time-invariant current sources) and the "integration matrix" (the current and conductance contributions from time varying and reactive elements). Once the voltages are computed the branch currents are computed. The function `create-integration-matrix` uses object oriented techniques to collect the conductance and current contributions of the reactive components to the admittance matrix. It retrieves the function for computing an element's contributions from the element and then invokes the function. We show these fragments to emphasize the amount of work the simulator must perform to compute the next state. Function retrieval and invocation occur for each component. A matrix is created and inverted for each time step. Figure 4 shows the power of compiled simulation by presenting the result of compiling this code for the simple RLC circuit of Figure 2. All intermediate structured values and control constructs completely vanish leaving only straightline arithmetic code.

```
(LAMBDA (STATE)
  (LET*
      ((TEMP1 (VREF STATE 1))
       (TEMP2 (VREF STATE 0))
       (TEMP3 (VREF TEMP1 1))
       (TEMP4 (VREF TEMP2 0))
       (TEMP5 (* TEMP4 .00005))
       (TEMP6 (+ TEMP5 TEMP3))
       (TEMP8 (VREF TEMP1 2))
       (TEMP9 (* TEMP4 .02))
       (TEMP10 (+ TEMP9 TEMP8))
       (TEMP12 (- TEMP10 TEMP6))
       (TEMP13 (VREF STATE 2))
       (TEMP14 (* TEMP12 49.6277915633))
       (TEMP15 (- TEMP14 TEMP4))
       (TEMP17 (* TEMP15 .02))
       (TEMP18 (- TEMP17 TEMP8))
       (TEMP19 (+ TEMP14 TEMP4))
       (TEMP21 (* TEMP19 .00005))
       (TEMP22 (+ TEMP21 TEMP3))
       (TEMP23 (* TEMP12 4.96277915633e-3))
       (TEMP24 (+ .1 TEMP13)))
    (VECTOR (VECTOR TEMP14) (VECTOR TEMP23 TEMP22 TEMP18) TEMP24)))
```

Figure 4: Compiled code for the transient analysis of the simple RLC circuit. This function accepts a state at time t and returns the state at time $t+h$. In this code h is 0.1 seconds. Not counting creating the next state, there are 20 instructions, of which 14 perform computation and 6 destructure the input state.
This code is optimal. Dead code elimination, constant folding, sign targeting, and arithmetic simplification are the major optimizations. For example, constant folding collapsed R, L, C, and H into constants such as .02 and 49.6278. Sign targeting eliminated TEMP7 and TEMP11. Arithmetic simplification eliminated TEMP16 and TEMP20.

tion `next-state` for the RLC circuit. The straight-lineness, compactness, and lack of structured values are the striking attributes of this code. No vestiges of the matrices produced or consumed during compilation, or of the control structures for doing so, or of the matrix inversion, or of the function retrievals and applications, appear in the final code. All address calculations vanish. There are many ramifications of the simplicity of the simulation program:

All uses can be planned. All values are explicit and can be explicitly planned. The compiler decides where in memory everything will reside. Heavily pipelined machines will be used very effectively. Also, state variables will be held in registers. Regular simulators cannot assign state variables to registers, which results in extra memory references and slower performance. (This drawback of normal simulators becomes worse and worse as processors get more and more registers.)

The program can be further optimized. Many opportunities for optimization arise because all values and their uses become explicit. Our system produces optimal code for the RLC circuit, an amazing result given our very inefficient prototype simulator. The most important optimizations are dead code elimination, sign targeting, arithmetic simplification, constant folding, and common subexpression elimination. It is virtually impossible for a human coding in assembler to create code as good as our system can create.

The program can be efficiently parallelized. As a corollary of being able to plan the use of all values, a compiler can also plan the efficient parallel use of the code for either tightly coupled or loosely coupled parallel systems. This ability is to be compared with the extremely hard task of producing good parallel code for a simulator itself, as we mentioned in the introduction. Because we parallelize the simulation program, not the simulator, we avoid many of the problems associated with attempting to parallelize simulators.

Component models are programmable with no overhead penalty. Today's simulators execute user-defined models less efficiently than they execute built-in models. The overhead comes from repeatedly looking up the model at simulation time. Compiled simulation performs the lookup once, at compile time, so that there is no cost difference between built-in and user-defined models.

Special purpose hardware. We can design extremely economical special purpose hardware for executing simulation programs. We anticipate that a machine that sustains

100 MFLOPs can be built for between $10,000 and $20,000. We believe we will keep five 20MFLOP ALUs busy with very little supporting hardware. Much of the hardware in existing "minisuper-computers" such as the Stellar and Ardent is support hardware. For example, in the Ardent, 30% of the gates are devoted to the scoreboard alone [Ardent]. In our proposed system we offload the function of such extra hardware into the compiler.

100 MFLOPS is a lot of power. If we pessimistically assume each timestep computation devotes 2000 floating point operations to each device, we can still comfortably simulate circuits which contain 50,000 devices.

4 General Compiled Simulation

In *general compiled simulation* we employ a program, called a *partial evaluator*, that accepts a simulator, a circuit, and a description of the data, and produces the simulation program. This is simpler and better than implementing a specific compiler for a specific simulator. General compiled simulation has many benefits:

Simulators become simpler to write. Because the circuit compiler does our optimizations for us, we need not waste our time trying to accelerate simulators. We will write coherent, simple, easily understood textbook style simulators. The importance of this should not be underestimated.

Component values can remain unspecified while compiling. When component values are unspecified during compilation the simulation program accepts the component values as parameters. A circuit isn't recompiled when these values change. This feature allows super-fast "*What-If?*" simulation. Also, Monte Carlo analysis and sensitivity analysis are speeded up dramatically. We see this super-amortization of the compilation time as a major strength of general compiled simulation.

Simulation programs can be recompiled for known component values. A Simulation program can be recompiled for given component values to constant fold those values into the program for even greater speedup. Therefore late binding of values need not have an effect on runtime speed. Note that once we have our partial evaluator, we get this added feature for free.

The Partial Evaluator

We feed the partial evaluator a function F, some real and some symbolic parameters, and we get out a new function G which accepts a parameter for each symbolic parameter to F. The function G returns a result as if we had called F on all its parameters at once.

The partial evaluator is an interpreter that handles symbolic values. Whenever the argument to a primitive function such as + is encountered, the partial evaluator adds an instruction to the program being created and returns a new symbolic value. Symbolic values carry information about the object they stand for. For example, a symbolic value can indicate it is a number, a string, or a vector or list of a certain length. The partial evaluator uses this information for several different tasks. For example, it performs compile-time type checking with this information. The partial evaluator also associates information with the symbolic values it creates. For example, when removing the head of a symbolic list that it knows contains five elements, it will create a symbolic list containing four elements. This methodology fully unrolls loops over lists whose lengths are known at compile time (such as a list containing the capacitors of a circuit).

For example, to create the compiled version of transient analysis upon circuit C with stepsize .1 seconds we type[2]

```
(define
  simulation-program
  (partially-evaluate transient-analysis c symbolic-value .1))
```

This call returns a simulation program that accepts component parameters and then performs a transient analysis.

The partial evaluator operates in three phases (Figure 5). The first phase builds a dataflow graph, the second phases optimizes the graph by applying graph transformations, the third phase allocates registers and produces code. For example, the first phase produces the dataflow graph shown in the top half of Figure 6, and the optimized version of this flowgraph appears in the bottom half. The code produced by the third pass was presented in Figure 4.

The first pass operates as described above, but symbolic values are augmented to be nodes of the dataflow graph. When a primitive such as + or * receives a symbolic value as an argument it creates a new node (symbolic value), sets up links in both directions between the incoming symbolic value and outgoing symbolic value, and then returns the new symbolic value.

[2]Of course, the system hides this ugliness from the user.

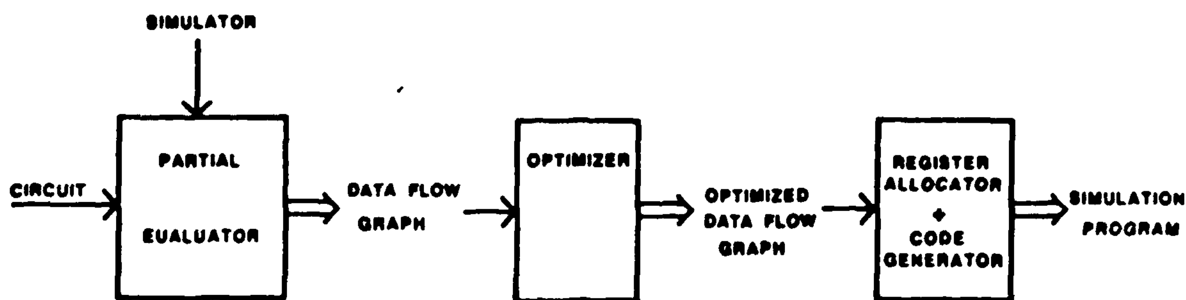


Figure 5: The partial evaluator. It has three phases. The first phase maps a simulator and a circuit into a dataflow graph. The second pass optimizes the dataflow graph. The third phase produces code. Fi gure 6 presents example outputs from the first two phases.

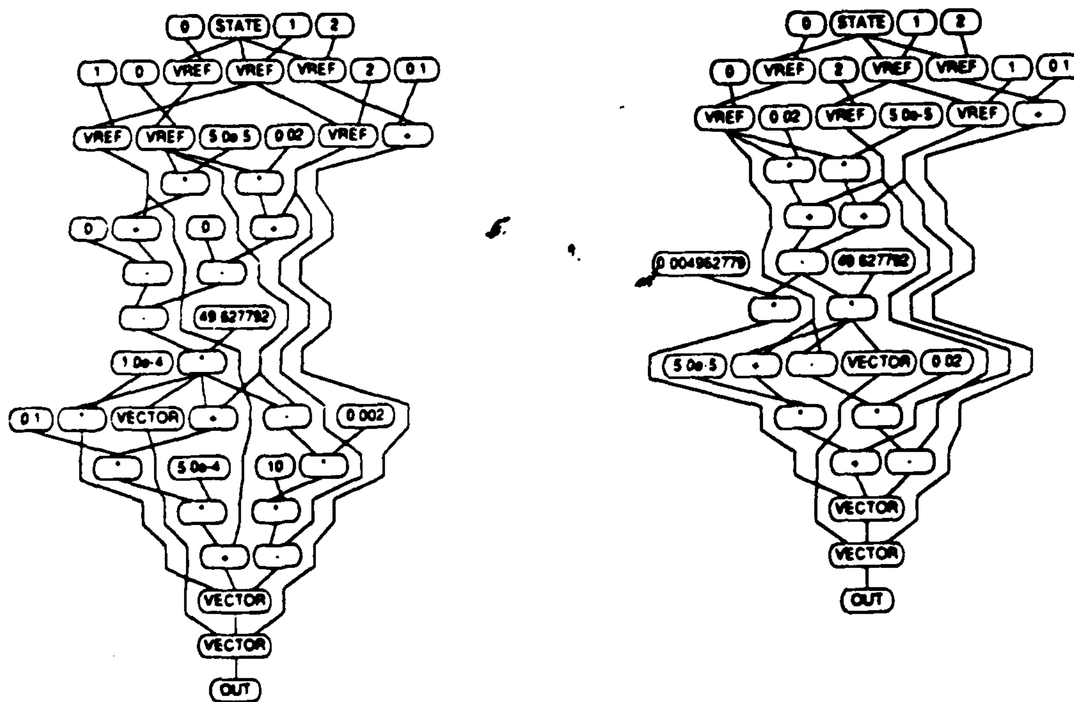


Figure 6: The dataflow graphs produced by the partial evaluator when compiling the RLC circuit. The graph on the left is the output of the first phase, the graph on the right is the output of the second phase.

#Components	Spice3	Simulation Program	Speedup
25	30s	2s	15
49	60s	5s	12
73	100s	7s	14
97	130s	10s	13
121	164s	12s	14

Figure 7: Speed of our system versus Spice3. All timings are reported in seconds. These experiments were performed on an HP9000/350 with 16MB. The circuits are all ladder networks. The circuits were simulated for 1000 timesteps. The numbers indicate that simulation programs run about 13 times faster than Spice3.

The second pass performs optimizations such as common subexpression elimination, dead code elimination, sign targeting, and constant folding. It implements these optimizations via graph transformations. Because it operates directly on the dataflow graph it can performs these optimizations very efficiently. Efficiency is very important: if the optimizer executes 5000 instructions to eliminate a given instruction, then, for the optimization to be worthwhile, the eliminated instruction would have had to been executed at least 5000 times.

The third pass performs register allocation and code generation. We are actively working on this part of the system. The code generator produces C code, which precludes any meaningful register allocation. We need to produce machine code to reap the full benefits of compiled simulation. We anticipate noticable improvements in our speedups once the code generator produces machine code. The most important issue for serial machines is better register allocation to minimize memory traffic.

These three passes are implemented in only 800 lines of code. Pass 1 is 400 lines, Pass 2 is 200 lines, and Pass 3 is 200 lines. These numbers will grow as we make the system run faster and as we move to more complex simulators. Nonetheless, we believe that outperforming Spice by at least a factor of five using only a 800 line compiler is a very interesting result.

5 Results

We have tested our system on circuits of up to 120 devices. The results are shown in Figure 7. To ensure that both simulators ran the same number of iterations we performed these experiments using a maximum step size much smaller than that needed for complete accuracy. Our system produced a simulation program written in C which was then compiled

with the same compiler used to compile Spice3. (A future version of our system will have an integral assembler.) For both simulators we counted the time to perform the simulation, not the time taken to read or write files.

Our results show our system running 13 times faster than Spice3. This speedup number is misleading and will change as our research progresses. First, we have not accounted for circuit compilation time in these figures. Because our partial evaluator is not coded for speed, and because it is running interpreted rather than compiled, circuit compilation time swamps simulation time. We have run experiments that indicate we can make the compilation time be equal to the time it takes to simulate ten timesteps.

Second, we aren't sure if our algorithms match Spice3's algorithms. In particular, Spice3 may be performing Newton-Raphson iterations. If so, it may be doing up to twice the necessary work. Once we start simulating non-linear devices the comparisons against Spice3 will be much fairer.

Third, because the partial evaluator emits C code rather than machine code, the speedups are not what they could be. We can achieve substantial reductions in memory accesses by using all available floating point registers. This is an important issue: floating point coprocessors are developing more and more registers.

For these reasons we have been conservatively stating that our system runs at least five times faster than Spice3.

There are two issues in scaling up our results to large non-linear circuits. The first issue is evaluating non-arithmetic functions such as exponentials and logarithms. These functions take longer to compute than arithmetic functions, so that as they become a larger fraction of the compute stream our speedups will decrease. We don't believe the decrease will be large. Offsetting this decrease is the time to solve the matrix, which dominates component evaluation time as circuit size grows.

The second issue is exponential blowup in compiled code size with respect to the number of nodes. Because the code for solving matrices is explicit, if there are A arithmetic operations in solving a matrix, then there will be at least A instructions in the simulation program. If we accept the experimental evidence that the complexity of matrix solution for electrical simulation is $N^{1.24}$ [Nagel] where N is the number of nodes, and pessimistically assume a five-fold space overhead factor, then, if one million instructions (4MB) were committed to solving the matrix, a circuit up to 88346 nodes could be handled. Since we don't anticipate simulating a circuit that large on a single processor, code blowup is not a problem.

6 Summary and Future Research

We have built a prototype general circuit compiler for linear circuits. It outperforms Spice3 by at least a factor of five. We are confident that we can extend the system handle Spice level simulation of large non-linear time-varying systems with equivalent speedups and no loss of accuracy.

Our novel contribution is the use of a partial evaluator to create simulation programs. This approach yields benefits on the compiler front, the simulator front, and the performance front. It wins on the compiler front because it has been very simple to create — a mere 800 lines of code produces exceptionally good results. It wins on the simulator front because we aren't tied down to a particular simulator. To speed up other types of simulations or to take advantage of special structures, such as those that arise in switched capacitive networks, we need only write a simulator and let the partial evaluator do the rest. It wins on the performance front because we are dramatically outperforming Spice3.

We have 6 tasks before us:

1. **Implementing simulation of non-linear devices.** This is straightforward and will yield more meaningful comparisons against Spice.

2. **Improving our code generation techniques.** The system outputs C code to be compiled, which incurs both an overhead time penalty and a performance penalty because C compilers are generally very stupid. We will gain performaince by having an integral assembler. We will lose portability, but simulation programs are so simple that porting the assembler will be a simple task.

3. **Improving the speed of the partial evaluator.** We need to get the time cost of compilation below the time it takes to simulate 10 timesteps.

4. **Producing code for parallel architectures.** We will investigate both tightly coupled and loosely coupled architectures.

5. **Designing economical hardware.** This hardware will execute our simulation programs at a sustained rate of 100 million floating operations per second. We want to keep five 20MH floating point chips busy with as little supporting hardware as possible.

6. **Generalizing our techniques to other scientific computations.** We believe our approach will work superbly whenever the structure of the data can be compiled away. We will have source code whose clarity and elegance is matched only by the speed of

the compiled system. Once we have Spice under our belt we will investigate using our techniques for the standard benchmarks of parallel scientific systems.

References and Bibliography

[Ardent] Lecture by Glen Miranker of Ardent at the EE380 seminar.

[ASTAP] W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Qassemzadeh, and T. R. Scott, "Algorithms for ASTAP - A Network Analysis Program," *IEEE Trans. on Circuit Theory*, Vol. CT-20, No. 6, November 1973, pp. 628-634.

[Barzilai] Z. Barzilai, J. L. Carter, B. K. rosen, and J. D. Rutledge, "HSS - A High-Speed Simulator," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6-4, July 1987, pp. 601-617.

[Bryant] Randal Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, Thomas Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proceedings of the 24th Design Automation Conference*, June 1987, Miami Beach, Florida, pps 9-16.

[Lewis] David Lewis, "A Programmable Hardware Accelerator for Compiled Electrical Level Simulation," in *Proceedings of the 25th Design Automation Conference*, June 1988, Anaheim California, pps, 172-177.

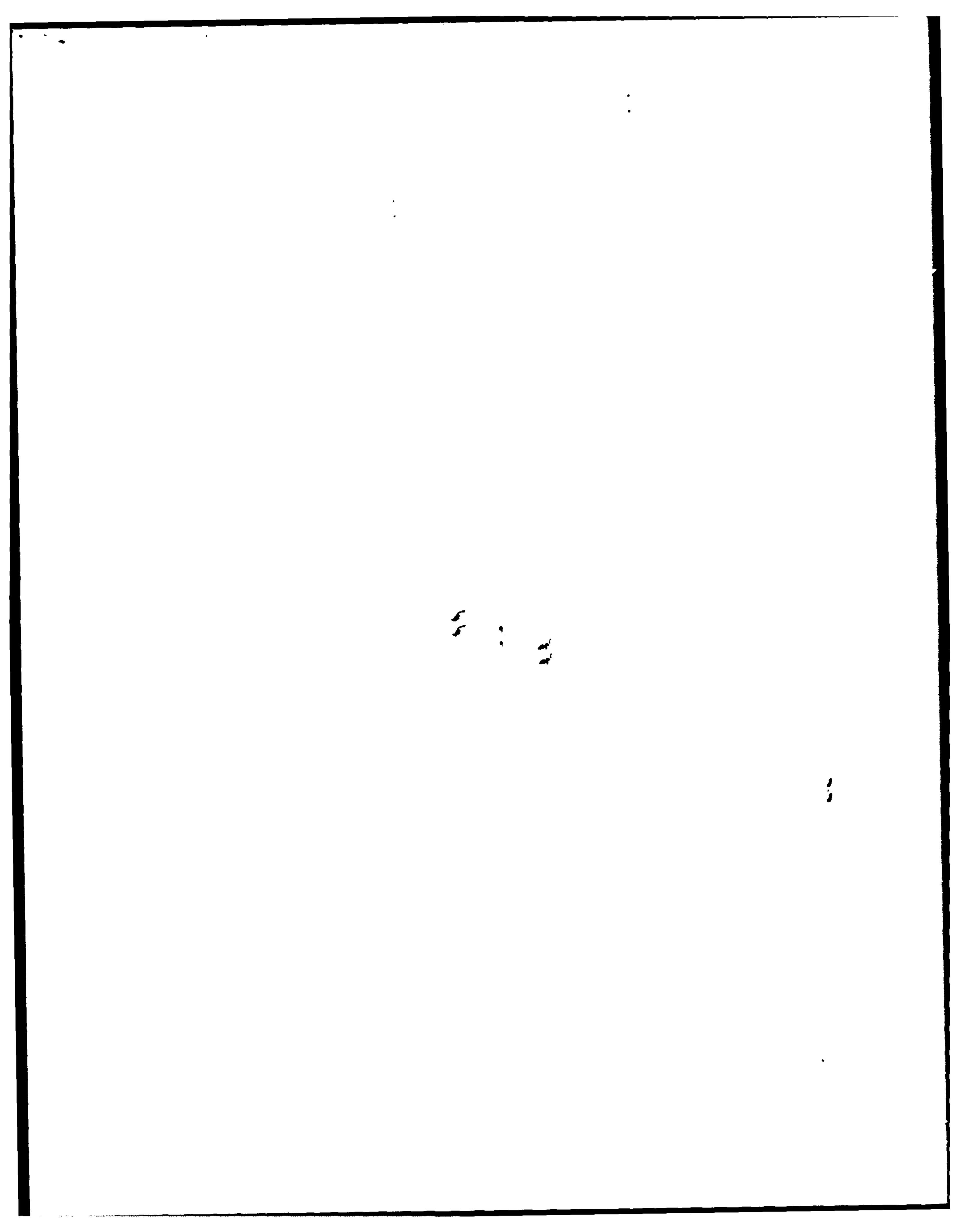
[Hansen] Craig Hansen, "Hardware Logic Simulation by Compilation," *Proceedings of the 25th Design Automation Conference*, June 1988, Anaheim California, pps, 712-715.

[Nagel] Laurence Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Electronics Research Laboratory *Report No. ERL-M520*, University of California, Berkeley, May 1975.

[Spice3] need this reference.

[Vladimirescu] Andrei Vladimirescu *et. al.*, "A Vector Hardware Accelerator with Circuit Simulation Emphasis," in *Proceedings of the 24th Design Automation Conference*, June 1987, Miami Beach, Florida, pps 89-94.

[White] Jacob K. White, *The Multirate Integration Properties of Waveform Relaxation, with Applications to Circuit Simulation and parallel Computation*, Ph.D. Dissertation, University of California, Berkeley, Electronics Research laboratory, November 18, 1985.



Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results

Wolf-Dietrich Weber and **Anoop Gupta**
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Working Draft - NOT for general distribution

November 16, 1988

Abstract

A fundamental problem that any scalable multiprocessor must address is the ability to tolerate high latency memory operations. This paper explores the extent to which multiple hardware contexts per processor can help to mitigate the negative effects of high latency. In particular, we evaluate the performance of a directory-based cache coherent multiprocessor using memory reference traces obtained from three parallel applications. We explore the case where there are a small fixed number (2-4) of hardware contexts per processor and the context switch overhead is low. In contrast to previously proposed approaches, we also use a very simple context-switch criterion, namely a cache miss or a write-hit to shared data. Our results show that the effectiveness of multiple contexts depends on the nature of the applications, the context switch overhead, and the inherent latency of the machine architecture. Given reasonably low overhead hardware context switches, we show that two or four contexts can achieve substantial performance gains over a single context. For one application, the processor utilization increased by about 65% with two contexts and by about 100% with four contexts.

Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results

Wolf-Dietrich Weber and **Anoop Gupta**
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

November 16, 1988

Abstract

A fundamental problem that any scalable multiprocessor must address is the ability to tolerate high latency memory operations. This paper explores the extent to which multiple hardware contexts per processor can help to mitigate the negative effects of high latency. In particular, we evaluate the performance of a directory-based cache coherent multiprocessor using memory reference traces obtained from three parallel applications. We explore the case where there are a small fixed number (2-4) of hardware contexts per processor and the context switch overhead is low. In contrast to previously proposed approaches, we also use a very simple context-switch criterion, namely a cache miss or a write-hit to shared data. Our results show that the effectiveness of multiple contexts depends on the nature of the applications, the context switch overhead, and the inherent latency of the machine architecture. Given reasonably low overhead hardware context switches, we show that two or four contexts can achieve substantial performance gains over a single context. For one application the processor utilization increased by about 65% with two contexts and by about 100% with four contexts.

1 Introduction

As shared-memory multiprocessors are scaled (the number of processors is increased), there will invariably be an increase in the latency of memory operations. While local memory references need not have higher latency, remote memory operations will encounter higher latency because of the larger physical size of the machine, if not for any other reason. Consequently, there will always be times when a processor sits idle, waiting for some remote operation to complete [2,11]. If more than one context resides on each processor, and context switch overhead is low, this idle time can be used by additional contexts. Typically each context corresponds to a process from one parallel program.

In this paper, we evaluate the utility of multiple contexts per processor for a directory-based cache coherent multiprocessor [1]. While the idea of using multiple hardware contexts per processor is itself not new, we believe our scheme is simpler to implement than other proposals [4,8,11,19,21] (discussed in Section 5). In our scheme, each processor contains a small fixed number (2-4) of hardware contexts with independent register sets to enable short context switch times. We also use a very simple context switch criterion, which is to switch contexts on a cache miss or on a write-hit to read-shared data or when a watchdog counter of 1000 expires.[1] This simple scheme helps keep context switch overhead low, because the decision to switch or not can be made in a single cycle.

Our multiple context scheme is evaluated using multiprocessor memory-reference traces obtained from three applications [13,16,20]. The results indicate that multiple contexts can achieve substantial gains in processor utilization. In some cases processor utilization is increased by 65% with two contexts and by 100% with four contexts.

The rest of the paper is organized as follows. The next section presents the architecture and simulator used in this study. We also introduce the applications and the method employed to gather the reference traces. Section 3 gives general results for the three applications. After that we present a number of issues concerning multiple contexts. This section also gives the results of the simulations. Finally, we have the related work, discussion and conclusion sections.

2 Architectural Assumptions and Simulation Environment

In this section, we discuss the architectural assumptions that we make and describe the simulation environment that we used to obtain our results. We also describe the applications used in this study and the performance metric employed to evaluate the multiple context scheme.

2.1 Base Architecture and Simulator

Figure 1 shows the basic architecture that we assume in this paper. The architecture consists of several nodes linked together by an interconnection network. Each node has a processor, a physical cache, and its share of the global memory. It is connected to the network through the directory (DIR) and network interface (N.I.). The processors may have one or more contexts. The caches are kept consistent using a directory-based cache coherence protocol as discussed in [1]. We study the performance as a function of several parameters such as the number of contexts, the context switch overhead, the latency of the network, and so on. Performance results as a function of the above parameters are given in Section 4.

[1] The watchdog counter is introduced to prevent one context from hogging a particular processor. This ensures that no context runs for longer than 1000 cycles at a time, preventing starvation and deadlocks.

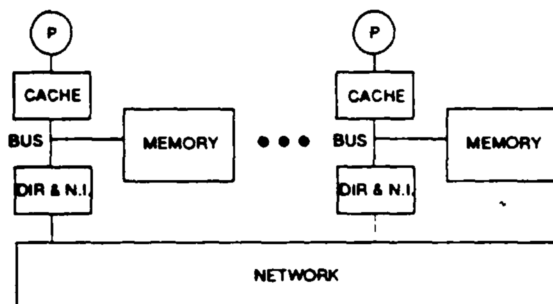


Figure 1: Architectural model

We use a trace-driven simulator, written by Truman Joe at Stanford, that emulates the above architecture to evaluate the effectiveness of multiple contexts. In the single context per processor case, the simulator works as follows. Before starting the simulation, we first divide the interleaved reference stream generated by the tracing program into separate streams for individual processors. Then, one reference stream is assigned to each of the processors. At every simulated clock cycle, each active processor reads the next reference from its associated reference stream. If the reference hits in the cache[2] the processor remains active and will issue another reference from the stream on the next clock tick. However, if it misses or a write to read-shared data occurs, it context switches. The cache sends a request over the network to fetch the missing line and/or update the state of the other caches in the system. During the period of time that the cache request is waiting to be satisfied, the processor remains in a suspended state and does not generate any more references.

In case of multiple contexts per processor, we have multiple memory reference streams associated with each processor — one for each context. At any given time only one of these contexts is active and the memory references come from that stream. However, when the active context enters the suspended state due to a cache miss or a write hit on read-shared data, a context switch occurs. The processor stays idle for the time required to perform the context switch. After that, memory references are issued from the newly activated context. If more than one context is ready when the active context blocks, a round-robin scheduling scheme decides which context is to be activated next.

The simulator that we use is quite detailed in that it models contention for the memory modules, for the bus on which the memory modules reside, for the directory associated with each node, and for the interconnection network. It is also possible to vary the delays associated with each of the above modules. We note that the interconnection network assumed in our simulations is a crossbar switch, but it could be any point-to-point network (e.g., grid [18], butterfly [3], omega [17]) depending on the number of processors we wished to interconnect. For the default parameters that we used (shown in Table 1), a remote read takes 27 cycles and a remote write takes 19 cycles with no contention. The local operations take 16 and 13 cycles respectively. With contention these numbers can grow to as large as 100 cycles in our simulations.

The simulator is driven by multiprocessor memory reference traces. Since the traces include 16 reference streams, we are limited to four processors if we wish to explore four contexts per processor. For runs with fewer than four contexts, only some of the reference streams were used. We model the scaling of the machine architecture to a larger number of processors by increasing the latency in the underlying network (see Section 4.3). We also vary the context switch overhead and the number of contexts per processor. Section 4 will present the issues involved and the results obtained.

Operation	Time
Memory Latency	6 cycles
Bus Transfer	4 cycles
Switch Latency	2 cycles
Switch Transfer	4 cycles
Directory Lookup	2 cycles

Table 1: Default Parameters for Simulator

One inaccuracy in our simulator is that we assume an infinite cache for each processor.[3] Thus, we do not model the interference in the caches when there are multiple contexts per processor. It is not clear, though, whether the sharing of caches is an advantage or a disadvantage. If the caches are small, interference might be a serious problem. With fairly large caches, however, the pre-fetch achieved by contexts working on the same shared data could actually be beneficial.[4] The caches in the architecture presented here are expected to be large as they serve as the main source of remote code and data.

2.2 Traces and Applications

The multiprocessor traces used in our simulations were gathered on a VAX 8350, using a combined hardware/software scheme [5]. Basically, the tracing works as follows. We spawn as many processes as the application desires under the control of a master process. The master process then single steps the application processes in a round-robin manner. After each step, it records all references made by the application processes. For each reference, the number of the processor producing it, the address of the reference and its type (read/write/ifetch) are recorded. The traces that we use correspond to 16-processor runs.

The traces used were obtained from three applications: LocusRoute, MP3D and P-Thor. LocusRoute [16,17] is a standard cell global router. While the tasks spawned by it are quite coarse in granularity (each may execute around 100,000 instructions), its central data structure (a global cost array) is shared at a fine granularity. MP3D [13] is a 3-dimensional particle simulator that determines the shock waves generated by a body flying at high speed in the upper atmosphere. It uses distributed loops for parallelization (each loop executes around 250 instructions) and it is a typical example of parallel scientific code. P-Thor [20] is a parallel logic simulator that uses the Chandy-Misra distributed simulation algorithm. Each parallel subtask (a component evaluation) in P-Thor takes about 300 instructions to execute.

[2] For writes, the location has to be owned in addition to being present in the cache.

[3] We are working on an a new version of the simulator that will remove this restriction.

[4] Note that in our execution model, several processes from the *same* application are using the multiple contexts. Thus the amount of shared data can be significant.

2.3 Performance Measure

The main figure of merit used in evaluating multiple contexts in this paper is *processor efficiency*. This is defined as the number of cycles spent doing useful work over the total number of cycles. Of course, the maximum is one reference per processor per cycle for 100% efficiency. The more time the processors spend idle, waiting for remote reads and writes, the lower the overall processor efficiency. In our simulations, we ran the system for a total of 500,000 clock cycles, and then counted the number of memory references consumed from the traces to get the efficiency.

3 General Results

In this section we present some general results obtained with the simulator. These results give an overall idea of the differences in behavior of the three applications. They also show the effect of increasing the switch latency on the read and write latencies seen by the processors. The numbers are for a 4-processor system with one context per processor. The tables below give data about the run lengths and latencies for the three applications. Run length is defined as the number of simulator cycles between each cache miss.[2] Read and write latencies are the number of cycles required to satisfy the cache miss.

Results for switch latencies of 2 and 16 cycles are presented. A switch latency of only two cycles is close to the minimum that can be achieved with any type of network. The switch latency of 16 represents the latencies that might be expected in a larger multiprocessor with many more nodes.

	Run Length		Read Ltncy		Write Ltncy	
Application	Avg	Med	Avg	Med	Avg	Med
MP3D	20	14	25	27	19	19
P-Thor	61	18	24	27	17	19
LocusRoute	107	45	24	27	17	19

Table 2: General application results with switch latency of 2 cycles

	Run Length		Read Ltncy		Write Ltncy	
Application	Avg	Med	Avg	Med	Avg	Med
MP3D	18	14	51	55	33	33
P-Thor	58	17	48	55	31	33
LocusRoute	100	44	54	55	37	30

Table 3: General application results with switch latency of 16 cycles

Both average and median values are given to convey more information concerning the distribution of the run-lengths and latencies. Median values are more representative in characterizing the typical run-length. In LocusRoute, for example, due to a few very long runs the averages are high even though the median values are much lower.

MP3D has the shortest run-length and longest latencies. There is a lot of global data traffic in MP3D and this leads to frequent misses, i.e. short run lengths. LocusRoute, on the other hand has very long run-lengths. The large size of the tasks and their relative independence allows for large portions of code that execute out of the cache without any misses. The latencies are close to the minimum expected for this architecture. P-Thor is somewhere in between the other two applications.

As the switch latency increases, the read and write latencies grow as well. Reads are affected more because they require a two-way transaction and so the higher latency is incurred twice. Run lengths should be unaffected by the increased latency, but in fact we do see a slight decrease in run lengths as the switch latency increases. This is probably due to a cold-start effect of the caches. Run-lengths near the beginning of the reference streams are shorter on average, because more cache misses are incurred.

4 Issues and Results

We wish to explore several questions concerning the performance of multiple contexts:

- How many contexts are required to achieve good processor utilization?
- How does the context switch overhead affect the performance?
- What is the effect of increasing the switch latency?
- When to switch contexts?
- How much does the performance vary with application?

This section explores all of these issues and presents results. We show graphs of processor efficiency. In each graph, we are plotting the number of active cycles over the total number of cycles against the switch latency of the architecture. We show efficiencies for one, two and four contexts. Different context switch overheads are presented on different graphs. Figures 2–4 show results for MP3D, Figures 5–7 give results for P-Thor and Figures 8–10 show results for LocusRoute.

4.1 Number of Contexts

Depending on the single context processor efficiency, it may or may not be worthwhile to use two, four or more contexts. Note that the single-processor efficiency is basically a function of the cache miss rate and the read and write latency for the architecture. For LocusRoute (Figures 8–10) the processor efficiency is already very high (about 90%) with a single context and little performance can be gained by adding more contexts. As a matter of fact, if the context switch overhead is high, four contexts do worse than one (Figure 10). MP3D on the other hand (Figure 2), has single context performance near 50% and achieves substantial gains with more contexts (efficiency is 77% with 2, 94% with 4).

As expected, the graphs show diminishing marginal returns as the number of contexts is increased (see Figure 5 for example). In every case going from one to two contexts yields a greater benefit than going from two to four contexts. A small number of contexts is also preferable because it allows simpler

[2] Both here and in the rest of the paper, by *cache miss* we actually mean references that can not be satisfied by the cache alone and need to access the memory, or the network, or both. These include regular cache misses but also write-hits to read-shared data. In the latter case, the network needs to be accessed to invalidate that location from other caches and to gain ownership of that cache line.

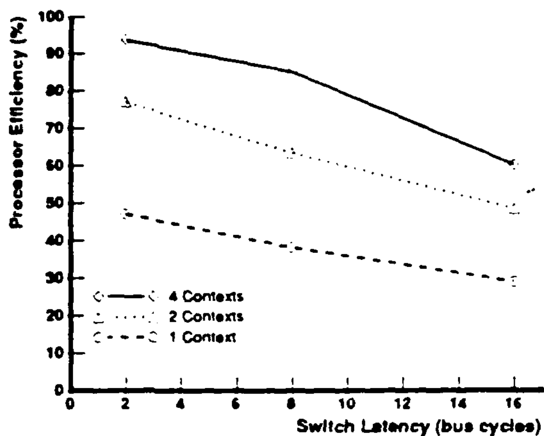


Figure 2: MP3D: Context Switch Overhead 1 Cycle

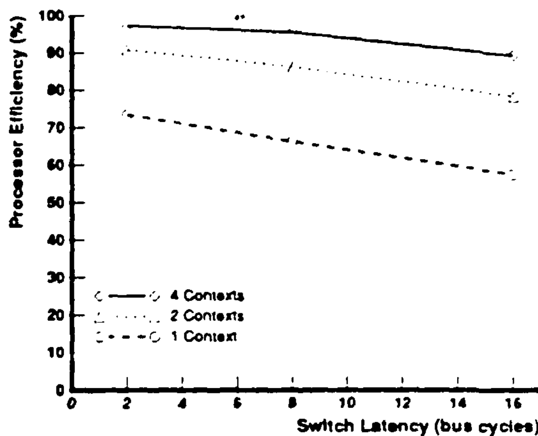


Figure 5: P-Thor: Context Switch Overhead 1 Cycle

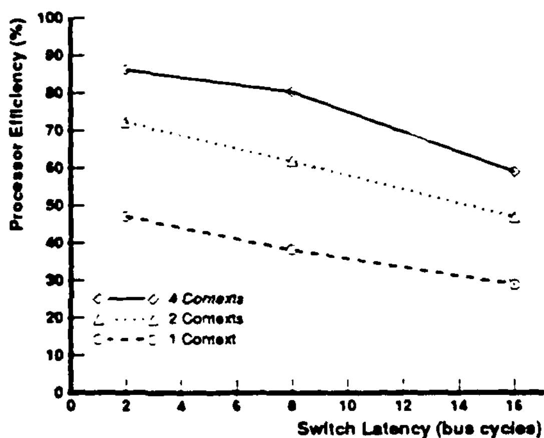


Figure 3: MP3D: Context Switch Overhead 4 Cycles

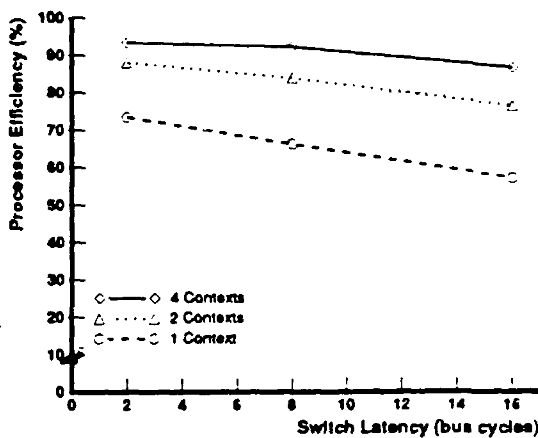


Figure 6: P-Thor: Context Switch Overhead 4 Cycles

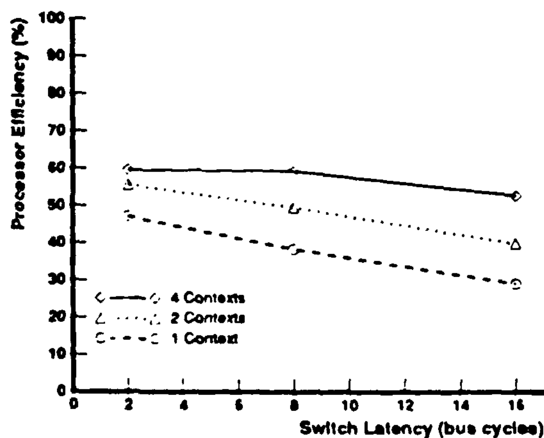


Figure 4: MP3D: Context Switch Overhead 16 Cycles

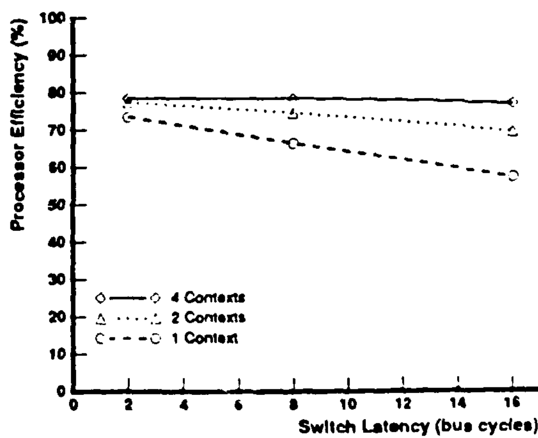


Figure 7: P-Thor: Context Switch Overhead 16 Cycles

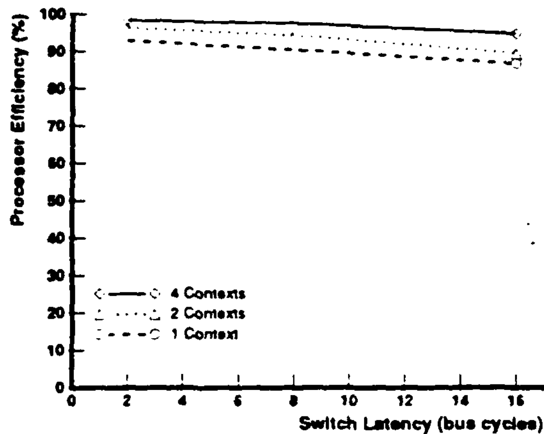


Figure 8: LocusRoute: Context Switch Overhead 1 Cycle

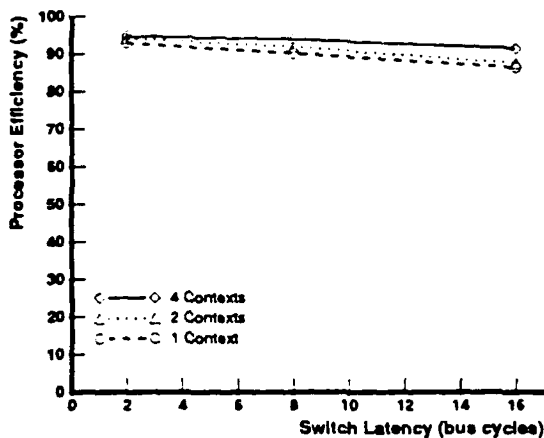


Figure 9: LocusRoute: Context Switch Overhead 4 Cycles

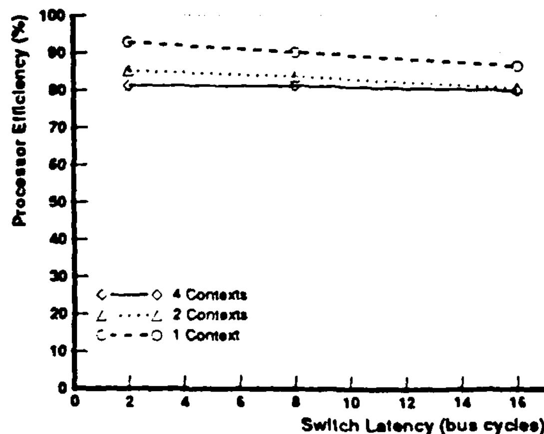


Figure 10: LocusRoute: Context Switch Overhead 16 Cycles

hardware. With a larger number of contexts, a penalty in the cycle time of the processor or an increase in context switch overhead may be inevitable. Also, a large number of contexts requires a large number of processes. Many applications may not be able to support such a large number of processes.

4.2 Context Switch Overhead

The context switch overhead depends on the number of contexts kept in hardware, the amount of state kept for each context, and the amount of hardware dedicated to context switching. We explore context switch overheads of 1, 4 and 16 cycles. A single cycle overhead can be achieved by keeping multiple copies of the pipeline registers and being able to swap in the whole state in a single cycle.[6] If the pipeline has to be drained and filled, a 4-cycle overhead is reasonable. Both of these options require multiple register banks, one for each context. If we want to load and store the registers to some fast local memory, we have to allow at least 16 cycles. It is clear that the hardware is more complex if we require the context switch to be faster. Of course, beyond some overhead value, multiple contexts do not help any more, since a long latency operation will complete before the context switch is achieved.

As expected, the results show that the effect of increasing the context switch overhead reduces the benefit achieved by having multiple contexts. Note that the single context graph line is identical for various context switch overheads (see Figures 2–4 for example), since there is no context switching in that case. When the context switch overhead is 16, none of the programs are gaining much processor efficiency with increased contexts. MP3D achieves a 12% increase in efficiency with 4 contexts (Figure 4). P-Thor gains only 5% (Figure 7) and LocusRoute actually looses 12% (Figure 10). For multiple contexts to be useful, the context switch overhead will have to be kept low, preferably on the order of a few cycles.

4.3 Latency

The amount of latency incurred in remote operations is important for the effectiveness of processors with multiple contexts. With very low latencies, context switch overhead may be too large to allow multiple contexts to achieve any performance gain. As the latency increases, the single context processors do increasingly poorly because more and more processor time is spent idle. This is where multiple contexts can help. As seen in Figures 5–7, the relative value of multiple contexts increases as the latency increases. In other words, a processor with multiple contexts will suffer less efficiency degradation due to high latencies than a single context processor.

One reason for varying switch latency in our evaluation of multiple contexts is to explore different types of architectures. A grid network, for example, is expected to have a much larger latency than a crossbar switch. At the same time the higher latencies can correspond to larger multiprocessors. As more processors are added to a parallel machine, the latencies increase due to deeper networks or more complex switches. Larger latencies present a greater opportunity for multiple contexts, because the single context efficiency is lower. At the same time we note that it is still possible to achieve very high efficiencies with just a few contexts. For example, with

[6]Alternatively multiplexors could be used to switch between multiple pipeline state copies.

a switch latency of 16 cycles, latencies are on the order of 50 and 30 cycles for reads and writes respectively (see Section 3). A network large enough to have this high a latency could well support several hundred processors. Yet processor efficiencies stay high for this latency (60% for MP3D, 89% for P-Thor and 94% for LocusRoute). The point is that even as multiprocessors grow and latencies increase, processors with just a few contexts achieve very good utilization.

4.4 When to Switch Contexts

Ideally, one would like to switch contexts whenever the context switch overhead is less than the latency of the operation being performed. Of course external operations may take longer or shorter depending on the congestion in the machine, and there is no easy way to predict how long a given operation will take. We thus choose the easiest context switch criterion: switch on any operation that requires a main memory access, either in the same cluster or remotely. Switching only on *remote* operations requires extra hardware, but is a feasible alternative if context switch overhead is relatively high. If a context switch takes 16 cycles, and local operations also take on the order of 16 cycles to complete, it does not make sense to initiate a context switch on every local operation.

Two of the applications had frequent memory accesses, but LocusRoute processes had long streaks of executing out of the cache. In order to prevent one context from hogging a particular processor we introduce a watchdog counter that pre-empts the current context after 1000 cycles. This ensures that no context runs for longer than 1000 cycles at a time, thus allowing all contexts on a particular processor to make progress.

4.5 Applications

The three applications exhibited very different behavior. LocusRoute and P-Thor have relatively little global traffic, whereas MP3D has a lot. While 1.8% of LocusRoute instructions cause references to shared data, this number is close to 12% for MP3D. This explains why the run-lengths presented in Section 3 are so different for the three applications. At the same time LocusRoute has very good caching behavior and very little interference between processes. Thus LocusRoute achieves very high efficiencies (around 90%), even with single context processors (see Figures 8–10). Very little can be gained by adding extra contexts.

P-Thor achieves 50-70% utilization with single contexts (see Figures 5–7). This can be boosted effectively by adding more contexts. Not only is efficiency increased as more contexts are added, but the processors also become more immune to the effect of high latency operations. This is seen by the spreading of the curves as the latency increases.

MP3D has a large amount of global traffic. When the switch latency increases, the switch becomes the bottleneck and it limits the gains achieved by multiple contexts. While some performance gain is achieved, the relative benefit of multiple contexts is greater for lower latencies. Note how the different context lines converge as the switch latency increases in Figures 2 and 3.

5 Related Work

The idea of multiple hardware contexts per processor in itself is not new. In this section we discuss how our approach differs from earlier proposals and present some advantages and disadvantages. We begin with the Alto personal computer from Xerox [21] which provided multiple hardware microcode-level contexts, allowing the CPU to be shared between the instruction set interpreter and the I/O devices. The contexts were statically assigned to devices and were not available to general user processes. The aim of the multiple contexts was to make the power of the processor readily available for time critical I/O processing, a task that is frequently delegated to separate processors in more recent designs. Unlike our motivation, the issue was not to hide memory latency from a very fast processor.

The HEP multiprocessor from Denelcor [19] also provided multiple hardware contexts per processor. Unlike the Alto, the contexts were available to arbitrary user processes. The processes shared a large set of registers and on each cycle an instruction from a different process was executed. A minimum of 8 *active* processes (those processes that are not waiting for a memory reference to complete) were needed to keep the execution pipeline full. The HEP machine tolerated memory latency well, but its main drawback was that a single process could get at most 1/8 of the pipelined processor. In order to keep the pipeline full, a large number of processes were needed. This is in stark contrast to modern pipelined processors [6,14] where a single process almost fully utilizes the pipelined processor. Now the HEP scheme would *not* be a problem if all applications could be split into an arbitrarily large number of processes. However, this is often not possible in practice as there may not be enough intrinsic parallelism in the application [7], or because doing so greatly increases the amount of overhead.

More recently, Iannucci [11] has proposed using multiple contexts for his hybrid data-flow/von Neumann machine. Each processor consists of a hardware queue of enabled *continuations*. The continuations are very small in size (containing just the program counter and the frame base-register), and the hardware can switch between them in a single cycle. However, to make this single cycle switch possible, processor registers are not saved on a context switch. Consequently, the software is structured so that it does not rely on registers being valid between potential context switch points. The switch points are synchronizing references, where a read to a location tagged *empty* results in that continuation being suspended. In our view, the disadvantages of Iannucci's approach are the following. First, processes can not make full use of the register sets, given that the run-lengths (the number of instructions executed between switch points) are very small [11] and registers are not preserved in between. We believe that extensive use of registers is absolutely critical to the performance of modern processors [6]. Second, a processor that supports a large number of continuations (contexts) in hardware, keeps track of which ones are enabled and uses a complex criterion for deciding which continuation to issue the next instruction from [12], is very complicated. We believe such a processor will have a significantly more complex pipeline and much larger area than a simple RISC processor. Consequently, the cycle time of such a machine would be slower than that of modern RISC processors. Thus the hybrid machine has to make up the large factor that it loses over conventional microprocessors, before it becomes competitive. On the other hand, the scheme that we propose does not lose

anything over modern RISC processors. In fact, it is possible to take multiple commercially available RISC processor chips (e.g., Motorola 88000 processor and cache chips) and connect them so as to simulate multiple contexts.

We now consider the MASA architecture proposed by Bert Halstead [8]. In this architecture each processor has a fixed number of hardware *task frames*. Each task frame is capable of storing a complete process context and consists of a set of auxiliary registers (like the program counter) and a set of general purpose registers. Since the number of processes may exceed the number of task frames, the process contexts are allowed to overflow into memory.[7] On each cycle, a context in the *enabled* or *ready* state may issue an instruction. However, once a process issues an instruction, it can not issue another instruction until the previous instruction has completed. Thus, in its current form, a process on MASA can get only 1/4 (inverse of pipeline depth) of the pipelined processor's performance. As discussed above for HEP, this is a major drawback. Halstead and group recognize it [8] and are exploring ways to remove this restriction.

We now discuss a more subtle but fundamental difference between the Iannucci and Halstead schemes and our scheme. In our scheme, the sole purpose of the multiple hardware contexts is to mitigate the negative effects of memory latency. The number of hardware contexts needed for a particular machine is fixed and depends mainly on the expected cache hit ratio and the memory latency for that architecture. In the Iannucci and Halstead schemes, the context mechanism is instead made to serve two purposes at the same time. It is used to mask memory latency as in our scheme, but it is also used as a hardware task queue. Thus when a parallel subtask is created, it manifests itself as a new context that is then managed and scheduled by the hardware. Since the number of parallel subtasks can be arbitrarily large, mechanisms are needed and provided to handle overflow of contexts. Also, the number of contexts that are needed is large. In our scheme, the issue of subtask management is completely separated and is handled in software. This permits great flexibility, including the possibility to schedule tasks in a manner similar to the Iannucci and Halstead proposals, if a particular application so warrants.[8] Thus instead of using full/empty bits and hardware queuing in I-structure memory [10], we may simulate full/empty bits in software and switch to a different subtask if a piece of data is not ready. It is not obvious which scheme works better. We will be able to tell only when such machines actually get built.

6 Discussion

This section contains the discussion of several topics that relate to the evaluation of multiple contexts as presented in this paper.

One question that we must ask is, what are the real advantages of having multiple contexts? Since processors are cheap, why not simply have a larger number of processors in the multiprocessor? The fallacy in this argument is that, while CPU chips (e.g., MC68030 chips) are relatively cheap, a fast processor is not — a fast processor nowadays has a large amount of cache built out of expensive and fast SRAMs; in addition, there are expensive functional units such as floating point ALUs. Furthermore, each new processor needs an extra port to the network, or to the bus that it is placed on. The extra port increases the depth of the network, or the loading on the bus, thus increasing the latency. Several contexts per processor can share these expensive resources, thus making more efficient use of them.

Another question that arises is how the multiple contexts should be implemented. The multiple contexts do not necessarily have to be implemented on a single chip. In the case where the size of each processing node is small, on the order of a few chips [9], we need to have several contexts on a single chip using duplicated register sets. However, having to design a special processor for a given architecture makes that architecture less practical. So for larger processing nodes, for example where each processor occupies a whole board, it may be quite feasible to use separate processor chips for the different contexts. While simplifying the hardware design effort, this approach duplicates not just the register set but all of the data path and control as well.[9]

There are some software issues to be resolved. In particular, how do you choose which processes to put on a single processor? Since the progress of contexts on any one processor is mutually exclusive, the correct placement of processes on processors may be important. If a given program section requires several contexts to be active in order to make progress, it is best to place these on separate processors.

7 Conclusions

In scalable multiprocessor architectures, processors with a small fixed number of contexts can achieve substantially greater efficiencies than single context processors. In some cases efficiencies increased 65% with two contexts and 100% with four contexts. Best improvements are found in architectures with high latency operations and low context switch overheads. Such high latency operations are to be expected in large-scale multiprocessors. Low context switch overheads can be achieved by having a small fixed number of contexts in hardware and by using a *simple switch criterion: the cache miss.*

One important difference between our context switch scheme and those proposed in [8,11,19] is that in our scheme the context switch mechanism is separated from the subtask management mechanism. This makes for simpler and faster hardware and allows greater flexibility and application-dependent performance tuning.

We are currently working on more detailed simulations, including the effects of finite caches and cache contention when a miss is satisfied from memory. We are also looking further into the issues and details of implementing our multiple context scheme.

8 Acknowledgements

We would like to thank Truman Joe for allowing us to use his simulator and for helping us understand it and modify it. We also wish to thank Richard Sites at Digital Equipment Corporation, Hudson MA, for supporting Wolf-Dietrich Weber.

[7] Such overflow and underflow operations are quite expensive, and care must be taken to minimize them.

[8] We would normally expect there to be some sort of a distributed task queue to handle the scheduling of subtasks.

[9] An alternative to this scheme which uses multiple CPU chips for contexts is to let all these chips be active all the time, sharing and stalling on the cache as they proceed. We have, as yet, not done any performance evaluation for such a scheme.

Anoop Gupta is supported by DARPA contract N00014-87-K-0828 and by a faculty award from Digital Equipment Corporation. Thanks also to Helen Davis, Margaret Martonosi, Jonathan Rose, Rich Simoni, Larry Soule and Mike Smith for reviewing early versions of this paper.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *15th International Symposium on Computer Architecture*, 1988.

[2] Arvind and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. In *15th International Symposium on Computer Architecture*, pages 426–436, 1983.

[3] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance Measurements on a 128-node Butterfly Parallel Processor. In *Intl. Conf. on Parallel Processing*, pages 531–540, 1985.

[4] William J. Dally et al. Architecture of a Message-Driven Processor. *The 14th Annual International Symposium on Computer Architecture*, 189–196, June 1987.

[5] Stephen R. Goldschmidt. Simulating Multiprocessor Memory Traces. December 1987. EE390 Report, Stanford University.

[6] T. Gross, J. Hennessy, S. Przybylski, and C. Rowen. Measurement and Evaluation of the MIPS Architecture and Processor. *ACM TOCS*, 6, August 1988.

[7] Anoop Gupta et al. Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis. *International Journal of Parallel Programming*, 17, 1988.

[8] R. H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *15th International Symposium on Computer Architecture*, pages 443–451, 1988.

[9] J. P. Hayes et al. A Microprocessor Based Hybrid Supercomputer. *IEEE Micro*, 6, October 1986.

[10] S. K. Heller. *An I-Structure Memory Controller (ISMC)*. Technical Report, Massachusetts Institute of Technology, June 1983.

[11] R. A. Iannucci. Toward a Dataflow / von Neumann Hybrid Architecture. In *15th International Symposium on Computer Architecture*, pages 131–140, 1988.

[12] Robert A. Iannucci. *A Dataflow / von Neumann Hybrid Architecture*. PhD thesis, Massachusetts Institute of Technology, 1988.

[13] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a Particle Simulation Method for Hypersonic Rarified Flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[14] D. Patterson. Reduced Instruction Set Computers. *Comm. ACM*, 28, January 1985.

[15] G.F. Pfister, W.C. Brantley, et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *International Conference on Parallel Processing*, IEEE, 1985.

[16] Jonathan Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Design Automation Conference*, pages 189–195, June 1988.

[17] Jonathan Rose. The Parallel Decomposition and Implementation of an Integrated Circuit Global Router. In *Proc ACM SIGPLAN PPEALS*, pages 138–145, July 1988.

[18] Charles L. Seitz, William C. Athas, Charles M. Flaig, Alain J. Martin, Jakov Seizovic, Craig S. Steele, and Wen-King Su. The Architecture and Programming of the Ametek Series 2010 Multicomputer. In *Hypercube Concurrent Computers and Applications*, 1988.

[19] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE*, pages 241–248, 1981.

[20] Larry Soule and Tom Blank. Parallel Logic Simulation on General Purpose Machines. In *Design Automation Conference*, pages 166–171, June 1988.

[21] C. P. Thacker, E. M. McCreight, et al. Alto: A Personal Computer. In C. Gordon Bell Daniel P. Siewiorek and Allen Newell, editors, *Computer Structures: Principles and Examples*, pages 549–572, McGraw-Hill, 1982.

HARDWARE C - A LANGUAGE FOR HARDWARE DESIGN

David C. Ku and Giovanni De Micheli

Technical Report No. CSL-TR-88-362

August 1988

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Abstract

High-level synthesis is the transformation from a behavioral level specification of hardware to a register transfer level description, which may be mapped to a VLSI implementation. The success of high-level synthesis systems is heavily dependent on how effectively the behavioral language captures the *ideas* of the designer in a simple and understandable way. This paper describes **HardwareC**, a hardware description language that is based on the C programming language, extended with notions of concurrent processes, message passing, explicit instantiation of procedures, and templates. The language is used by the HERCULES High-Level Synthesis System.

Key Words and Phrases: High-level synthesis, hardware description languages.

Copyright © 1988

by

David C. Ku and Giovanni De Micheli

HardwareC - A Language for Hardware Design

David C. Ku Giovanni De Micheli

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

1 Introduction

High-Level synthesis is the transformation from a behavioral level specification of hardware to a register transfer level description, which may then be mapped to a VLSI implementation. The success of high-level synthesis systems is heavily dependent on how effectively the behavioral language captures the ideas of the designer in a simple and understandable way. This paper describes *HardwareC*, a behavioral hardware description language that is used by the HERCULES High-Level Synthesis system [1,2].

The input to HERCULES consists of two sets of specifications - a description of the *functionality* and a set of *design constraints*. The functionality is described in a C-based language extended for hardware description called *HardwareC*. The *design constraints* specify the timing and resource limitations that are imposed on a given design. The HardwareC description is parsed and translated into a parse tree abstraction called the behavioral intermediate form, which is the basis for behavioral synthesis. Behavioral synthesis performs transformations similar to those found in optimizing compilers. Upon completion of behavioral synthesis, the optimized intermediate form is mapped to a register transfer level implementation.

2 Motivations

Many hardware description languages have been proposed and used in both academia and in industry. Most hardware description languages are oriented towards simulation. As high-level synthesis systems mature, a need arises for languages that aid not only in the *simulation* of hardware, but also in its *design* as well.

Several criteria must be met by a language for hardware design, they are described below.

1. *Supports full spectrum of design styles.*

   The language should support readily the varying spectrum of design styles of the designer, ranging from a pure behavioral description that is independent of the structural implementation, to a mixture of behavior and structure, to a pure structural description of the interconnection and instantiation of hardware modules.

   This criterion is crucial in a design environment since very often the designer has a particular structure in mind when designing hardware. This partial structure should be captured by the language, and reflected in the results of synthesis. A design often requires interfacing to an existing hardware unit, such as an ALU or incrementer. The ability to interface with external *structure* is of utmost importance in automated synthesis.

   Many synthesis systems and hardware description languages support only a specific design style, either pure structure or pure behavior. We believe a more effective approach to design is to use a flexible underlying language that captures the *essence* of the design from the designer, whether that essence be behavioral or structural.

2. *Supports simulation.*

   The language should support simulation in order to ascertain the correctness of a given description. As designs become bigger and more complex, it becomes more important to be able to simulate at all levels of synthesis, from behavioral to structural to logic to gate level.

3. *Simple to learn and use.*

   The language is a tool that the designer uses to capture and transform abstract ideas into complete designs. The tool must therefore be simple to learn and easy to use. Specifically, the language should contain the most basic constructs that are needed to describe a design. Details such as timing and delay should be left out of the language.

HardwareC attempts to satisfy the requirements stated above. As its name implies, it is based somewhat on the C programming language. However, several enhancements are made to increase the expressive power of the language, as well as to facilitate hardware description. The major features of HardwareC are described below.

- Notions of concurrent *processes* and *message passing*,
- *Templates* that allow a single description for a group of similar behavior (polymorphism). For example, an adder template describes all adders of any given size,

- *Instantiation* of procedures, similar to instantiating objects in object oriented languages, and

- Explicit *Input/Output* commands that access the ports of a given model.

HardwareC can be linked to the THOR simulation environment, which is also based on a C-like simulation language [4].

3 Modeling Hardware Behavior

Hardware behavior is modeled as a collection of concurrent and interacting processes. Each *process* consists of a hierarchy of *procedures*, and the processes interact and synchronize with each other through the use of *inter-process communication* mechanisms. This model is appropriate since hardware modules are allocated resources which continuously operate on a time varying set of inputs. A process upon completion will automatically restart execution with a new set of inputs.

The concept of processes and inter-process communication is powerful for both hardware and software models. In both domains, it allows the designer to:

1. Specify the parallelism between interacting modules at a high level, and

2. Isolate the communication and synchronization points between the processes in an explicit manner.

As an illustration of the use of processes and inter-process communication, consider the Intel 8251 UART (Figure 1). The UART is modeled as four concurrently executing processes. The *main* process accepts commands from the microprocessor and coordinates the execution of the other processes. The *transmitter* process writes data out on the serial interface, and the two receiver processes, *synchronous_receiver* and *asynchronous_receiver*, reads data from the serial interface. Note that the execution of each process is independent with respect to each other, and is synchronized through the use of inter-process communication. Inter-process communication is discussed further in Section 12.

HardwareC is a hardware description language for *synchronous* digital circuits. This is a reflection of the hardware model assumed by the HERCULES Synthesis system. Therefore, there is the notion of a *control state* that is sometimes used to describe the language. A *control state* is defined as an interval of time that corresponds to a system clock cycle in a synchronous system. When a particular operation is said to take one or more states, it means that the execution of the operation requires one or more clock cycles to complete before other operations that depend on it can begin.

The HardwareC language is described in the sections that follow.

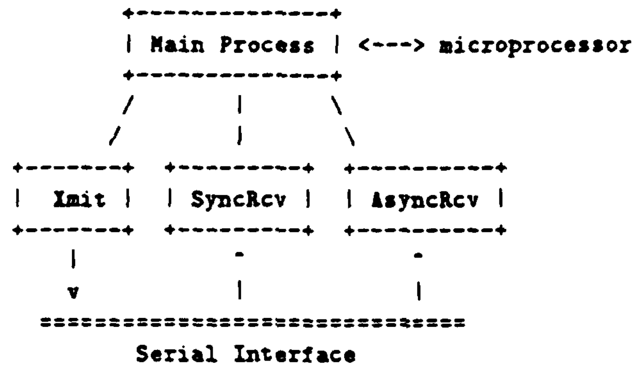


Figure 1: Hardware model for Intel 8251 UART

4 Program Structure

In *HardwareC*, there are two fundamental functional abstraction mechanisms – *process* and *procedure*. A process consists of a hierarchy of procedures, and executes concurrently and independently with respect to the other processes in the system. Similarly, a procedure is also a hierarchy of procedures. However, a procedure executes whenever it is called by another procedure or process.[1]

The transfer of data to and from a process is accomplished through the use of either parameters to the process or through message passing mechanisms (Section 12). The transfer of data to and from a procedure, on the other hand, is accomplished solely through the use of *parameters* to the procedure (Section 11). A procedure can neither return a value as the result of its invocation, nor use message passing to communicate with other procedures. The major differences between a process and a procedure are summarized below.

- *Process.* A process continuously operates on a time-varying set of input data. Upon completion of the last statement in its body, a process will restart its execution, operating on a possibly different set of inputs. An example of the definition of process *procA* is shown below. Note the use of the keyword **process** which prefixes the name of the process.

```
process procA( a, b, c )
      in boolean a;
      out boolean b;
```

[1] No recursive procedures are allowed

```
        inout boolean c[2];
{
        /* body of process */
}
```

- *Procedure*. A procedure can either be combinational or sequential, depending on whether the procedure requires any control states to execute. A sequential procedure begins execution whenever it is *called* by another procedure. Upon completion of execution, a procedure places valid data on its output ports, and returns control to the calling routine. For combinational procedures, execution involves propagating the input data through a network of combinational operations. An example of the definition of procedure *procB* is shown below.

```
procB( x, y, z )
        in boolean x;
        out boolean y;
        inout boolean z[2];
{
        /* body of procedure */
}
```

A procedure cannot be defined within the body of another procedure. This restriction follows the C language, which disallows nested procedural definitions. The resulting flattening of the procedural definition is appropriate since for hardware description, it is more convenient and secure to identify explicitly all inputs and outputs to a given procedure. A procedure defined within the scope of another allows access to all variables that are defined within the scope of its definition. As a result, a procedure's boundary is not well defined if nested procedural definitions are allowed.

Nested procedural definition is different from nested procedural invocation, the latter of which is both permitted and encouraged. For example,

```
/*                                  /*
 *  invalid procedure                *  valid procedure
 *    definition                     *    definition
 */                                  */

procA(a, b)                         validproc(x, y)
   ...                                 ...
```

```
{                                    {
    invalidproc(x, y)                    ...
    {                                }
        ...
    }                                procA(a, b)
    ...                                  ...
    invalidproc(...)                 {
                                         validproc( ... );
}                                    }
```

4.1 Statement Block

Statement block, more commonly known as compound statement, is used to group variable declarations and statements together so that they are syntactically equivalent to a single statement. A statement can either be a variable assignment, an if-then-else statement, a switch statement, a while statement, a for statement, an input/output statement, a message passing primitive, or a block. Semi-colons are used as terminators to statements. A semicolon by itself represents a null statement.

HardwareC supports two types of statement blocks – *parallelizable* blocks and *serial* blocks. Parallelizable blocks are encapsulated using curly braces ({ and }), whereas serial blocks are encapsulated using square brackets ([and]). The differences between the two types are:

- Parallelizable Block { }– The statements within a parallelizable block can all execute in parallel, subject to the data dependencies that exist between the statements. For example,

  ```
  {
      variable_declarations;

      statement1;
      statement2;
  }
  ```

  means that *statement1* can be executed concurrently with *statement2*. The degree of parallelism is determined by the synthesis system.

- Serial Block [ ] – The statements within a serial block are guaranteed to execute in serial order, starting from the first statement in the block. For example, *statement1* will always execute before *statement2*, regardless of their data dependencies.

```
[
        variable_declarations;

        statement1;
        statement2;
]
```

Serial block allows the designer the ability to specify control dependencies between otherwise data independent statements.

A description written using only serial blocks is always guaranteed to be correct – that is, the control dependencies between the statements are fully described. However, the description may not be efficient, since inter-statement parallelism is not exploited. In order to specify such parallelism, the designer should use whenever possible parallelizable blocks ({ }) in describing hardware.

4.2 Parameter Classes

The parameters to processes and procedures are categorized into three different classes: in, out, and inout. Input (in) parameters can only be referenced within the body of a routine: assignments to input parameters are illegal. Output (out) parameters can be modified within the body of a routine; references to output parameters are illegal. Input/output (inout) parameters are bidirectional lines that can be either referenced or assigned. The access protocol to this bidirectional line is left to the designer, and specified as part of the high-level description. Note that an inout parameter is not simply data that will both be read and modified in the routine. It is reserved for the description of bidirectional lines.

For example, *Busy* is an in parameter that controls the access to an inout parameter *Data*. *AllZero* is an output parameter that returns a flag on whether *Data* is all zero.

```
process test( Busy, Data, AllZero )
        in boolean Busy;
        inout boolean Data[8];
        out boolean AllZero;
[
        while ( Busy )
                ;
        /* write to Data */
        Data = newdata;
        write Data;
        AllZero = (Data == 0);
```

]

Notice the use of the serial block ([ ]) to ensure that the write to *Data* occurs after the busy waiting while loop, which does not have any data dependencies with respect to the write.

4.3 Declare Before Use

Whenever a procedure is called, the arguments to the invocation are checked for both compatibility in the variable size and type, as well as for compatibility in the parameter classification. For instance, an input parameter cannot be used as the argument to a procedure call that requires an output parameter. Similarly, an output parameter cannot be used as the argument to a procedure call that requires an input parameter. This compile time consistency checking improves the security of the language.

In order to provide this information to the parser, it is necessary to *declare* a procedure before it can be called. The declaration of a procedure involves specifying:

1. *Name* of the procedure - can be any alphanumeric string beginning with a character.
2. *Number* and *order* of parameters - only Boolean parameters are allowed.
3. *Sizes* of the parameters - the size of a parameter can be specified in terms of a constant, or an expression that evaluates to a constant.
4. *Classes* of the parameters - in, out, or inout.

An example of the declaration for a procedure is shown below.

```
# define MAX 4

declare example( a, b, c )
    in boolean a;
    out boolean b[MAX];
    inout boolean c[MAX+1];
```

The actual names of the parameters are irrelevant; they are used only for the purpose of specifying the classes and sizes of the corresponding parameters. Another example is shown below.

```
declare sum( x, y, z )
    in boolean x;
    out boolean y[2];
```

```
      inout boolean c;

foo( ... )
{
      /* ... */
      sum( a, b, c );
      /* ... */
}
```

If the declaration of *sum* is not supplied, then the subsequent call in foo will be invalid. Similarly, for inter-process communication through message passing, it is necessary to predeclare a particular process before sending or receiving messages from it. The declaration for a process is exactly similar to the declaration for a procedure, with the sole exception of the keyword process that prefixes the name. For example, the declaration for a process named *foobar* is as follows.

```
declare process foobar( a, b, c )
      in boolean a[3];
      out boolean b[4];
      inout boolean c[2];
```

5 Data in HardwareC

There are two types of *data* entities in the language – constants and variables. They are described in the following sections.

5.1 Constants

There are two types of constants in the language – *integer constants* and *hexadecimal constants*. Integer constants are positive numbers described in the decimal notation. For example, 5 and 223 are integer constants. Hexadecimal constants are numbers described in the hexadecimal notation. They are prefixed by 0*x*, followed by a string of hexadecimal digits { 0 – 9, a, b, c, d, e, f }. For example, 0*xf* represents 15, and 0*x*10 represents 16. Binary constants are subsets of hexadecimal constants, where 1 is represented as 0*x*1, and 0 is represented as 0*x*0.

Negative constants are not represented in the language. This restriction stems from the independence of HardwareC to a particular style of complementation. Therefore, if the designer wishes to specify −3 in one's complement notation, then he should specify the bit-wise representation of the value using

hexadecimal constants. For an 8-bit number in one's complement notation, -3 is represented as $0xf8$.

5.2 Variables

There are two variable *types* in the language – *Boolean* and *integer*. Boolean variables are mapped to wires or registers in the final hardware, whereas integer variables are provided for the convenience of the description, and will be resolved at compile time during behavioral synthesis.

A variable may be declared within any block ({ } or [ ]) of any arbitrary nesting. The semantic follows that of block structured languages, where a variable is visible only within the scope of its definition. A variable with the same name at a deeper nesting block level will override any current definition of the variable.

For instance, all declarations in the following example are valid.

```
{
        int i;
        boolean x;
        {
                int i; /* new integer */
                boolean x, y;
                ...
        }
        /* y is not defined here */
}
```

No global variables are allowed in *HardwareC*. This restriction is due to the fact that global variables allow side effects that are not explicitly identified. This is undesirable from the standpoint of security, verifiability, and program readability. If some data must be shared between two routines, then the data should be explicitly specified as common parameters to the two routines.

Integer Integer variables can only be scalar quantities. Integer variables may be used in any arithmetic, Boolean, and relational expressions. They can also be used as indices to constant iteration loops (for loop), and as indices for accessing components of Boolean vectors and matrices. The following example demonstrates the use of integer variables and expressions in accessing components of a Boolean vector.

```
/*
 *    swaps the two nibbles in "a" to "b"
 */
```

```
swap(a, b)
    in boolean a[8];
    out boolean b[8];
{
    int i, j, k;

    k = 3;

    /* copies LSB nibble to b */
    for i = 0 to k do
        b[ i+4:i+4 ] = a[ i:i ];

    /* does exactly the same thing */
    i = 0;
    j = 3;
    b[ i+4:j+4 ] = a[ i:j ];

    /* copies MSB nibble to b */
    b[ i:j ] = a[ i+4:j+4 ];

    write b;
}
```

The exact syntax on accessing components of a Boolean vector is discussed in the next section. The example below shows the use of integer expressions and values in control structures.

```
int i;
boolean vec[24];

for i = 0 to 7 do {
    switch (i) {
    case 0:
        vec[ 3*i:3*i+2 ] = 0x7;  /* binary 111 */
        break;
    default:
        vec[ 3*i:3*i+2 ] = i;
        break;
    }
}

/* vec should have the following value:
        111 110 101 100 011 010 001 111
        MSB                         LSB
 */
```

Boolean A Boolean variable represents one or more signals, where each bit of the variable corresponds to a *signal* that can be either 0 or 1. Boolean variables can be scalar, vector, or matrix. For example, the following declarations are all valid Boolean variables. In particular, a is a scalar, b is a vector of five elements starting from index 0, and c is a matrix of 25 elements, with the rows starting from 0 to 4, and columns starting from 0 to 4.

```
boolean a;          /* scalar */
boolean b[5];       /* vector */
boolean c[5][5];    /* matrix */
```

In Boolean vectors, specifying the variable name without brackets, or with empty brackets, represent the *entire* vector. For example, b and b[] are equivalent to b[0:4]. Columns of a Boolean matrix can be accessed similarly. For example, c[2] and c[2][] are equivalent to c[2][0:4]. Since assignments to matrices are not permitted, a reference to c will automatically be converted to c[0][0:4], the first row of the matrix.

For Boolean vectors and matrices, it is also possible to access a *subrange* of values. This is specified by the colon (:) notation. For example, b[2:3] represents a vector of two values that corresponds to the third and fourth element of b. The most significant bit (MSB) is always the higher index, with the least significant bit (LSB) being the smaller index.

Integer variables and expressions can be used in variable declarations to specify the dimensions of the variable, or they can be used to access components and subranges of Boolean variables. For example,

```
int i;

i = 3;
c[i][i:i+1] = b[0:1];
{
    boolean q[i+1];     /* q has 4 elements */
}
```

Boolean variables are further classified as *local*, *static*, and *register*.

- **boolean** – Local Boolean variables are the default. A local boolean is initialized to zero, and its value is not saved across procedure invocations. For example,

```
boolean flag;
boolean vectorflag[2], matrixflag[2][3];
```

- **static** – Static Boolean variables are similar to local Boolean variables, with the semantic difference that their values are retained across procedural invocations. For example,

```
static internal_state[2];
```

Static variables will always be implemented with storage elements such as registers.

- **register** – Register Boolean variables are architected registers that are specified by the designer. Similar to static variables, they also retain their values across procedural invocations. Every assignment to a register variable immediately loads the corresponding register with a new value. For example,

  ```
  register status[8];
  ```

  The difference between register and static variables is in how assignments are handled, which is discussed next.

Assignments to boolean and static variables are resolved during behavioral synthesis, and hence do not require any control states for run-time execution. In contrast, each assignment to a register variable corresponds to the loading of the register with a new value, and hence requires a control state at run-time.

To demonstrate the differences between static and register variables, consider the two examples below. In procedure *foo*, each assignment to the static variable *c* will not consume a control state at run-time. This is due to the fact that the assignment only changes subsequent references to c, and hence does not imply loading the register that implements *c* with a new value.

```
foo()
{
    static c;

    c = 1;     /* change reference only */
    c = 0;     /* change reference only */
    c = 1;     /* change reference only */
    c = 0;     /* last value of c is 0 */
}
```

Similarly, the register variable *c* in procedure bar also has a final value of 0. The difference is that during the execution of bar, the register is loaded with 4 values, corresponding to each assignment to the variable. Therefore, the procedure requires four control states to execute.

```
bar()
{
    register c;
```

```
    c = 1;     /* register has 1 */
    c = 0;     /* register has 0 */
    c = 1;     /* register has 1 again */
    c = 0;     /* last value of c is 0 */
}
```

In terms of port behavior, static and register variables perform the same action – retain values across invocations. Architected registers allow the designer explicit control over the contents of the register. They are useful for testability purposes where the designer wishes to check the contents of a particular register during execution. For example, architected registers are often used as a status register in a processor description.

5.3 Variable Declaration

The dimension of a Boolean variable may be specified as either a constant, an integer variable, or an integer expression. For instance, consider the declarations for Boolean variables a and b.

```
{
    int i;

    i = 3;
    {
        boolean a[i];  /* a has 3 elements */
        ...
    }
    i = 8;
    {
        boolean b[i+3];  /* b has 11 elements */
        ...
    }
}
```

In fact, even the dimensions of the parameters can be specified as arbitrary integer expressions. The delayed binding of variable dimension to variable definition greatly increases the expressiveness of the language, and improves the flexibility and adaptability of an input description. An illustration of the use of integer expressions in parameter declaration is shown below.

```
# define MAX 8

declare foo(a, b)
    in boolean a[MAX+1];     /* 9 elements */
```

```
out boolean b[MAX+2];    /* 10 elements */
```

Note that the integer expressions (MAX+1) and (MAX+2) are used to declare the dimensions of the parameters.

6 Control Flow Constructs

HardwareC supports a single-in, single-out control flow, similar to the Pascal programming language. This implies that no gotos and returns are allowed in the language. Such restriction is appropriate since by supporting a single-in, single-out control flow, the semantics of the language is made simpler. which greatly aids in the correctness verification of programs. The four major control flow constructs are *if*, *switch*, *for*, and *while*; they are described below.

- *if*

  selects among two alternatives. depending on whether the conditional expression evaluates to TRUE or FALSE. non-zero or zero. respectively. The conditional expression can be any arithmetic, Boolean. and relational expression that involves both integer and Boolean variables. The else part may be unspecified. For example.

  ```
  if ( ! b & chipselect )
      a = 0;     /* any statement */
  else
      a = 1;     /* any statement */
  ```

- *switch*

  selects among one or more alternatives. depending on the value of the switch conditional expression. The individual cases in the switch statement may be cascaded, and are delimited through the use of break statements.

  ```
  switch ( <switch expr> ) {
  case num1:
      <statement>
  case num2:
  case num3:
      <statement>
      break;
  ...
  default:
      break;
  }
  ```

break statements are illegal in any other contexts, such as for premature exits from while loops.

- *for*

  is a constant bound iteration on a given integer variable. The exact syntax is as follows,

  ```
  for <int var> = expr1 {to|downto} expr2 [step expr3]
      do <statement>
  ```

  where expr1, expr2, and expr3 can be any constant or integer expression. The step clause is optional, and has a default of one.

- *while*

  is a data dependent iteration on a given Boolean expression. The syntax of the while loop is as follows.

  ```
  while ( loop_expr )
      <statement>
  ```

  loop_expr can be any integer, relational, or Boolean expression.

There are only two types of iterative loop constructs in HardwareC - for loops and while loops. For loops are deterministic iteration loops, whose bounds are known at compile time. A while loop is a non-deterministic iteration whose exit condition can be data dependent, and hence is in general unknown at compile time. Whereas there are many variants of data-dependent loops, such as do-while and repeat-until, they can all be written in terms of while loops. Including the many variants of data-dependent loops does not increase the expressive power of the language. Therefore, for reasons of simplicity, HardwareC supports only one style of data-dependent loops - the while loop.

7 Assignments

When a program references a particular variable at different locations in the code, it may reference different *values*, depending on whether the variable has been re-assigned between the references. The *value* of a variable is defined to be the data most recently assigned to it. [2] An assignment to a variable modifies the *value* of the variable.

Both Boolean and integer variables, as well as inout and out parameters, can be assigned. Note that an assignment is not an expression that returns a

[2] Data is defined to be the results of procedure call, binary and unary operators, or message passing.

value. For example, $a = a + 1$ does not return the new value of a, which is in contrast to the C programming language.

The semantics of assignment to different types of variables are described below.

- **out** or **inout** parameters.

  Assignments to an output or input/output parameter will update the *value* of the port. The actual reflection of the port values to externally visible signals is performed explicitly through the use of input/output commands, discussed in Section 11.

- **int**, **boolean**, or **static** variables.

  Assignments to these variables will be resolved and removed during behavioral synthesis. Therefore, the assignments serve only to update the *value* of the variable for subsequent references, and do not consume any control states at run-time. Only constants or integer expressions can be assigned to integer variables. There is no restriction on the values that can be assigned to Boolean variables.

- **register** variables.

  An assignment to an architected register loads the register with a new value. Therefore, every assignment requires a control state for execution at run-time.

8 Templates

Very often two descriptions differ in only very restricted ways. For example, they are the same with the sole exception that the variable sizes are different, as illustrated below for a four-bit and a five-bit adder.

```
/*
 *  Four-Bit adder
 */
adder4(a, b, c, cin, cout)
    in boolean a[4], b[4], cin;
    out boolean c[4], cout;
{
    /* 4 bits add */
}

/*
 *  Five-Bit adder
 */
```

```
adder5(a, b, c, cin, cout)
    in boolean a[5], b[5], cin;
    out boolean c[5], cout;
{
    /* same as above, but for 5 bits */
}
```

It is much simpler and expressive if only *one* description is given for the adder function which takes an argument specifying the size of the operation. This approach offers the advantages of (1) consistency of descriptions, (2) economy of code, which decreases design time, and (3) reusability of code (polymorphism).

In HardwareC, the mechanism which supports parameterized descriptions is a *template*. A template can either be used to generate a procedure or a process. Templates are similar to generic packages in ADA, or generic classes in several object oriented languages. A template takes one or more integer arguments as parameters, and given a particular mapping of integer values to the integer parameters, a corresponding *instance* can be obtained. A good analogy can be made between templates and module generation; in fact, a template is a form of high-level module generation. The exact syntax of templates for procedures and processes is given below.

- Procedure Template definition:

```
template <procedure_name> ( <parameters> )
                    with ( <integer_parameters> )
    <parameter declarations>
{
    <body>
}
```

- Process Template definition:

```
template process <procedure_name> ( <parameters> )
                            with ( <integer_parameters> )
    <parameter declarations>
{
    <body>
}
```

The keyword `template` prefixes the name of the template, and the keyword `with` separates the Boolean parameters from the integer parameters. The `integer_parameters` are the names of the integer parameters, and are separated by commas (,) if more than one is present. These integer parameters can be

used in both parameter declaration and the body of the template as integer constants. Specifically, **assignment to an integer parameter is not allowed.**

Let us consider the description of a template for the ripple-carry adder function.

```
/*
 *     ripple carry adder
 */
template adder(a, b, c, cin, cout) with (size)
    in boolean a[size], b[size], cin;
    in boolean c[size], cout;
{
    int i, j;
    boolean temp;

    temp = cin;
    j = size -1;
    for i = 0 to j do {
        c[i:i] = a[i:i] xor b[i:i] xor temp;
        temp = a[i:i] & b[i:i] |
               temp & (a[i:i] | b[i:i]);
    }
    cout = temp;
    write c, cout;
}
```

Templates can be used in two ways, corresponding to the environment level and the language level.

1. *Environment Level* – The HERCULES Synthesis system can create any number of instances of a given template. This is useful for example in generating library units such as adders or incrementers.

2. *Language Level* – Within the description, the designer can make references to particular instances of a template through *instantiating* templates. Template instantiation is described next.

9 Instances

HardwareC supports *explicit instantiation* of procedures and procedure templates in the description. A instance of a procedure represents an *object* that encapsulates both behavior and state. In a similar manner, a Boolean variable is also an *object* whose behavior is specified by the language in terms of the semantics of accessing and modifying the variable. Instances can therefore

be treated as *instance variables* that are declared and used in the scope of its definition. The syntax of procedure instantiation is described below.

```
instance <procedure_name> inst1, inst2, ..., instn;
```

The keyword `instance` prefixes the name of the procedure , followed by the names of the instances to be created, separated by commas. If a template is instantiated, the syntax is described as follows.

```
instance <template_name>( <integer_arg> ) t1,...,tn;
```

the arguments to the integer parameters should be specified, separated by commas. The integer arguments must be constants, and cannot be any variables or expressions. Note that the scoping rules for variable visibility also apply to instances. Consider the example below, where `counter` is a procedure that increments an internal variable each time it is called.

```
{
    instance counter a;
    instance adder(4) o4;     /* 4 bit adder */

    {
        instance counter a, b;

        a(...); /* new counter */

        /* can access o4 also */
    }

    a(...);  /* old counter */

    /* b is undefined here */
}
```

The instance a of `counter` is different for each different nesting of the block.

9.1 Calling a Procedure

A procedure may be *called* by another process or procedure. This is accomplished by specifying the name of the procedure to be called, along with the arguments to the procedure separated by commas and enclosed in parentheses. A procedure must be *declared* or *defined* before it can be called, otherwise the call will result in an error.

Valid arguments to a procedure call depend on the particular class of the corresponding parameters. Specifically,

- **In Parameter** - All in parameters, inout parameters, local boolean, static, and register variables and expressions are allowed to be used as arguments in the procedure call.

- **Out Parameter** - All out parameters, inout parameters, local boolean variables, and static variables are allowed. No register variables are allowed.

- **Inout Parameter** - Only inout parameters are allowed.

There are two types of procedure calls - *generic* or *instantiated* calls. A generic call is a call made to a given procedure *type*. The particular instance of the procedure type that is used to implement the call is not specified. To invoke a particular instance of a procedure, the name of the instance simply replaces the name of the procedure in a procedure call. This style of procedure call is called an *instantiated* procedure call. For example,

```
{
    instance counter x;

    counter(...);      /* generic call */
    x(...);            /* instantiated call */

    counter(...);      /* generic call */
    x(...);            /* instantiated call */
}
```

All the calls to x will invoke the same instance. However, if a procedure is invoked without specifying the instance (generic procedure calls), then the synthesis system is free to determine whether the call can be shared with other generic calls, or whether to allocate an instance to the call.

9.2 Advantages of Instantiating Procedures

There are several advantages in supporting both generic and instantiated procedure calls. They are briefly described below.

1. *Resolves Ambiguity in the behavior.* The designer can *completely* describe the behavior that is intended without relying on hidden assumptions.

2. *Access to both State and Behavior.* The designer can access not only behavior through procedure calls, but also internal state information as well.

3. *Supports Spectrum of Design descriptions.* Depending on the style of the designer, hardware can be described in a spectrum ranging from *pure*

behavior that is free from structural implications to *pure structure* that describes the interconnection and instantiation of hardware components. A procedure instantiation is similar to instantiating a hardware module. and therefore HardwareC supports fully the spectrum of design description styles.

4. *Specifies Resource Sharing at description level.* Although it is not required by the synthesis system, it is possible for a designer to specify the sharing of resources (procedure instances) through the use of instantiating procedures. For example, the designer can specify whether only one adder should be used to implement a description verses one that uses two adders.

9.3 Motivation and Example

A major drawback with many languages is the inability to specify exactly which instance of a given procedure is invoked in a procedure call. This restriction is reasonable for procedures that describe only the functionality without internal state information. However, if a procedure has internal state associated with it (through the use of either static or register variables), such restriction severly handicaps the usability and expressiveness of the language. In fact. the such deficiency can result in either inefficient or even incorrect implementation. depending on whether the assumptions made by the synthesis system matches those made by the designer.

Consider the description of a counter below:

```
/*
 *    each call to it increments by 1
 */
counter(value)
    out boolean value[8];
{
    static state[8];

    state = state + 1;
    value = state;
    write value;
}
```

Every call to the counter module will increment the corresponding internal state variable by one. If a call is made to counter without specifying the particular instance that is to be invoked, then one of two situations will arise.

1. Single instance assumption – If the synthesis system assumes that one and only one instance is associated with a procedure, then a call to counter

will always increment the same internal state (corresponding to the single instance).

However, this approach is overly restrictive since one of the powers of synthesis systems is to explore the spectrum of design tradeoffs between parallel and serial implementations, and by always assuming one instance per procedure this exploration is not possible.

2. No assumption on the invoked instance – On the other hand, if no assumptions are made on which instance a given call will invoke, the synthesis system will then have the flexibility to either dedicate an instance to the call, or share several procedure calls onto the same instance. However, if the procedure has internal state information, then the description can be incorrect, dependent on the particular mapping of procedure calls to procedure instances.

The assumptions that are made by the synthesis system may not be what the designer had in mind when writing the description. For instance, in the code segment below, **counter** is called twice.

```
counter(sum1);
...
counter(sum2);
...
```

The designer can either view the two calls as incrementing the same value twice, or he can view the two calls to be distinct, each incrementing a value independent of the other. Through instantiation of procedures, the designer can explicitly specify the exact semantics of a procedure call. For example, if the designer wishes to increment a single value twice, then the corresponding code is given below.

```
{
    instance counter value;

    value(...);     /* increment */
    value(...);     /* increment again */
}
```

On the other hand, if the designer wishes to increment two different values, then the code is as follows.

```
{
    instance counter value1, value2;

    value1(...)    /* increment value1 */
```

```
        value2(...)     /* increment value2 */
        value1(...)     /* increment value1 again */
    }
```

The designer can instantiate not only procedures, but also procedure *templates*. This is accomplished by supplying the values to the integer parameters to the corresponding template, separated by commas if more than one value is required. The example below makes use of the adder template described in Section 8.

```
    {
        instance adder(4) o4;     /* 4 bit adder */
        instance adder(5) o5;     /* 5 bit adder */

        o4(...);
        o5(...);
    }
```

10 Operators

HardwareC supports all Boolean and relational operators available in the conventional C programming language. It also supports all arithmetic operators. The operators can be unary or binary, and take both integer and Boolean variables as operands. Mixed operations between Boolean and integer variables are also allowed if it makes sense. For instance, Boolean inversion on an integer variable is illegal.

The operators are summarized below.

Arithmetic { +, −, *, / } Applies to both integer and Boolean variables and expressions.

Boolean { !, &, |, xor } bitwise Boolean operators, shift left (<<), shift right (>>), rotate left (rl), and rotate right (rr). Applies to only Boolean variables and expressions.

Relational { != , ==, <=, >=, <, > } Applies to both integer and Boolean variables and expressions.

Auto-Increment/Auto-Decrement { ++, −− } Applies to both integer and Boolean variables and expressions. For example, $a++$ and $++a$ are equivalent to $a = a + 1$, whereas $a--$ and $--a$ are equivalent to $a = a - 1$.

11 Input/Output

HardwareC has explicit input and output commands to allow reading from and writing to the ports of a process or procedure. The three main commands are *write*, *free*, and *read*, and are described below.

write writes the most recently assigned 'value' of an output or inout parameter onto the corresponding ports. Different semantics exist for different types of parameters; they are summarized below.

- No **write** is specified for an **inout** or **out** parameter – any change made to that parameter will not be visible. In the example below, the assignments to the **inout** parameter **a** do not affect the value of the port; they only serve to alter the value of **a** for subsequent references.

```
inout boolean a;
...

a = 1;    /* port unaffected */
a = 0;    /* port unaffected */
a = 1;    /* port unaffected */
...
```

- Singe **write** to an out parameter – in many situations, the designer wishes to connect an output port directly to the result of a particular operation. There are two advantages for using direct connection. First, it does not waste a state at run-time. Second, direct connection allows external visibility of a particular operation.

  Direct connection is achieved by specifying a single **write** for an output parameter in the body of the routine. For instance, the output parameter **z** is connected directly to the output of the adder in the example below.

```
while (run) {
    temp = temp + 1;
    z = temp;
    write z;    /* direct connection */
}
```

Any value written to a port will be retained until either the next **write** or **free** statement. In the example below, the out parameter **c** will generate a pulse on the ports.

```
out boolean c;
```

```
    ...
c = 0
write c;  /* port has 0 */
    ...
c = 1;
write c;  /* port has 1 */
    ...
c = 0
write c;  /* port has 0 again */
```

free sets the corresponding output or inout port to high impedance float value. For both **free** and **write**, the effect of the change on port boundary will take place exactly one cycle after the statement begins execution. Any write to a port that has been set to float state will overwrite it with the new value. For example,

```
out boolean d;
...

d = 1
write d;     /* port has 1 */
free d;      /* port has high-Z */
write d;     /* port has 1 again */
```

read samples the corresponding in or inout port into a register, and returns the output of the sampling register. Execution of a **read** statement will take one cycle to complete. For example,

```
y = read( x );
/* y is sampled version of x */
```

12 Inter-Process Communication

There are two paradigms for inter-process communication – *shared medium* and *message passing*. Shared medium communication refers to the transfer of information between modules through a common set of ports. The protocol which governs correct handshaking between the modules is provided by the designer, and is described as an integral part of the high-level description. Message passing communication, on the other hand, utilizes explicit *send* and *receive* operations to synchronize between the two concurrent processes.

Each approach has its advantages and limitations. For example, in communication through shared medium. the performance advantage is offset by an increase in the complexity of the resulting high-level description. Likewise, the conceptual elegance of message passing solves both synchronization and communication in systems, but may result in unacceptable implementation complexity if it is used without restraint.

HardwareC offers both approaches. First, it allows *shared medium* communication through the use of *parameters* to processes or procedures. Second, it allows a synchronous *send-receive* message passing scheme with fixed-size messages. The size of a message represents the number of bits that is communicated between the processes, and may be specified by the designer in the input description. Synchronous message passing provides a simple yet powerful approach to inter-process synchronization and limited data transfer without incurring the cost of message buffering.

12.1 Message Passing Primitives

There are three primitive operations in message passing: *send*, *receive*, and *msgwait*. Only processes can use the message passing primitives. *send* transmits a fixed size message to another process. The current process will wait and synchronize until the corresponding process issues a *receive*, whereupon the transfer of information will take place. For example, `targetprocess` is the receiving process, and `message` is the message to be sent.

```
send( targetprocess, message );
```

receive accepts a message from a given process, and will wait and synchronize until the corresponding process issues a *send*. For example, `sourceprocess` is the sending process, and `buffer` is the message received.

```
receive( sourceprocess, buffer );
```

msgwait is a query that returns a scalar Boolean flag signifying whether the specified process is currently sending to the current process. For example, `Producer` and `Consumer` are two processes that synchronize with each other using message passing.

```
process Producer(...)
{
    /* generate item */
    send( Consumer, item );
}

process Consumer(...)
{
```

```
        if ( msgwait(Producer) )
        {
            receive( Producer, item );
            /* consume item */
        }
        else
            /* producer not ready */
    }
```

There is a system wide *message size*, which is the bandwidth of the communication channel between the processes. The default is 8 bits wide, and it can be changed by specifying the size in the description as follows.

```
    /* <num> is new message size */
    ipcsize = constant_number;
```

ipcsize is a keyword in the language, and **constant_number** is a positive integer constant. The message size change must be done before any message passing operation takes place, as the parser will check to ensure proper size messages and buffers are used in the send and receive operations. The assignment should not be within the body of any particular process or procedure, and should lie between procedural definitions.

13 Miscellaneous

HardwareC relies on the C preprocessor during parsing to handle macro definition (#define) and file inclusion (#include) facilities. The designer is free to use any C preprocessor commands in the description.

14 Appendix

Four detailed examples of hardware description using HardwareC are described below. The first is a four bit carry look-ahead adder. The second is a counter process that uses the four bit adder. The third is the traffic light controller described in the Mead-Conway book. The final example is the Intel 8251 UART description.

Four-Bit Adder

```
add4bit( a, b, carryin, result, carryout )
      in boolean a[4];
      in boolean b[4];
      in boolean carryin;
      out boolean result[4];
```

```
    out boolean carryout;
{
    int i;
    boolean P[4], G[4], new;
    /*
    * calculate propagate and generate
    */
    for i = 0 to 3 do
        P[i:i] = a[i:i] xor b[i:i];
    for i = 0 to 3 do
        G[i:i] = a[i:i] & b[i:i];
    /*
    * calculate carryout
    */
    carryout = G[3:3] | (P[3:3] & G[2:2])
        | (P[3:3] & P[2:2] & G[1:1])
        | (P[3:3] & P[2:2] & P[1:1] & G[0:0])
        | (P[3:3] & P[2:2] & P[1:1] & P[0:0] & carryin);
    /*
    * calculate sum
    */
    new = carryin;
    for i = 0 to 3 do {
        result[i:i] = P[i:i] xor new;
        new = G[i:i] | ( P[i:i] & new );
    }

    write result, carryout;
}
```

Counter

```
process counter( run, load, updown, data, sum )
    in boolean run,
        load, updown,
        data[4];
    out boolean sum[4];
{
    boolean temp[5];

    while ( run ) {
        if ( load )
            temp = data;
```

```
        else {
            if ( updown )
                add4bit(temp, 1, 0, temp[0:3], temp[4:4]);
            else
                add4bit(temp, 0xf, 0, temp[0:3], temp[4:4]);
        }
        sum = temp[0:3];
        write sum;
    }
}
```

Traffic controller

```
/*
 * Mead/Conway Traffic Light Controller
 */

# define HIWAY_GREEN  0
# define HIWAY_YELLOW 1
# define FARM_GREEN   2
# define FARM_YELLOW  3

# define GREEN  1
# define YELLOW 2
# define RED    3

# define TRUE  1
# define FALSE 0

process traffic ( run, Cars,
              TimeoutL, TimeoutS,
              HiWayL, FarmL, StartTimer )
   in boolean run;
   in boolean Cars,
            TimeoutL,
            TimeoutS;
   out boolean HiWayL[2],
            FarmL[2],
            StartTimer;
{
   static state[2];
   boolean newstate[2];

   while ( run ) {
```

```
/* combinational logic
   to determine nextstate
*/

switch (state) {
case HIWAY_GREEN:
   HiWayL = GREEN;
   FarmL = RED;

   if (Cars & TimeoutL) {
      newstate = HIWAY_YELLOW;
      StartTimer = TRUE;
   } else {
      newstate = HIWAY_GREEN;
      StartTimer = FALSE;
   }
   break;

case HIWAY_YELLOW:
   HiWayL = YELLOW;
   FarmL = RED;

   if ( TimeoutS ) {
      newstate = FARM_GREEN;
      StartTimer = TRUE;
   } else {
      newstate = FARM_YELLOW;
      StartTimer = FALSE;
   }
   break;
case FARM_GREEN:
   HiWayL = RED;
   FarmL = GREEN;

   if ( ! Cars | TimeoutL ) {
      newstate = FARM_YELLOW;
      StartTimer = TRUE;
   } else {
      newstate = FARM_GREEN;
      StartTimer = FALSE;
   }
   break;
case FARM_YELLOW:
```

```
            HiWayL = RED;
            FarmL = YELLOW;

            if ( TimeoutS ) {
               newstate = HIWAY_GREEN;
               StartTimer = TRUE;
            } else {
               newstate = FARM_YELLOW;
               StartTimer = FALSE;
            }
            break;
         }

         state = newstate;
         write HiWayL, FarmL, StartTimer;
      }
   }
```

Intel 8251 UART There are four processes – main, xmit, sync_recv, and async_recv. They communicate through send/receive message passing primitives.

```
/*
 *      i8251 UART - HardwareC version
 *
 *      Written by David Ku
 *      Stanford University
 */

# define      DataSize      8

# define      forever       1
# define      wait(f)       while (f)

# define      TRUE          1
# define      FALSE         0

/*
 *      field definition
 */

# define      eh           control[7:7]
# define      ir           control[6:6]
```

```
# define      rts         control[5:5]
# define      er          control[4:4]
# define      sbrk        control[3:3]
# define      rxE         control[2:2]
# define      dtr         control[1:1]
# define      txen        control[0:0]

# define      dsr         status[7:7]
# define      syndet      status[6:6]
# define      fe          status[5:5]
# define      oe          status[4:4]
# define      pe          status[3:3]
# define      txe         status[2:2]
# define      rxrdy       status[1:1]
# define      txrdy       status[0:0]

# define      scs         mode[7:7]
# define      nsbits      mode[6:7]
# define      esd         mode[5:5]
# define      ep          mode[4:4]
# define      pen         mode[3:3]
# define      nbits       mode[1:2]
# define      brate       mode[0:0]

/*
 *      hunt_mode()
 *
 *      searches in synchronous
 *      receive mode for sync chars
 */

hunt_mode( rxd, drdy, sync1, sync2, mode )
      in boolean rxd;
      in boolean drdy;
      in boolean sync1[DataSize],
                 sync2[DataSize],
                 mode[DataSize];
{
      boolean done;
      boolean data[DataSize];
      boolean ncount[3];

      done = FALSE;
```

```
        while ( ! done ) {

                data = 0xff;
                while ( data != sync1 ) [
                        wait ( drdy );
                        data[7:7] = read ( rxd );
                        data = data >> 1;
                        done = TRUE;
                ]

                if ( mode[7:7] == 0 )  {

                        ncount[2:2] = 1;
                        ncount[0:1] = nbits;
                        while ( ncount ) [
                                wait ( drdy );
                                data[7:7] = rxd;
                                data = data >> 1;
                                ncount = ncount - 1;
                        ]

                        done = (data == sync2);
                }
        }
}

/*
 *      xmit - transmit process
 */

declare process i8251(ChipSelect,
            WriteEnable, ReadEnable, ChipData,
            data, valid, sync1, sync2, mode,
            control, status)
        in boolean ChipSelect;
        in boolean WriteEnable;
        in boolean ReadEnable;
        in boolean ChipData;
        inout boolean data[DataSize];
        out boolean valid;
        out boolean sync1[DataSize];
        out boolean sync2[DataSize];
        out boolean mode[DataSize];
        out boolean control[DataSize];
```

```
      in boolean status[DataSize];


process xmit(cts, txd, xdrdy, valid,
          mode, status, control,
          sync1, sync2)
      in boolean cts;
      out boolean txd;
      in boolean xdrdy;
      in boolean valid;
      in boolean sync1[DataSize],
                 sync2[DataSize];
      in boolean mode[DataSize],
                 control[DataSize];
      out boolean status[DataSize];
{
      int i;
      boolean sync_mode;          /* sync mode */
      boolean sync_flag;
      boolean data_ready;
      boolean par;
      boolean ncount[3];          /* # bits */
      boolean dbuf[DataSize];
      boolean xdata[DataSize];
      boolean okay[DataSize];

      free status;

      /*
       *     initialization - valid
       *     true when sync1/sync2 is ready
       */

      if ( valid ) {

            txd = 1; txe = 0;
            write txd;

            sync_mode = (mode[6:7] == 0);

            /*    wait for enable   */

            wait ( txen & cts );
```

```
/*      check for sbrk      */

if (! sync_mode & sbrk)
[       txd = 0;
        write txd;
        wait ( (!sbrk) | (!txen) | (!cts) );
        txd = 1;
        write txd;
]

/*
 *      wait if in async mode,
 *      or send sync char if sync
 */
txe = 1;
write txe;
free txe;

if ( sync_mode )
{       /* check if message are pending */
        if ( msgwaiting(i8251) )
                receive(i8251, xdata);
        else
        {       if (sync_flag)
                        xdata = sync1;
                else
                        xdata = sync2;
                sync_flag = ! sync_flag;
        }
}
else
        receive(i8251, xdata);


/*
 *      send start bit
 */
if ( ! sync_mode )
[       wait (xdrdy);
        txd = 0;
        write txd;
]

/*
```

```
 *      send data
 */
ncount[2:2] = 1;
ncount[0:1] = nbits;
while ( ncount )
[      wait (xdrdy);
       txd = dbuf[0:0];
       write txd;
       ncount = ncount - 1;
       dbuf = dbuf >> 1;
]

/*
 *      send parity bits if required
 */
if (pen)
[
       par = xdata[0:0];
       for i = 1 to 7 do
              par = par xor xdata[i:i];

       if (! ep)
              par = ! par;
       wait (xdrdy);
       txd = par;
       write txd;
]

/*
 *      send stop bits
 */
if (! sync_mode)
[      wait (xdrdy);
       txd = 1;
       write txd;
]

write status;
      }
]

/*
 *      rcvr_sync -
 *
```

```
 *      receiver synchronous process
 */

process rcvr_sync(rxd, drdy, valid,
          mode, control, status,
          sync1, sync2)
      in boolean rxd;       /* receive serial */
      in boolean drdy;      /* data ready */
      in boolean valid;
      in boolean mode[DataSize];
      in boolean control[DataSize];
      in boolean sync1[DataSize];
      in boolean sync2[DataSize];
      out boolean status[DataSize];
{
      boolean sync_mode;
      boolean par;
      boolean ncount[3];
      boolean data[DataSize];

      /*
       *      free up line
       */

      free status;

      /*
       *      determine initialization
       */

      if ( valid ) {

            sync_mode = (mode[6:7] == 0);

            if ( sync_mode ) [

                  /*
                   *      wait for mode
                   */
                  if ( eh )
                        hunt_mode(rxd, drdy,
                           sync1, sync2, mode );

                  /*
```

```
                 *      start shifting data in
                 */
                ncount[2:2] = 1;
                ncount[0:1] = nbits;
                while ( ncount ) [
                      wait ( drdy );
                      data[7:7] = read ( rxd );
                      data = data >> 1;
                      ncount = ncount - 1;
                ]

                /*
                 *      send data to main process
                 */
                send( i8251, data );

          ]

          write status;
    }
}


/*
 *      rcvr_async -
 *
 *      receiver asynchronous process
 */

process rcvr_async(rxd, drdy, valid,
      mode, control, status)
      in boolean rxd;       /* receive serial data */
      in boolean drdy;       /* data ready */
      in boolean valid;
      in boolean mode[DataSize];
      in boolean control[DataSize];
      out boolean status[DataSize];
{
      int i;
      boolean sync_mode;
      boolean par;
      boolean ncount[3];
      boolean data[DataSize];
```

```
/*
 *      determine initialization
 */

free status;
if ( valid ) {

        /* assume mode is stable now */

        sync_mode = (mode[6:7] == 0);

        if ( ! sync_mode ) [

                /*
                 *      wait for start bit
                 */
                wait ( rxd );
                wait ( ! rxd );

                /*
                 *      start shifting data in
                 */
                ncount[2:2] = 1;
                ncount[0:1] = nbits;
                while ( ncount ) [
                        wait ( drdy );
                        data[7:7] = read ( rxd );
                        data = data >> 1;
                        ncount = ncount - 1;
                ]

                /*
                 *      sample parity bit
                 */
                if ( pen ) [

                        par = data[0:0];
                        for i = 1 to 7 do
                                par = par xor data[i:i];

                        if ( ep )
                                par = ! par;
                        wait ( drdy );
                        if ( par != rxd ) {
```

```
                        /* parity error */
                        pe = 1;
                        write pe;
                    }
                ]

                /*
                 *      sample stop bit
                 */
                wait ( drdy );
                if ( rxd == 0 ) {
                    /* framing error */
                    fe = 1;
                    write fe;
                }

                /*
                 *      send data to main process
                 */
                send( i8251, data );
            ]

            write status;
        }
}


/*
 *      main process for intel 8251
 */

process i8251(ChipSelect, WriteEnable,
   ReadEnable, ChipData, data, valid,
   sync1, sync2, mode, control, status)
      in boolean ChipSelect;
      in boolean WriteEnable;
      in boolean ReadEnable;
      in boolean ChipData;
      inout boolean data[DataSize];
      out boolean valid;
      out boolean sync1[DataSize];
      out boolean sync2[DataSize];
      out boolean mode[DataSize];
```

```
out boolean control[DataSize];
in boolean status[DataSize];
[
    boolean modebuf[DataSize];
    boolean dbuf[DataSize];
    boolean decode[3];
    boolean sync_mode;

    valid = 0;
    write valid;

    /*
     *      reset sequence: read mode character
     */
    wait ( ChipSelect & WriteEnable & ChipData );

    modebuf = read ( data );

    mode = modebuf;
    valid = 1;

    sync_mode = (modebuf[6:7] == 0);

    /*
     *      read sync characters if necessary
     */

    sync1 = 0;
    sync2 = 0;

    if ( sync_mode )
    [       /* read first sync char */
            wait (ChipSelect&WriteEnable&ChipData);
            sync1 = read ( data );

            /* read second sync char */
            if ( ! modebuf[7:7] )
            [       wait (ChipSelect&WriteEnable&ChipData);
                    sync2 = read ( data );
            ]
    ]

    /* write to output port */
    write valid, mode, sync1, sync2;
```

```
/*
 *      main interp loop
 */

decode[0:0] = ChipData;
decode[1:1] = ReadEnable;
decode[2:2] = WriteEnable;

while ( ChipSelect )
{       switch (decode) {
        case 0x2:           /* read data */
                [
                if (sync_mode)
                        receive(rcvr_sync, dbuf);
                else
                        receive(rcvr_async, dbuf);

                wait ( ! WriteEnable );
                data = dbuf;
                write data;
                ]
                break;
        case 0x3:           /* read status */
                [
                data = read ( status );
                wait ( ! WriteEnable );
                write data;
                ]
                break;
        case 0xC:           /* write data */
                dbuf = read ( data );
                send(xmit, dbuf);
                break;
        case 0xD:           /* write control */
                dbuf = read ( data );
                control = dbuf;
                write control;
                break;
        }
}

]
```

References

[1] David C. Ku, G. De Micheli, *HERCULES - A System for High-Level Synthesis* Proceedings of the 25th ACM/IEEE Design Automation Conference, Anaheim, 1988.

[2] David C. Ku, G. De Micheli, *Using the HERCULES High-Level Synthesis System* Internal report, 1988

[3] Frederic Mailhot, G. De Micheli, *Structural/Logic Intermediate Form Specification* Internal report, 1988

[4] Robert Alverson, Tom Blank, et. al., *THOR User's Manual: Tutorial and Commands* Stanford Technical Report CSL-TR-88-348, January, 1988

accepted for publication in
Journal of Systems and Software, 1989

The *Hermod* Behavioral Synthesis System

**Masayasu Odani, Sun Young Hwang, Tom Blank,
and Tom Rokicki**

**Center for Integrated Systems
Stanford University**

Abstract

Hermod is an interactive behavioral synthesis program developed at Stanford University. Using a combined control and data flow graph (C/DFG) as an intermediate representation, Hermod generates functional blocks and their interconnection from behavioral descriptions. Hermod supports a menu-driven interface, displaying the control and data flow graph with a set of legitimate *timing-cuts* and its hardware representation. Emphasizing user participation, the system allows the user to control state partitioning and resource sharing through a graphical interface to explore the maximal design space. Written in an object-oriented language C++, Hermod generates a hardware representation in several minutes from a behavioral description of practical size on a VAXstation II/GPX.

Indexing Terms: behavioral synthesis, structural synthesis, control and data flow graph, register-transfer level description, design space exploration.

* Note: Revised Copy for "Journal of Systems and Software".
-- 8 June 1988

1. Introduction

Silicon compilation is the process of automatically mapping an abstract design representation to a physical structure [19]. Depending upon the input language, silicon compilers are classified into behavioral compilers (or behavioral synthesizers) and structural compilers. A behavioral synthesizer translates a behavioral description into a structure, creating structural designs consisting of functional blocks and their interconnection. In a behavioral synthesis system, the design is specified by a functional relationship between input and output ports described in a hardware description language [4, 11, 26]. The behavior of output ports is specified in terms of input ports and internal state. The output from a behavioral synthesizer contains hardware modules[1] (*data paths*) required to implement the given behavioral specification, and their scheduling (*control*).

The synthesis task includes generation of data paths and their control blocks. Data path synthesis consists of the following subtasks: module bindings, state bindings (control step partitioning), and register and connection bindings. In the module binding process, a functional module is assigned to each abstract operation, and a register is assigned to data carried across state transitions. The functional modules (or registers) can be shared among more than one abstract operation (or variable). Further, the binding process relies on library, which may contain structural modules at several abstraction levels [9]. The module binding process has a great impact on the system cost and performance, since an abstract operation can be realized in many ways. The module binding process is performed before control logic synthesis, even though the process can be iterated later if imposed constraints are not met. State binding implies assignment of each operation to a machine state. Machine states are created depending on the clock period of the system and the delay of hardware modules. Based on the clock period and the number of maximum allowed states, the system assigns each abstract operation to a machine state such that maximum parallelism can be achieved within available hardware modules. Connection bindings imply the interconnection between functional modules and registers, creating the data transfer paths between them. Connections can be implemented using bus structure or multiplexors.

Control synthesis creates the finite state machine that controls the data path units for the proper execution of code sequences. The goal is to define the sequence of the micro-operations and the timing of the control signals to the data path. To generate the control block, the data path must be fully defined, and the required operations must be specified as a linear ordered list of micro-operations which affect either the control flow or data path. Two different design styles have been used for control block implementation: structured and

[1] In this paper, a functional module represents a hardware block which can execute an operation like addition and subtraction, while a hardware module is used to represent a functional module, register, or wire.

custom implementations. In a structured implementation, the next state information and signals for data path selection are encoded in a structured array. This implementation is flexible and easy to modify. A custom implementation uses random logic for efficiency, exploiting the particular features of the data path. Detailed description of various algorithms for control synthesis can be found in [7, 14, 18, 24].

Automatic synthesis from behavioral specifications is an exploratory area for design automation. A number of behavioral compilers have been reported that generate structural descriptions from behavioral level descriptions [6, 8, 14, 16, 17, 20, 23, 25]. Those systems automatically generate a structural description from a behavioral description using the design constraints given by a designer. However, supported design styles are limited in most systems, giving few chances for the designer to change what the system generates. Thus, designer may feel alienated from the system with no choices other than to accept the machine-generated designs, even if he is not satisfied with the output.

This paper describes the Hermod behavioral synthesis program that gives a hardware designer full system control to select the design style he likes through a menu-driven graphical interface. The system is not intended for use with design descriptions which would require thousands of components for realization, but for designs at high abstraction levels where design space exploration is of primary concern in the early stage of design. The Hermod program is included in an integrated environment for hardware simulation and synthesis under development at Stanford University. The functional models written in a behavioral description language *ILSP* [15] can be simulated on the *THOR* logic/functional/behavioral simulator [2] without translation. The behavioral models that have been verified through the THOR simulator are input to Hermod to generate RTL descriptions, which can be again simulated by THOR simulator for verification purposes. Efficient algorithms are incorporated in Hermod for generating timing-cuts for state partitioning, checking consistency after modifications to the system-generated designs, and optimizing through resynthesis.

2. Input Language and Internal Representation

2.1 ILSP: Behavioral Description Language for Hermod

ILSP (Input Language for Synthesis Program) is used to describe the function or behavior of the hardware to be designed. Based on the C-language, ILSP has conditional (*if* and *switch*), and loop (*while-* and *do-loop*) control constructs, and allows explicit specification of the actual hardware interface to the outside world. Many features of the C language considered redundant or unnecessary for behavioral representation of a hardware module are omitted in ILSP. For instance, only integer type variables are supported, and parameter passing is handled through interface declarations. Compared to the ISPS (Instruction Set Processor Specifications) [4] which describes the behavior and structure of the design at register-transfer and behavioral levels, the ILSP description is purely procedural. The features of ILSP are briefly reviewed next. A detailed description can be found in [15].

Signal Declarations: Three fundamental object types are supported. The objects declared as integers are local variables to the procedure in which they are declared. The objects declared in the signal declaration sections (*IN_LIST*, *OUT_LIST* and *ST_LIST* sections) as signals (*SIG*) are one-bit-signal variables and those declared as groups (*GRP* or *BUS*) are multiple-bit-signal variables. The SIG- and GRP-type objects are the abstract representations of *registers* in hardware realization. An integer object may be realized by a register, or just as a wire depending on its usage and lifetime.

Control Constructs: Most control constructs in the C language are supported: conditional statements (if- and switch-statements) and loop (while- and do-loop) constructs.

Expressions and Statements: Expressions and statements supported in ILSP are a subset of those in the C language. Arrays of complex data structure (like arrays of structure with several data fields) are not supported. Array structure is allowed only to represent a group of signals, which will be realized by registers or memory modules. The differences from the C language in expressions and statements are summarized as follows:

- Structures and pointers are not allowed. An exception is that a pointer is passed to a subprocedure as an argument for a GRP-type object in a procedure call.
- Array structures are used to specify bit position for group signals. A GRP-type object followed by a range in square brackets specifies a portion of group signals. The expression x[] implies the entire signal group of x will be treated as an integer. The expression x[3] represents the signal value of the third bit of x, and x[7:4] means the partial signal group between the seventh bit and fourth bit of x treated as an integer.
- Increment expressions (++ and --) are allowed for integer variables only.

- Procedure calls that return one or more values are supported. The *"(receiver-list) = procedure-call-expression;"* form is used for procedure calls that return multiple outputs and distribute the values to the variables and group signals in the *receiver-list*.
- A *break* statement is allowed only in a switch statement to eliminate abrupt loop exits.
- Parameter passing in a procedure call is handled through the hardware interface mechanism. That is, the input and output parameters are declared in the interface declaration sections (*IN_LIST* and *OUT_LIST* declarations), unlike C procedures.
- A return statement is allowed only at the end of procedure. No expressions are allowed after a return statement. Instead, a procedure can return values by assigning values to the variables declared in the *OUT_LIST* section.

Figure 1 shows a procedure describing the design that takes two groups of signals, in and enable, and calculates the summation of the value of in, setting the signal group out and signal line valid. The objects declared in the *IN_LIST* section represent input signals or ports. Enable represents one signal line, and in represents a group of signals consisting of 8 signal lines. Likewise the objects in the *OUT_LIST* section represent output signals or ports set by the procedure. The statement "r = in[]" means that the signals grouped as in are packed into an integer (r). The statement "out[] = s" sets the group of output signals, out, by unpacking the integer (s).

2.2 Graph Representation

In the behavioral synthesis process, a behavioral representation is translated into an intermediate representation in graph form, which is subsequently transformed and translated into a structural description. In Hermod, a graph representation is chosen that reflects both the control sequencing and the data flow in the program. In the graph, a node can represent a data carrier, an abstract operation, or a control construct. An edge can be a control edge representing control sequencing in the behavioral description, or a data edge representing data usage or data flow depending on the types of the nodes connected by it. The graph consists of several data flow subgraphs corresponding to basic blocks of the behavioral description, each of which consists of straight line codes and control nodes connecting them. The graph shows not only the dependency or parallelism of each operation but also the global control and data flow in the model. The C/DFG allows hierarchical design by incorporating a procedure-call node. A procedure-call node representing some hardware block can be specified by another graph. This graph representation is similar to the McFarland's Value Trace [12]. However, unlike VT, nested loop constructs are used in the representation, which is a natural way to handle loops.

There are five types of nodes: data, constant, operation, control, and temporary nodes.

```
sum()
{
    IN_LIST                     /* declare input ports */
        SIG( enable );
        GRP( in, 8 );
    ENDLIST;

    OUT_LIST                    /* declare output ports */
        SIG( valid );
        GRP( out, 16 );
    ENDLIST;

    int r, s;                   /* declare local variables */

    valid = 0;
    if ( enable ) {

        r = in[];
        s = 0;
        while (r >= 0) {
            s = s + r;
            r --;
        }

        out[] = s;
        valid = 1;              /* set the flag */
    }
    return;
}
```

Figure 1: A behavioral model calculating the summation of a given integer.

Each node has one or more input ports and output ports. Each port is identified by its io-attribute and port-id. Data, constant, temporary, and operation nodes (together with directed edges into and out of the nodes) reflect the data flow and data dependencies, while control nodes are used for control sequencing and to mark the boundaries of basic blocks.

- *Data/Constant/Temporary Nodes*: A data (or constant) node corresponds to an object (variable or constant) in the behavioral description. Basically, for each appearance of a variable, there is a corresponding data node in the graph. Temporary nodes are used to represent the data produced by an operation and used by another operation or control node.
- *Operation Nodes*: An operation node corresponds to an abstract operation in the description. A procedure call is considered an operation with multiple inputs and outputs.
- *Control Nodes*: A control node shows the beginning and end of a basic block. There are seven different control nodes: start, end, fork (for if-statements), sfork (for switch statements), join, loop, and loop-end nodes.

An ILSP procedure consists of three types of blocks: straight line code, conditional statements, and while- and do-loops.

Straight Line Code: A block of straight line code consists of expressions and assignments. Expressions are realized in such a way that operation nodes take edges from operand data nodes. Assignment is normally realized by the edge from an operation node to a data node, which means that the result of the operation is stored in the variable represented by the data node. Temporary data nodes are inserted if necessary. Retrieving data from or assigning values to a subset of group signals is allowed. To construct the subgraph showing such a partial retrieval or assignment, the *fpack* and *funpack*[2] procedure-call nodes are used as shown in Figure 2.

---

[2]These are intrinsic library functions in the THOR simulation system [3]. *Fpack* converts a group signal of specified bit width into an integer and *funpack* converts an integer into a group signal.

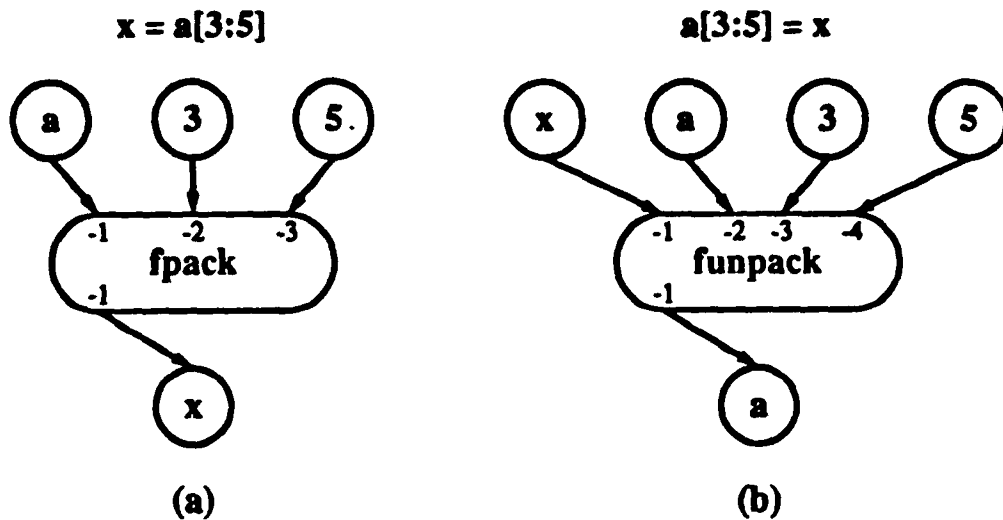


Figure 2: Graph representation for the functions (a) *fpack* (b) *funpack*.

Conditional Statements: The if block consists of two sub-blocks corresponding to the then-part and else-part, respectively. The then-part (else-part) sub-block starts from the *TRUE* (*FALSE*) port of a fork node and ends at a join node. The subgraph corresponding to the conditional expression is connected to the *CONTROL* port of the fork node.

While/Do Loop Constructs: A while-loop subgraph is also constructed with join and fork nodes. In this subgraph, the join node is followed by the subgraph corresponding to the conditional expression of the while statement, which is connected to the *CONTROL* port of the fork node. The subgraph of the while-loop body starts from the *TRUE* port of the fork node, and ends at the join node through *BACK* edges to show the iterative nature of the block. A do-loop subgraph consists of one block which starts from a loop node and ends at a loop-end node. The output of the conditional expression is connected to the *CONTROL* port of the loop-end node. The loop-end node has two output ports: the *TRUE* port connected to the loop node by a *BACK* edge, and the *FALSE* port from which a new basic block begins after the do-loop statement.

Figure 3 shows the graph representation of the procedure shown in Figure 1. Rectangles represent operation nodes that will be mapped into structural components in the module binding process, and circles represent variables and constants (data nodes). Control nodes are represented by trapezoids (fork and join nodes) or hexagons (start and end nodes). The outer fork-join node pair corresponds to the if-statement that is controlled by the value of enable. The inner join-fork pair corresponds to the while-loop statement. The control input to the fork node is from the condition expression.

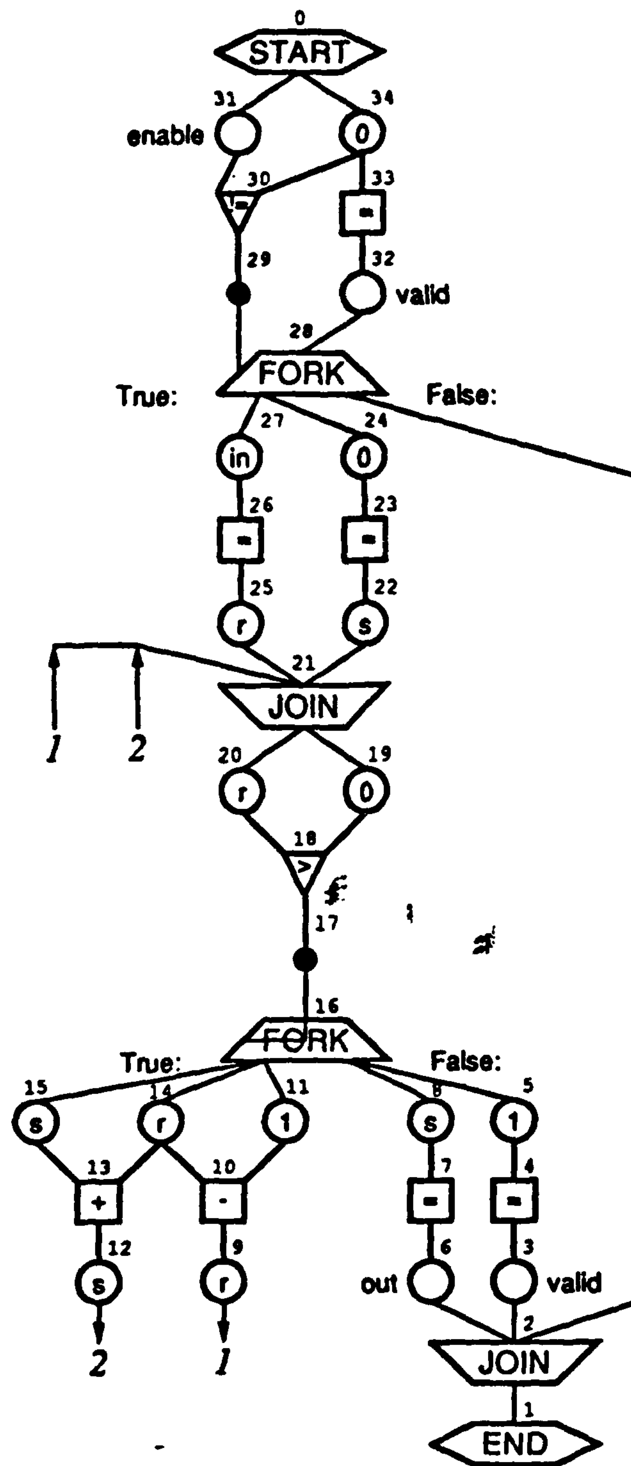


Figure 3: Graph representation of the procedure of Figure 1, generated by Hermod.

3. The Synthesis Process in Hermod

3.1 System Overview

Figure 4 shows an overall picture of Hermod. Taking a procedure describing the desired behavior of a design, Hermod converts it into the intermediate graph form (C/DFG), then builds the hardware realizing the behavior in two passes. In the first pass, the operations in the graph are assigned to machine states and the initial hardware is created by allocating functional modules to the operation nodes, and wires or registers to the arcs. Two graphs are produced as a result of the first pass. One is a data path graph that consists of nodes representing functional modules or data storage and edges representing interconnections. The other is a state transition diagram in which each node corresponds to a machine state and each edge shows a state transition and its conditions. In the second pass, functional modules and registers are merged, if they have no usage conflicts. Those optimization processes can be performed automatically by the system using the information on number of available modules and their functionality, or can be directed by the user through a graphical interface. As a final result, Hermod produces a netlist of the created data path and a control unit specification in the truth table format (π format) for PLA descriptions [5].

3.2 Initial Synthesis Process

3.2.1 Functional Module Binding

The module binding process transforms the functional block representation provided by the data path allocator to a hardware-bound level of description [9]. In Hermod, the module binding process is embedded in data path synthesis in the initial synthesis process. During initial synthesis, no restrictions are imposed on the number of hardware modules used. The system assigns hardware modules to each node and timing-cut edge. Hardware modules are selected from a module library. Module selection specifies the type, functionality, and other attributes (such as delay, area, control setting, and io-ports) for each abstract operation. A dedicated module is assigned to each abstract operator during the initial synthesis phase. This can exploit the parallelism inherent in the original behavioral description. However, it would be desirable to share functional modules among operators to reduce the overall hardware requirements for implementation. In Hermod, the resource sharing is handles during optimization phase.

3.2.2 Control Step Partitioning and State Binding

When the C/DFG is generated, only control and data dependencies are represented in the graph. The state binding process partitions the graph into machine states. Each operation node in the graph is assigned to a particular state. Thus state binding determines the parallelism of the generated design.

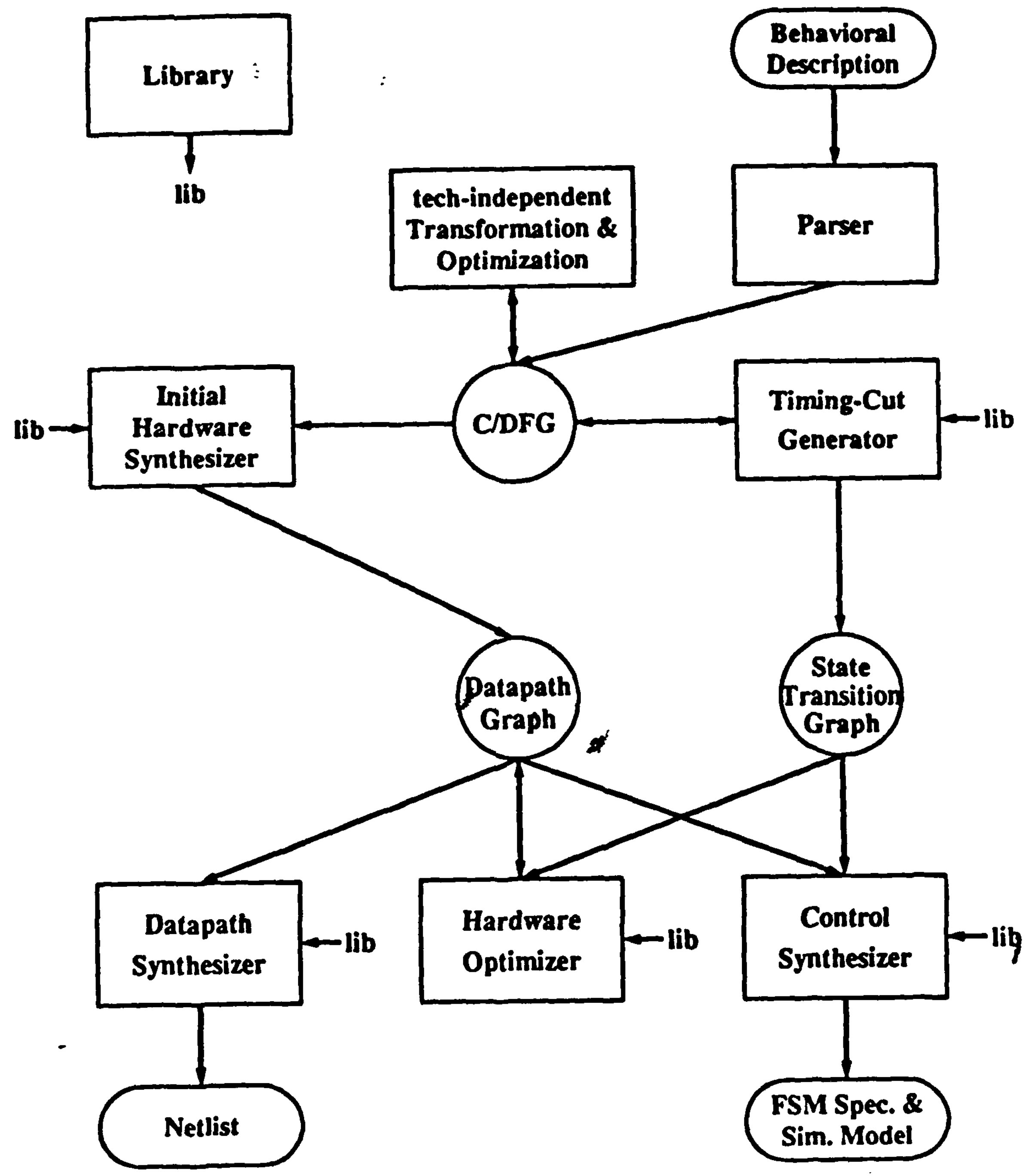


Figure 4: System overview.

A. *Automatic Operation Scheduling Process*

Timing-cuts are used in Hermòd for control step partitioning. A timing-cut is a set of edges that forms a cut-set of the subgraph corresponding to a basic block of code. A set of timing-cuts divides the graph into several data flow subgraphs each of which forms a machine state. For example, refer to Figure 5. The graph is divided into five subgraphs. Note that states 2 and 4 overlap. It is due to the loop construct in the graph. At state 2, values are assigned to the symbols r and s, which occurs when entering the while-loop. At State 4, condition variable is checked after each iteration of the loop. Timing-cuts are generated depending on the system clock period and the module delay. In this process, Hermod employs the *as soon as possible* (ASAP) scheduling algorithm that schedules each operation in a greedy fashion [16, 25]. It schedules as many operations as possible in each machine state as long as the delay does not exceed the clock period. Timing-cuts are inserted in the following two cases;

- When the delay exceeds the clock period. (Hermod supports chaining and multi-cycling [16].)
- In front of a fork/sfork node (starting node of if/switch statements). In a conditional statement, only one of the branches is executed in the next machine state. To determine which branch should be taken, the previous state must end at the fork/sfork node regardless of the execution delay. This also prevents the race condition of the loop constructs.

The automatic timing-cut generation process is based on the longest path search. Starting from the start node of the graph, the system calculates the maximum execution delay for each node reachable from the start node until a fork/sfork node or end node is encountered. Then, each node on the path is assigned to a particular machine state according to their maximum delay. The edges connecting operation nodes in different machine states form a timing-cut. When a fork/sfork node is encountered during the search, a timing-cut is created consisting of the edges connected to the node. Then, all the branches from the fork/sfork node are put in a *branch queue*. While the branch queue is not empty, the process retrieves the first entry of the queue and resumes the search. The search continues until another timing-cut or fork/sfork node is encountered. The state binding process determines the values to be stored for use in the next control steps. In the hardware implementation, latches are inserted in the data transfer paths where timing-cuts are generated.

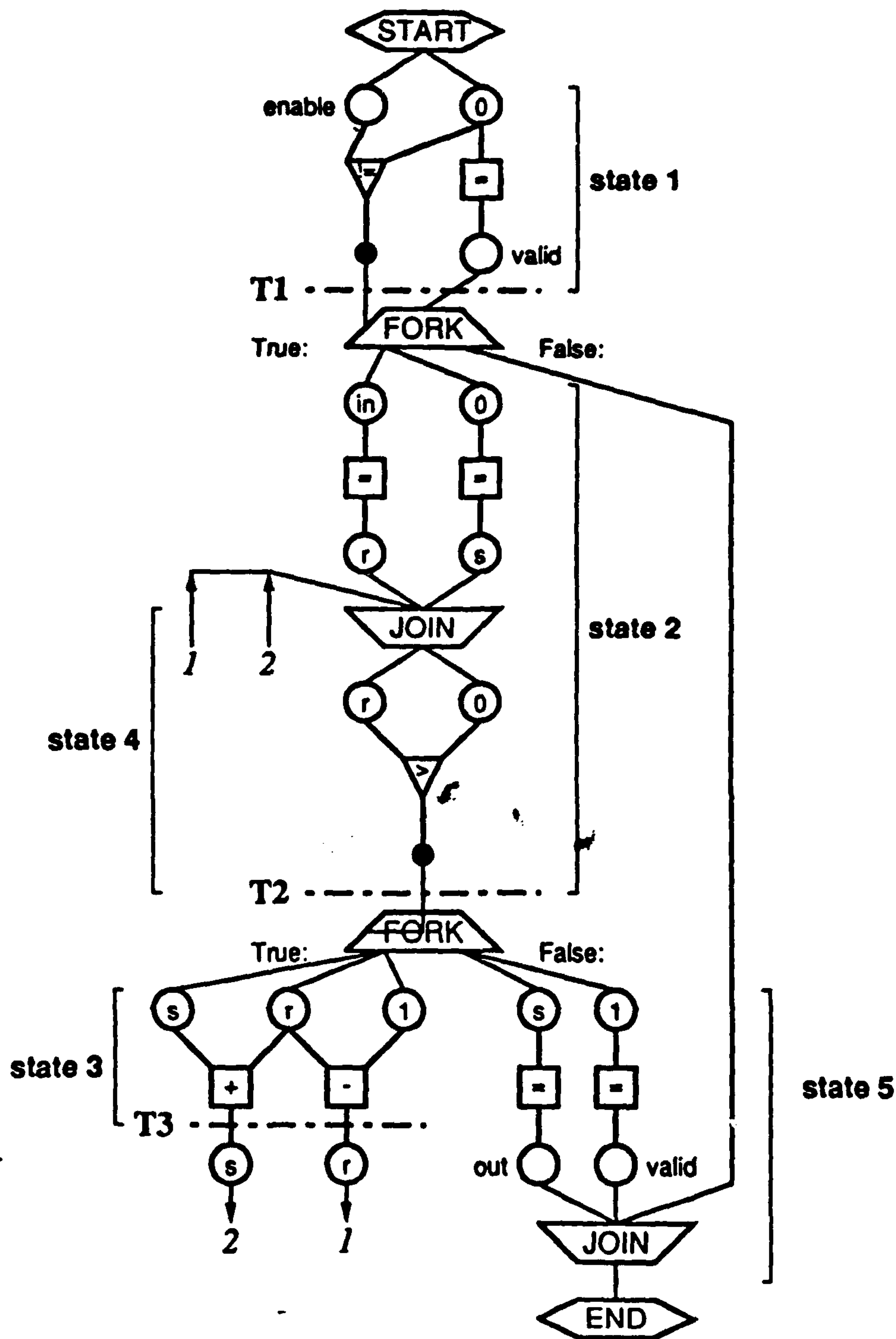


Figure 5: The intermediate graph representation with timing-cuts.

B. Manual Timing-cut Modifications

To maximally utilize hardware resources, modifications are allowed on the C/DFG such as adding, deleting, and moving timing-cuts. Through a graphical interaction tool, the graph is displayed in a window and the user is allowed to pick up the edges to construct a new timing-cut, or pick up an existing timing-cut to add, delete, or move it by clicking the mouse. If the user defines a new set of edges as a timing-cut, the system checks if those edges forms the cut-set of a basic block subgraph. If the changes are legal, that is, they don't violate the behavioral specifications and design constraints, the system accepts the changes. When the user deletes or moves some timing-cuts, the clock period may be changed (become longer) and the execution delay of each state is recalculated. If the maximum delay exceeds the clock period in any state, Hermod asks the user if the clock period may be increased. Those modifications may change the state binding of some operations. Once the modifications are accepted, the system generates a new state transition diagram based on the new state binding, and the new data path.

C. Example

Figure 6 shows the data path for the behavioral description in Figure 1. It is generated using the state binding of Figure 5. The design is obtained by setting the clock period to be the module delay (all the functional modules used here have the same delay). It consists of five machine states, and uses four dedicated functional modules. The registers available in the module library have only reset and load control inputs.

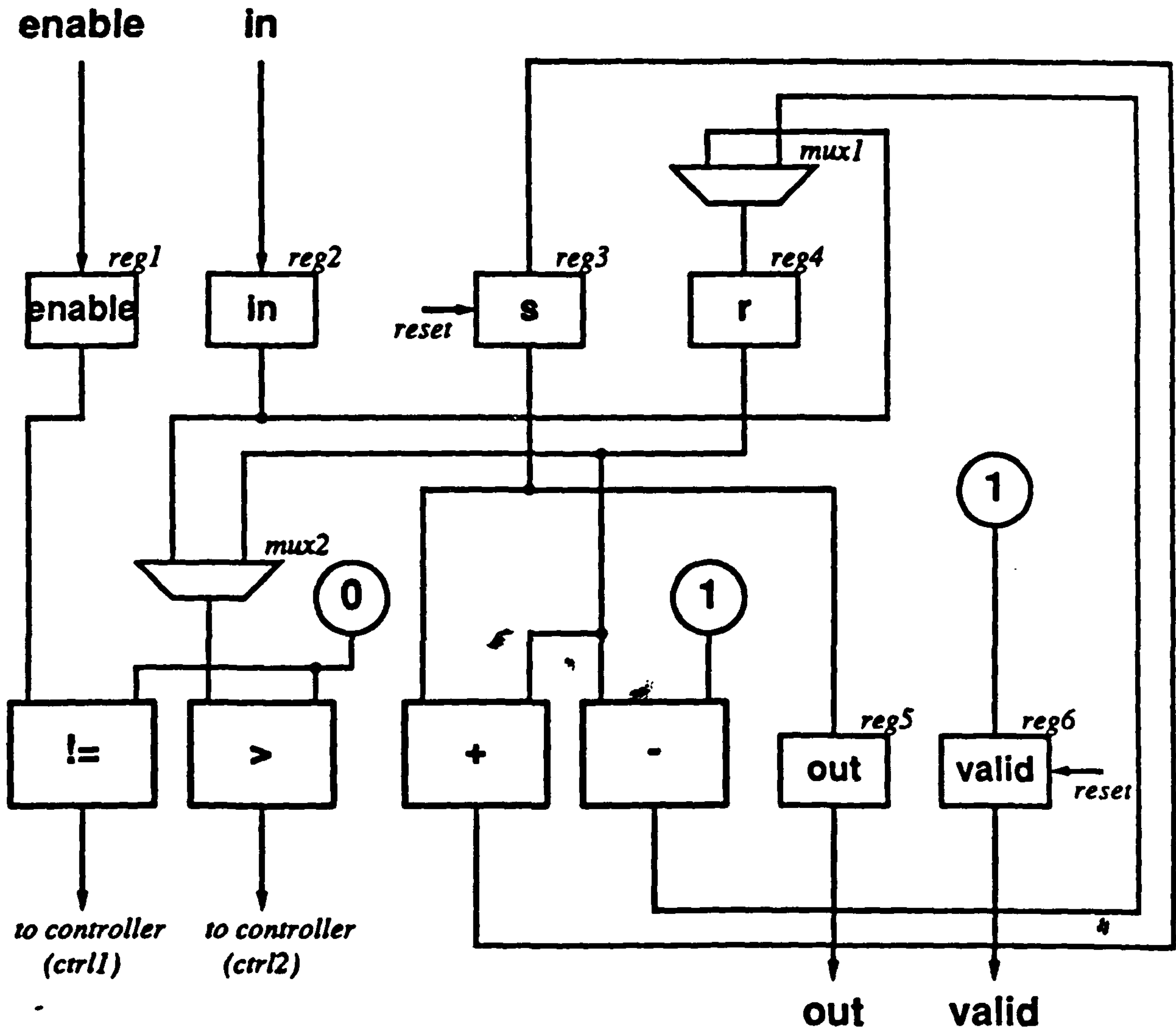


Figure 6: A hardware representation of the graph in Figure 3.

3.2.3 Register Binding

Registers are assigned to store the temporary results generated by the operators in a state when these results are used in the following states. The system assigns a register to each edge belonging to a timing-cut, unless the edge comes from a constant node. The system also allocates registers to the data nodes connected to timing-cut edges. The registers allocated to the same variable are merged into one in the optimization phase. Caution must be taken when the same variable appears more than once in a timing-cut edge. For example, in Figure 7, edges *e2* and *e3* are connected to the data nodes representing the same variable *a*. Since the data node connected to the edge *e3* is the last definition of the variable *a* in that state, the register associated with the edge *e3* must be associated with the variable *a*. The register associated with the edge *e2* becomes the temporary register. In order to deal with this problem, the register binding routine first determines the last definition node for each variable in each state. Then it assigns a temporary register to the timing-cut edge connected to the data node which is not the last definition node. In the optimization pass, registers allocated to different symbols that have non-overlapping lifetime are merged.

3.2.4 Connection Binding

Allocation of hardware resources (functional modules and registers) implies that there exist physical paths among them. These paths can be obtained by analyzing the paths between the abstract operations. The connection binding routine analyzes the connections of each machine state one by one. First, the connection binding routine extracts the data flow subgraph corresponding to the currently processed machine state. Then, it creates the connections between the functional modules and registers by tracing the edges connected to the operation and data nodes. If an operation uses inputs or produces outputs across the state boundary (e.g, timing-cut), a connection wire is created between the functional modules corresponding to the operation nodes and the register corresponding to the timing-cut edge. Each time a new connection wire is created, the system encodes the currently processed state into the wire data structure for later use during the control generation process.

To allow the maximum sharing of functional modules and registers, multiplexors/busses are created and inserted wherever necessary to transfer data between operands (registers) and operators (functional modules).

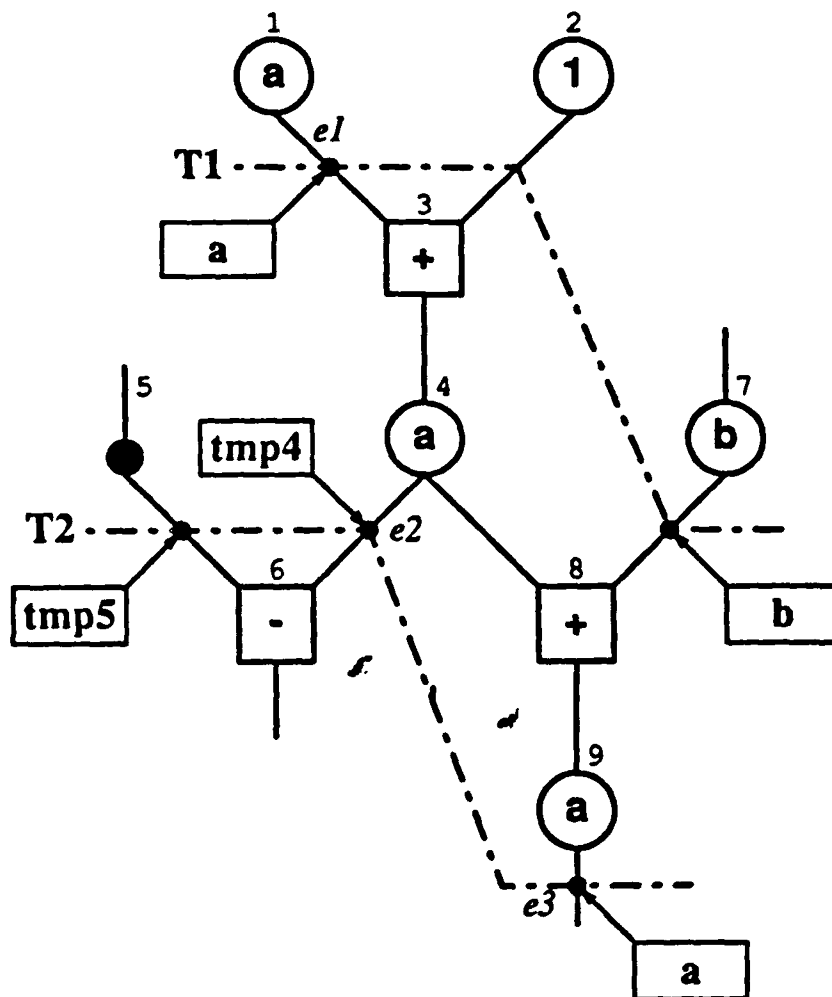


Figure 7: Register binding for timing-cuts.

3.2.5 Control Generation

Control structure is not specified in the behavioral description. Instead the control block is automatically generated from the description using the information on available resources (library modules) used in the data path synthesis process. The control block can be generated at the same time as the data path. However, it is straightforward to generate the control block after the data path structure is determined, since the timing and sequencing of the control signals are embedded in the state transition diagram and the data path graph. In other words, the control signals are determined by the state binding and resource allocation. The result of data path synthesis (module/state/connection bindings) is a symbolic microcode for control generation. The control model in Hermod is a finite state machine followed by an encoder for generation of control signals for particular data path modules. A finite state machine for the control of the data path in Figure 6 is specified in *it* format [5] in Figure 8.

3.3 The Design Optimization Process

Although the hardware built in the initial synthesis pass realizes the behavior of a given description and satisfies the user-defined timing constraints, it may use more hardware resources than necessary, increasing the wiring and routing complexity. Basically, two hardware modules (functional modules, registers, or connection wires) can be merged if there are no usage conflicts among them in any machine state. Several hardware optimization procedures have been proposed based on the above principle [25]. In Hermod, optimization is performed separately for functional modules and registers in a pre-defined order. Although reducing the number of those hardware resources is the main concern of the optimization process, the numbers of interconnections (nets) and multiplexers should also be taken into account, because those interconnections usually take a large portion of the realized chip area [13]. The final optimization results including the interconnections and multiplexers depend heavily on the execution order and technique of each optimization process.

The Hermod hardware optimizer takes two inputs generated during initial synthesis, the data path graph and the state transition diagram, and produces an optimized data path. The optimizer consists of a rebinding process for each type of hardware resources: functional modules, registers, and connection wires. (The connection rebinding process is currently under development.) The rebinding processes provide menu-driven graphical user interaction tools so that the user can specify the pre-bindings for several hardware modules. The user can also unbind the old module bindings partially or totally and re-optimize the data path using the remaining bindings. The user is allowed to pick functional modules or registers by clicking mouse on them to force them to be merged or split. The user-directed bindings are checked if they have usage conflicts. Once a rebinding process is done, the system reconstructs the data path using the new binding information to rebuild the data path.

```
.i 5
.o 19
.ibl s2 s1 s0 ctrl1 ctrl2
.ol  s2 s1 s0 reg1_l reg1_r reg2_l reg2_r reg3_l reg3_r
reg4_l reg4_r reg5_l reg5_r reg6_l reg6_r mux1_ctrl mux2_ctrl

000--    0011010-1-10000--
0010-    0000000-10000-1--
0011-    0100000-10000-1--
010-0    10100001010000000
010-1    01100001010000000
011--    10000001010000011
100-0    101000000000000--
100-1    011000000000000--
101--    000000000001010--
.e
```

Figure 8: Control block in *tt* format for the data path in Figure 6.

The rebuilt data path is displayed on the screen so that the user can evaluate the automatic rebinding results.

3.3.1 Functional Module Rebinding

Two functional modules in a data path can be merged into one if they are not activated simultaneously in any machine state and they can be realized by a library module. (The second condition is not strict, because a library module that can execute those operations may be added later.) Let G_c be the *usage compatibility graph* consisting of nodes representing the functional modules of the data path and edges connecting the functional modules that satisfy the above conditions. Then, finding the minimum number of functional modules necessary to realize the data path is reduced to clique partitioning problem of the graph G_c [25].

Figure 9 shows the functional module rebinding procedure in Hermod. The procedure is based on the cluster development method.

Step 1: Build the usage compatibility graph G_c for the data path graph.

Step 2: Determine the kernel functional modules.

Step 3: While compatibility graph G_c is not empty, do the following:

Step 3.1: Calculate the cost function for each edge of G_c.

Step 3.2: Choose edge *e* with minimal cost.

Step 3.3: Merge two functional modules connected by *e*.

Step 3.4: Update the graph G_c.

Step 4: Generate the new data path.

Figure 9: Functional module rebinding procedure.

In the procedure, *kernel* functional modules are determined first. The kernel functional modules are the modules that construct the maximum independent set of the graph G_c. Let N be the number of the kernel functional modules. Then, N gives the minimum number of the functional modules necessary to realize the data path. Finding the maximum independent set of the given graph is NP-complete [10]. However, the maximum independent set of the graph G_c can be determined using the following heuristics. Suppose the data path is realized by library modules $L_1, L_2, ..., L_m$. First, count the number of functional modules realized by the library module L_i in each machine state. Then, for each library module L_i, find out the machine state S_i that requires the maximum number of functional modules realized by L_i. If more than one state requires the maximum number of the module L_i, then one is chosen

arbitrarily. Finally, collect the functional modules realized by L_i in S_i. The collected functional modules form the maximum independent set of the graph G_c and become the kernel functional modules.

Once the kernel functional modules are determined, the modules other than the kernel modules are merged into one of the kernel modules one by one. The interconnections (wires and multiplexers) are taken into account when functional modules are merged. Among pairs of functional modules that can be merged (connected by edges in the usage compatibility graph), one is selected with minimal cost. The cost function for the edge connecting modules u_i and u_j is

$C(u_i, u_j)$
= c1 * number of multiplexors required to merge u_i and u_j
+ c2 * number of wires added
- c3 * area reduction due to merging of two modules,

where c1, c2, and c3 are constants determined empirically.

After merging a pair of functional modules u_i and u_j, the compatibility graph is updated. If either one of them is a kernel functional module, say u_i, then u_j is removed from the graph. Then, the edge (u_k, u_i) is removed from the graph, unless there was edge (u_k, u_j) in the compatibility graph before merging. Figure 10 shows the results of automatic functional module rebinding on the initial design of Figure 6. Three functional modules are merged into one, and one three-input multiplexor is added due to the merging. This new configuration is reflected when generating the control block for this data path.

3.3.2 Register Rebinding

Two registers can be merged into one if and only if they are not simultaneously *live* at the entry and exit points of any machine state. The register allocation based on this property has been employed in behavioral synthesis systems as well as in optimizing compilers [1]. Tseng and Siewiorek reduced the register optimization problem into the clique partitioning problem [25]. The algorithm is implemented in Hermod supplemented with menu-driven interactive tools. Using the tools, the user is able to interact with the system in the register rebinding process by picking particular registers for merging. Or the user can ask the system to retract a particular merging to further explore the alternatives. Figure 11 shows the results of register rebinding on the partially optimized design of Figure 10. Two registers are saved after register rebinding for this particular example, i.e., registers **in** and **r** are merged, as are **out** and **s** registers.

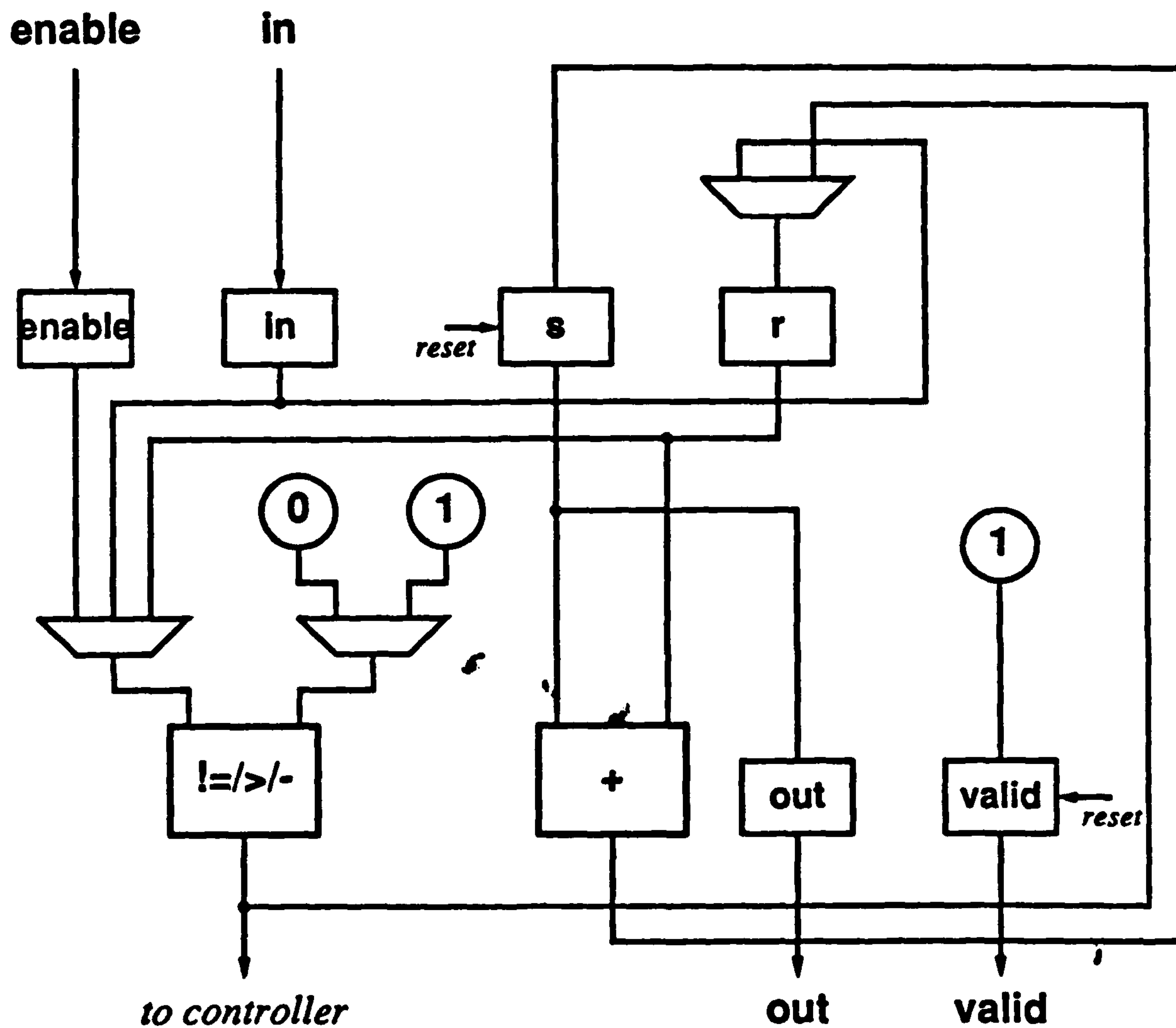


Figure 10: Hardware representation of the procedure in Figure 1 after functional module rebinding.

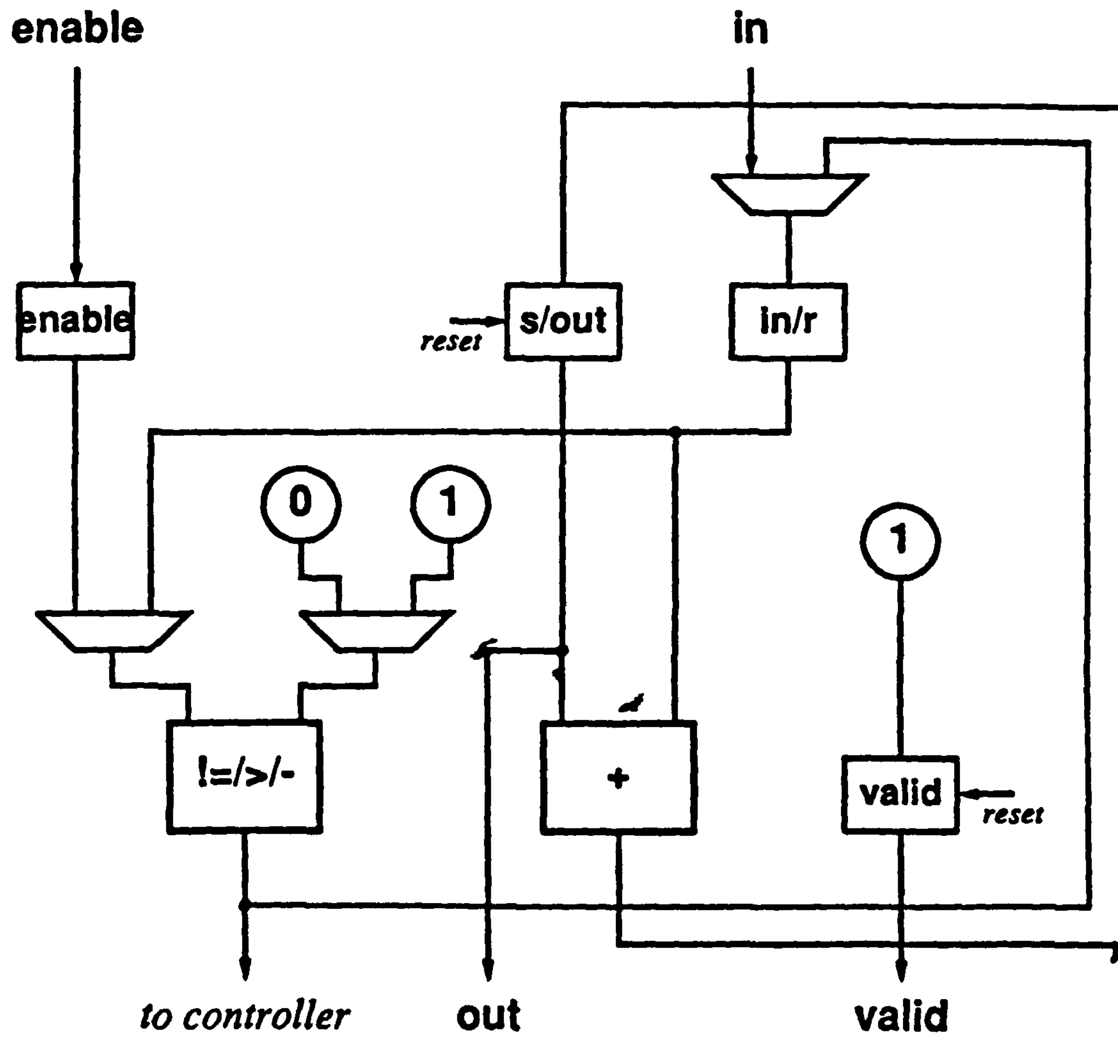


Figure 11: Hardware representation after register rebinding on the data path of Figure 10.

4. Synthesis Examples

The Hermod behavioral synthesis program is implemented in C++, an object-oriented extension of the C language, and runs on a VAXstation II/GPX under UNIX™. This section shows the synthesis results by Hermod for two CPU chips: FRISC microprocessor, a stack-based 16-bit microprocessor [22], and PDP-8 CPU [21].

4.1 FRISC Design

The top-level description of the FRISC processor is given in the Appendix. Here, memory read/write is simulated by the procedures *m_read* and *m_write*. One variable is used as a dummy output of the procedure *m_read*. The data path synthesis routine ignores the dummy variable and no registers are assigned to it, since it is never referenced in the main procedure.

Figure 12 demonstrates the schematic of the data path synthesized by Hermod. The design was done automatically except for slight manual modifications in state bindings to avoid memory access conflicts. In this design, four functional modules (two ALUs, one shifter, and one comparator) and six registers are used. One ALU is used to perform increment and decrement operations for stack pointer S and program counter P. The other ALU performs plus, minus and logical operations. Two registers A and B are used as operand registers for the ALU, while register B is also used as memory buffer. Register I is the instruction register, and register M is used for subroutine calls.

4.2 PDP-8 Design

The second example is taken from the ISP behavioral description of the PDP-8 in [21]. Figure 13 shows the data path generated by Hermod. In this design, several 1-bit inverters and gates are used, as well as 12-bit functional modules such as ALU, shifter, and comparators. Those 1-bit logical operators are employed to realize the expressions of the if-constructs. After initial data path is generated by the system, manual optimization tools are invoked to optimize the 1-bit modules in the design, because better optimization results can be obtained by considering the logical meaning and structure of the circuit. The remaining portions of the data path are optimized automatically. It took less than 20 minutes to finish the data path design using manual and automatic optimization tools in Hermod. Of course, the control FSM is automatically generated.

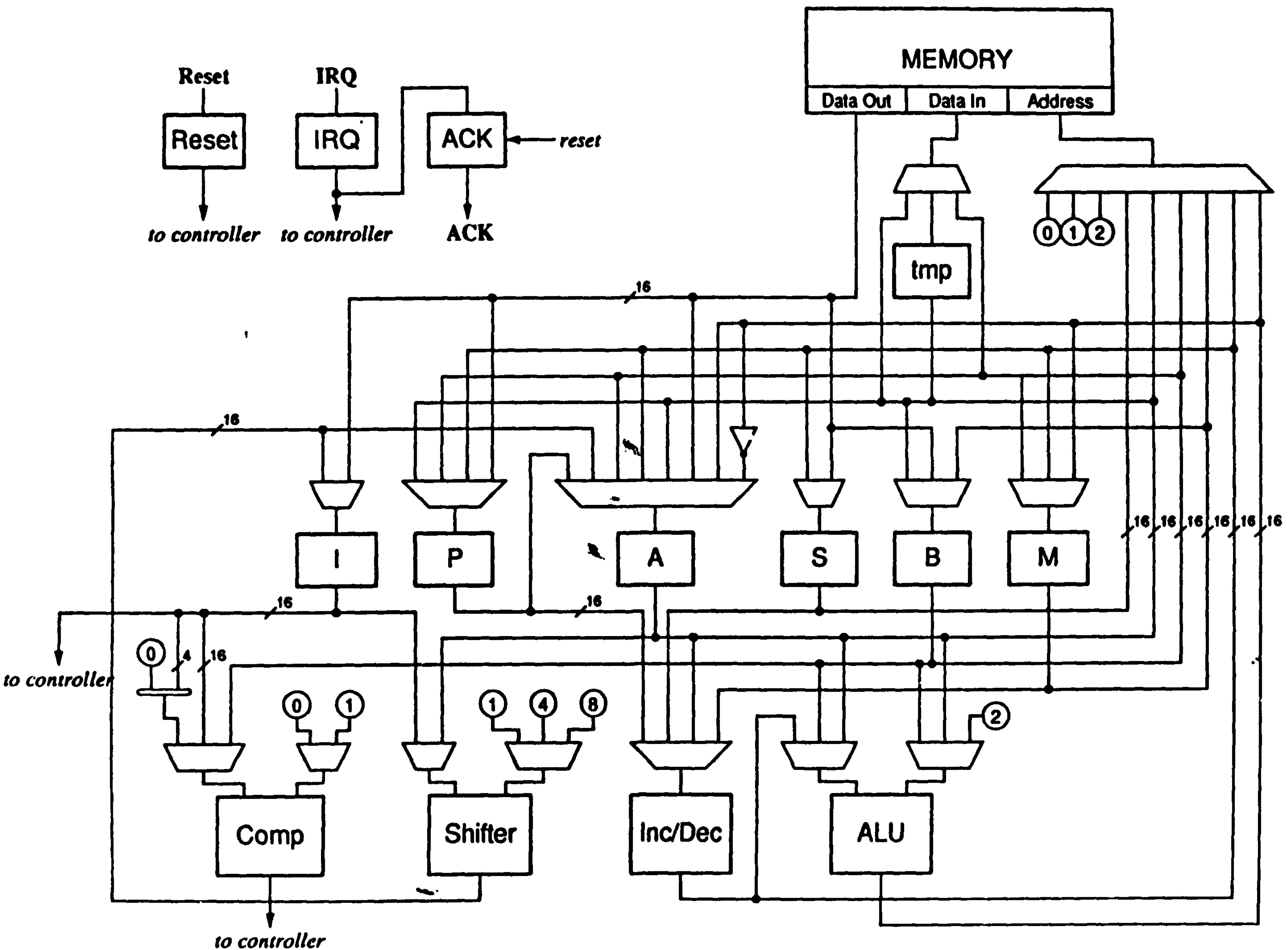


Figure 12: FRISC data path designed automatically by Hermod.

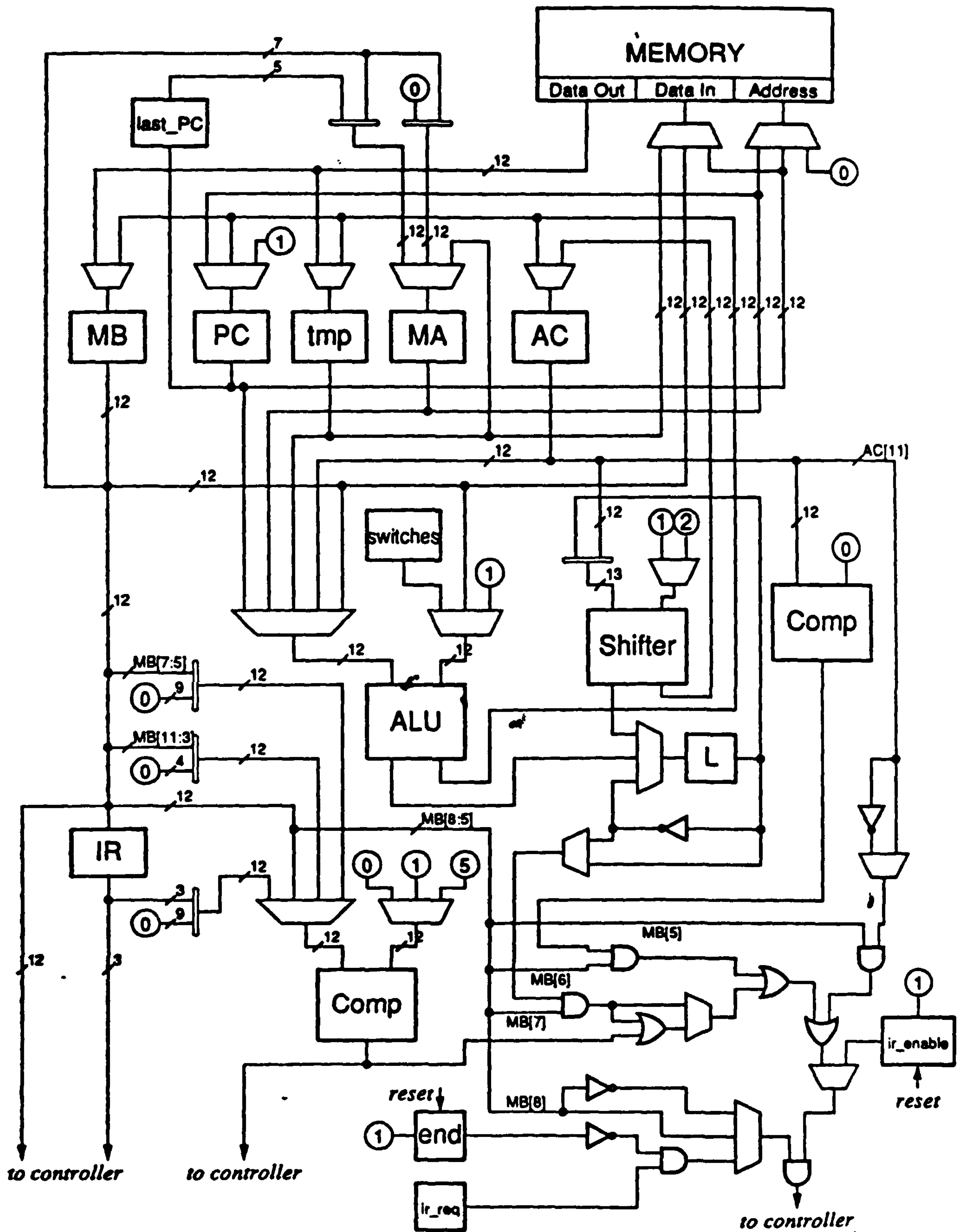


Figure 13: PDP-8 data path designed automatically by Hermod.

5. Concluding Remarks

Hermod is a behavioral synthesis system developed to provide designers an interactive environment within a graphical frame, thus to equip them with tools for direct control of synthesis process. From behavioral descriptions, Hermod generates and displays the control and data flow graph (with a set of legitimate *timing-cuts*) and its hardware representation. Unlike the other synthesis systems, Hermod allows the designer to control the synthesis process in state partitioning and resource sharing through a menu-driven graphical interface to explore the maximal design space. When the designer wants changes in a machine-generated hardware, the system checks the legitimacy of a user's request, then generates a new hardware representation that results from the changes. This system gives designers a clear view of the synthesis process, and suggests a systematic way to create and modify the designs.

Hermod provides a framework for tool development. It is simple to hook up new tools into the system to upgrade its synthesis and verification capabilities. Hermod is not intended for use with design descriptions which would require thousands of components for direct hardware realization. Instead, the system can be effectively used for design descriptions at high abstraction levels in the early stage of design, where (1) the desired behavior is normally described in a hierarchical fashion and (2) design space exploration is of primary concern. Further, it has no predefined data paths, thus does not impose any particular design style for hardware generation.

Although Hermod has some limitations in its capabilities in the current implementation, it can be extended without major modifications in the program. Future extension will include the interconnection hardware optimization and development of more tools for intelligent hardware synthesis.

Acknowledgement

This material is based upon work supported partly by ONR/DARPA under Contract N00014-87-K-0828 and partly by Toshiba Corporation. The authors would like to thank Professor Giovanni De Micheli for his helpful discussion and suggestions. Finally, the authors are grateful to the referees for their valuable comments.

References

[1] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1979.

[2] R. Alverson, T. Blank, K. Choi, S. Y. Hwang, A. Salz, L. Soule, and T. Rokicki, *THOR User's Manual: Tutorial and Commands*, Technical Report CSL-TR-88-348, Stanford University, Stanford, Calif. , January 1988.

[3] R. Alverson, T. Blank, K. Choi, S. Y. Hwang, A. Salz, L. Soule, and T. Rokicki, *THOR User's Manual: Library Functions*, Technical Report CSL-TR-88-349, Stanford University, Stanford, Calif. , January 1988.

[4] M. R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications", *IEEE Trans. Computers*, Vol. C-30, No. 1, January 1981, pp. 24-40.

[5] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, MA, 1984.

[6] R. Camposano, "Synthesis Techniques for Digital Systems Design", in *Proc. 22nd Design Automation Conference*, ACM/IEEE, June 1985, pp. 475-481.

[7] G. De Micheli, "Synthesis of Control Systems", in *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, (editor), Martinus Nijhoff Publishers, 1987, pp. 327-364.

[8] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems", *IEEE Trans. Circuits and Systems*, Vol. CAS-28, No. 7, July 1981, pp. 634-645.

[9] E. Dirkes, *A Module Binder for the CMU-DA System*, Technical Report CMUCAD-85-43, Carnegie-Mellon Univ., Pittsburgh, PA , May 1985.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Company, San Francisco, Calif., 1979.

[11] D. D. Hill, *Language and Environment for Multi-Level Simulation*, Technical Report 185, Stanford University, Stanford, Calif. , March 1980.

[12] M. C. McFarland, *The Value Trace: A Data Base for Automated Digital Design*, Master's Thesis, Carnegie-Mellon Univ., Pittsburgh, PA, December 1978.

[13] M. C. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions", in *Proc. 23rd Design Automation Conference*, ACM/IEEE, June 1986, pp. 474-480.

[14] A. W. Nagle, R. Cloutier, and A. C. Parker, "Synthesis of Hardware for the Control of Digital Systems", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*., Vol. CAD-1, No. 4, October 1982, pp. 201-212.

[15] M. Odani, S. Y. Hwang, T. Blank, and T. Rokicki, *The ILSP Behavioral Description*

Language and its Graph Representation for Behavioral Synthesis, Technical Report CSL-TR-88-350, Stanford University, Stanford, Calif. , March 1988.

[16] B. M. Pangrle and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.*, Vol. CAD-6, No. 6, November 1987, pp. 1098-1112.

[17] N. Park and A. Parker, "Sehwa: A Program for Synthesis of Pipelines", in *Proc. 23rd Design Automation Conference*, ACM/IEEE, June 1986, pp. 454-460.

[18] A. C. Parker, "Automated Synthesis of Digital Systems", *IEEE Design and Test of Computers*, Vol. 1, No. 4, November 1984, pp. 75-81.

[19] A. C. Parker and S. Hayati, "Automating the VLSI Design Process Using Expert Systems and Silicon Compilation", *Proceedings of IEEE*, Vol. 75, No. 6, June 1987, pp. 777-785.

[20] P. G. Paulin, J. P. Knight, and E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", in *Proc. 23rd Design Automation Conference*, ACM/IEEE, June 1986, pp. 263-270.

[21] D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, NY, 1982.

[22] J. R. Southard, "MacPitts: An Approach to Silicon Compilation", *IEEE Computer*, Vol. 16, No. 12, December 1983, pp. 74-82.

[23] H. Trickey, "Flamel: A High Level Hardware Compiler", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.*, Vol. CAD-6, No. 2, March 1987, pp. 259-269.

[24] C. J. Tseng, A. M. Prabhu, C. Li, Z. Mehmood, and M. M. Tong, "A Versatile Finite State Machine Synthesizer", in *Proc. Int. Conf. Computer-Aided Design*, IEEE, November 1986, pp. 206-209.

[25] C. J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems.*, Vol. CAD-5, No. 3, July 1986, pp. 379-395.

[26] R. Waxman, "Hardware Design Languages for Computer Design and Test", *IEEE Computer*, Vol. 19, No. 4, April 1986, pp. 90-97.

A. Behavioral Description of FRISC

```
/*
 *   FRISC - 16-bit Microprocessor
 */

#define CPUSIZE 16

frisc()
{
    IN_LIST
        SIG ( RESET ) ;
        SIG ( IRQ ) ;
    ENDLIST ;

    OUT_LIST
        SIG ( IACK ) ;
    ENDLIST ;

    ST_LIST
        GRP ( P , CPUSIZE ) ;
        GRP ( S , CPUSIZE ) ;
        GRP ( M , CPUSIZE ) ;
        GRP ( A , CPUSIZE ) ;
        GRP ( B , CPUSIZE ) ;
        GRP ( I , CPUSIZE ) ;
    ENDLIST ;
    int dum , dum2 , T ;                    /* local variables */

    IACK = IRQ;

    if (RESET)
    {
        P[] = m_read ( 0 ) ;
        S[] = m_read ( 1 ) ;
    }
    else if (IRQ)                           /* Interrupt request? */
    {
        S[] = S[] + 1;
        dum = m_write ( S , B ) ;
        S[] = S[] + 1;
        dum2 = m_write ( S , A ) ;
        B[] = M[];
        A[] = P[];
        M[] = S[] + 1;
        P[] = m_read ( 2 ) ;
        IACK = 0;
    }

    I[] = m_read ( P ) ;
```

```
P[] = P[] + 1;                          /* Increment program counter */

while( I[] )
{     /* Until no opcodes left in buffer, decode ops. */
   if (I[3:0] == 1)
   {
      switch (I[7:4]) {
         case 0:                        /* Nand */
            A[] = ~(A[] & B[]);
            B[] = m_read ( S );
            S[] = S[] - 1;
            break;
         case 1:                        /* Subtract */
            A[] = B[] - A[];
            B[] = m_read ( S );
            S[] = S[] - 1;
            break;
         case 2:                        /* Shift right */
            A[] = A[] >> 1;
            break;
      }
      I[] = I[] << 8;                   /* Shift out opcode */
   }
   else                                 /* Normal op code */
   {
      switch (I[4:0]) {
         case 2:                        /* Constant */
            S[] = S[] + 1;
            dum = m_write ( S , B );
            B[] = A[];
            A[] = m_read ( P );
            P[] = P[] + 1;
            break;
         case 3:                        /* Get S */
            S[] = S[] + 1;
            dum = m_write ( S , B );
            B[] = A[];
            A[] = S[];
            break;
         case 4:                        /* Set S */
            S[] = A[];
            B[] = m_read ( S );
            S[] = S[] - 1;
            A[] = B[];
            B[] = m_read ( S );
            S[] = S[] - 1;
            break;
         case 5:                        /* Get M */
            S[] = S[] + 1;
            dum = m_write ( S , B );
```

```
        B[] = A[];
        A[] = M[];
        break;
    case 6:                                 /* Load */
        A[] = m_read ( A );
        break;
    case 7:                                 /* Store */
        dum = m_write ( B , A );
        A[] = B[];
        B[] = m_read ( S );
        S[] = S[] - 1;
        break;
    case 8:                                 /* Go to */
        P[] = A[];
        A[] = B[];
        B[] = m_read ( S );
        S[] = S[] - 1;
        break;
    case 9:                                 /* If */
        if ( B[] > 0 )
            P[] = A[];
        A[] = B[];
        B[] = m_read ( S );
        S[] = S[] - 1;
        break;
    case 10:                                /* End */
        A[] = B[];
        B[] = m_read ( S );
        S[] = S[] - 1;
        break;
    case 11:                                /* Mark */
        S[] = S[] + 1;
        dum = m_write ( S , B );
        B[] = A[] ;
        A[] = M[] ;
        M[] = S[] + 2;
        break;
    case 12:                                /* Call */
        T = P[];
        P[] = A[];
        A[] = T;
        break;
    case 13:                                /* Return */
        P[] = B[];
        S[] = M[];
        B[] = m_read ( S );
        S[] = S[] - 1;
        M[] = B[];
        B[] = m_read ( S );
        S[] = S[] - 1;
```

```
                        break;
                    case 14:                        /* Add */
                        A[] = B[] + A[];
                        B[] = m_read ( S ) ;
                        S[] = S[] - 1;
                        break;
                    case 15:                        /* Increment */
                        A[] = A[] + 1;
                        break;
                    default :
                        break ;
                }
                I[] = I[] << 4;                     /* Shift out opcode */
            }
        }
        return ;
}
```

Parallel Global Routing for Standard Cells

Jonathan Rose
Computer Systems Laboratory
Center for Integrated Systems
Stanford University, Stanford CA 94305

Abstract

Standard cell placement algorithms have traditionally used cost functions that poorly predict the final area of the circuit, and so can result in placements with good wire length but large final area. A good estimation of the area can be obtained by *global routing* the placement, but routing has been considered too slow to be used as the placement metric. This paper presents a new, fast global routing algorithm for standard cells and its parallel implementation. The router is based on enumerating a subset of all two-bend routes between two points, and results in 16% to 37% fewer total number of tracks than the TimberWolf global router for standard cells [Sech85]. It is comparable in quality to a maze router and an industrial router, but is faster by a factor of 10 or more. Three *axes* of parallelism are implemented: wire-by-wire, segment-by-segment and route-by-route. Two of these approaches achieve significant speedup — route-by-route achieves up to 4.6 using eight processors, and wire-by-wire achieves from 10 to 14 using 15 processors. Because these axes are *orthogonal*, when combined we demonstrate that their respective speedups multiply each other. A simple model is used to predict speedups of up to 61 using 120 processors.

1 Introduction

The best way to evaluate a placement of circuit modules is to route it and determine the final area. Since routing is a time-consuming task typical placement algorithms [Hana72,Breu77] use other metrics such as total wire length or crossing counts that are easier to calculate. With the advent of usable commercial multiprocessors it is possible to consider using more compute-intensive cost functions *if* efficient parallel algorithms can be developed. The aim of the *Locus Project* is to integrate placement and routing into one optimization process, and to do this in a practical way, by using multiprocessing to increase the speed of the routing.

This paper presents the first step in the Locus Project: *LocusRoute*, a new global routing algorithm for standard cells, and its parallel implementation. Our goal is to make the recalculation time of an area-based cost function on a multiprocessor the same as conventional cost functions on a uniprocessor. The intention is for the global router to be invoked to rip-up and re-route wires whose end points have changed when one or more cells have been moved. This goal implies that routing time must be about one to two milliseconds per net on a VAX 11/780-class machine.

The routing performance of LocusRoute, as measured by total number of routing tracks, is better than that of TimberWolf 4.2 [Sech85] and is comparable to a maze router and an industrial router. It is fast because it investigates only a subset of two-bend routes between pairs of pins to be routed. The routing speed is increased further by parallelizing the algorithm in three ways: routing several wires at once, routing several two-point segments simultaneously, and evaluating possible two-bend routes in parallel. The wire-by-wire parallel approach achieves speedups ranging from 10 to 14 using 15 processors. The route-by-route approach achieves speedups of up to 4.6 using 8 processors. These two "axes" of parallelism are *orthogonal* to each other, and so when used in tandem their speedups will multiply. This is demonstrated on 15 processors, and used to predict speedups in excess of 60 using 120 processors for standard benchmark circuits.

Previous work on parallel routing [Breu81, Blan81, Adsh82, Nair82, Rute84, Iosu86, Won87] has generally focused on a fixed hardware mapping for the Lee routing algorithm [Lee61]. As such they lack the flexibility that is required in practical CAD software such as the global routers described in [Kamb85,Yama85]. Another drawback of special hardware for the Lee algorithm is that a uniprocessor implementation can be made very efficient using special software data structures that cannot be put easily into fixed hardware.

There have been few publications on global routing for standard cells, other than [Kamb85] and [Yama85] which give little detail of the process. The cost-model used in [Pate85] is similar to that used in LocusRoute. A survey of global routing that touches on standard cells appears in [Lore88]. An early version of this work was presented in [Rose88b].

This paper is organized as follows: Section 2 defines the global routing problem for standard cells and describes the serial LocusRoute algorithm. Section 3 gives performance comparisons with the Timberwolf 4.2 global router [Sech85], a maze router, and the UTMC Highland Router [Robe87]. Section 4 presents three approaches for speeding up the new router using parallel processing, and performance results. Section 5 presents experiments with combining two of the approaches which are then used to model and predict speedups for larger numbers of processors.

2 The LocusRoute Algorithm

This section defines the standard cell global routing problem, and describes the new LocusRoute approach to solving it.

2.1 Problem Definition

Global routing for standard cells decides the following for each wire in the circuit:

1. For each group of electrically equivalent pins (pin clusters) it determines which of those pins are actually to be connected.

2. If there is no path between channels when one is required, it must decide either which built-in feedthrough to use or where to insert a feedthrough cell.

3. It decides which parts of a channel to use for a wire, including the use of two distinct wires in the same channel if this is desirable.

4. It must determine the channel to use in routing from a pad into the core cells.

In this discussion of global routing there will be no differentiation between feedthrough cells and built-in feedthroughs - they are referred to jointly as *vertical hops*. The decision to insert a feedthrough cell or use a built-in feedthrough is deferred to a post-processing step. This does result in some inaccuracy in the track count, and is discussed further in Section 3.4.

The objective of a global router is to minimize the sum of the channel densities of all the channels (hereafter called the *total density*). It is important to note that the total density can be traded off with the number of vertical hops, so to compare the total density of two global routings fairly they should both use the same number of vertical hops.

2.2 The Basic LocusRoute Algorithm

In the LocusRoute algorithm, each wire sequentially goes through the following five steps:

1. **Segment Decomposition.** A multi-point wire is decomposed into a minimum spanning tree of two-point *segments*, using Kruskal's algorithm [Krus56]. This algorithm has running time $O(n^2)$ in the number of pin clusters. The effect of the sub-optimality of this decomposition is discussed in Section 3.2 below.

2. **Permutation Decomposition.** The segments are further decomposed, if necessary, into *permutations*, which are the set of possible routes between each pin in a pin cluster.

3. **Route Generation and Evaluation.** A low-cost path is found for each permutation by evaluating a subset of the two-bend routes between each pin pair. The definition of the *cost* of a wire is given below, in Section 2.2.2. The permutation with the best cost is selected as the route for that segment.

4. **Reconstruct.** This step joins all the segments back together, and assigns unique numbers to distinct segments of the same wire in each channel. This is so that a channel router can distinguish between two segments and will not inadvertently join them together.

5. **Record.** The presence of the newly routed wire is recorded so that later wires can take it into account.

In addition, LocusRoute uses the iterative technique described in [Nair87]. Briefly, this means that after the first time all wires are routed, each is sequentially ripped up and then re-routed. By routing each wire several times (typically four is sufficient), the final answer is improved by five to ten percent because later wires can take earlier wires into account after the first iteration. This also reduces the effect of the wire order dependency.

The details of the second, third and fifth steps above are described in the following sections. The others are simple enough that the above description suffices.

2.2.1 Decomposition into Permutations

Each two-point segment consists of pairs of *pin clusters* that contain electrically equivalent pins. The LocusRoute algorithm considers routes between every pin in one cluster and every pin in the other cluster. Each such route is called a *permutation*. Figure 1 illustrates three of the four possible permutations between clusters A and B, which have two pins each. The four possible permutations are: (A_1,B_1), (A_1,B_2), (A_2,B_1), (A_2,B_2). If clusters A and B are separated by only a short horizontal distance, then the (A_1,B_2) permutation is most likely the least-cost path of the four. If the horizontal distance is large then it is possible that any one of the four permutations could have the low-cost path, and

hence all should be investigated. This has been confirmed experimentally, and a constant horizontal separation (300 routing grids) has been determined beyond which total density will improve if all four permutations are evaluated.

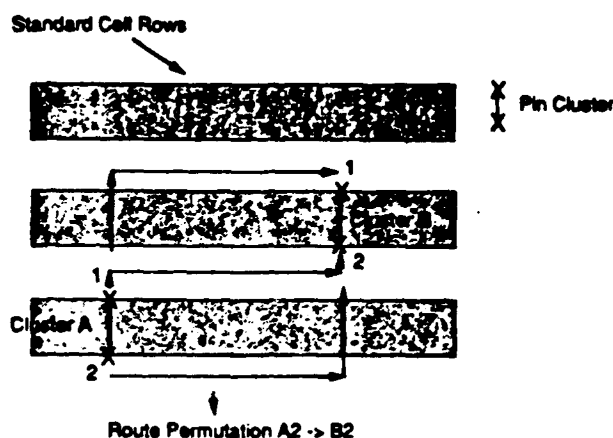


Figure 1 - *Permutation Decomposition of Segment*

2.2.2 Route Evaluation

The route evaluation step introduces two crucial notions of the LocusRoute algorithm: the cost model, which dictates the cost assigned to a path chosen for a wire, and the basic method of choosing routes based only on paths that have two or less bends.

Cost Model. Each possible routing position in a channel (also called *routing grid* of that channel) is represented as one element of an array as shown in Figure 2. The array, called the *Cost Array*, has a vertical dimension of the number of rows plus one, and a horizontal dimension of the width of the placement in routing grids. Each element of the Cost Array contains two values: H_{ij} and V_{ij}. H_{ij} contains the number of wire routes that pass horizontally through the grid at channel i in position j. This value changes as wires are routed. Similarly, V_{ij} is the cost, assigned by parameter, of traversing a row in travelling from channel i to channel $i+1$ at grid position j. A wire is represented as a list of (i, j) pairs of locations in the Cost Array, corresponding to the locations of pins to be joined.

This model implies that more than one vertical hop can exist in one grid location, and that the assignment of a vertical hop does not disturb the placement. While these assumptions are strictly incorrect, their effect is minimal as discussed in Section 3.4.

Under this model, the objective is to find a minimum-cost path for each wire. The wire's cost is given by the sum of all of the H_{ij} and V_{ij} that it traverses. After a path is found for a wire that goes through location (i, j) its presence is recorded in the Cost Array (the appropriate H_{ij} and V_{ij} are incremented) so that subsequent wires can take it into account. The more wires going through a particular location in a channel, the less likely it is that area will be used. Note that in this model the total density is not directly minimized, but rather a combination of average density and wire length.

Two-Bend Route Generation and Evaluation. The LocusRoute algorithm searches for a low-cost path

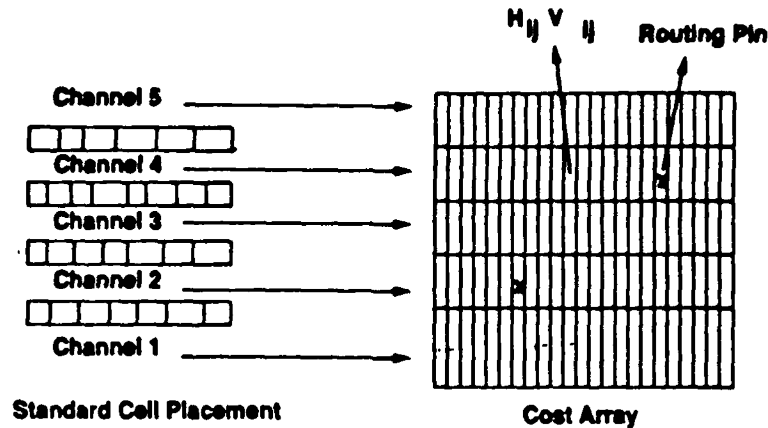


Figure 2 - *Cost Model*

for a permutation by *evaluating* the cost of a number of different routes and choosing the best. The basic approach is to evaluate a subset of all two-bend routes between the two pins, and then choose the one with the lowest cost. Generation of two-bend routes is discussed in [Ng86]. Figure 3 illustrates three possible two-bend (or less) routes inside a representation of the Cost Array as a small example.

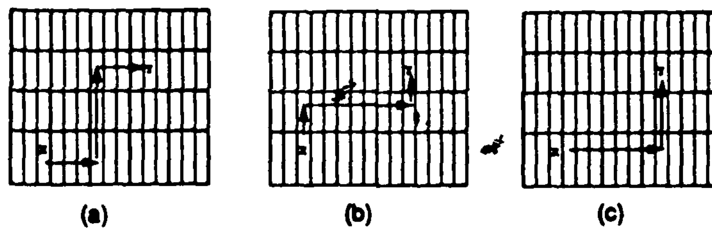


Figure 3 - *Sample Two-Bend Routes*

If the horizontal distance between the two pins is H routing grids, and the vertical difference is C channels then the total number of possible two-bend routes is $C+H$. In the LocusRoute algorithm the percentage of all the possible two-bend routes to be evaluated is a parameter. If fewer than 100% of all the routes are to be evaluated, the set of all possible routes is prioritized as follows: first all principally horizontal routes (those with bends only at the left and right extremes) are evaluated. Then the principally vertical routes (those with bends at the upper and lower extremes) are evaluated. Horizontal routes are evaluated first because it is important that all of the potential channels for the route be examined at least once. Within the horizontal and vertical groups, routes are searched in bisection order; i.e. if the limits of the group span are normalized to [0,1] then the routes are prioritized as $0, 1, \frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}$, and so on. This ensures that the possible space of routes is evenly spanned.

To calibrate the number of two-bend routes to be evaluated the two-bend router was compared against a least-cost path maze router. Both routers were not allowed to go beyond the bounding box of the two end points of the segment. Experimentally, it was determined that if only 20% of the two-bend routes were evaluated, then this would result in a path as good as that found by the maze router, as compared on the basis of total density for the entire circuit. On all of the test circuits used in the experiments discussed in the Section 3, the LocusRoute router's total density was within 2% of that obtained by the two-point maze router, with one exception of 3.3%. Most of the differences were below 1%. This is surprising in that the maze router looks for not only two-bend routes but for three or more bend routes. It implies that two-bend routes provide a sufficiently rich route set for the standard cell routing problem.

2.2.3 Recording A Wire

The last step in the algorithm is to record the presence of the wire's route in the Cost Array, so that the cost of using any part of that path will increase for other wires. This is done simply by incrementing the appropriate cells of the cost array. In the next iteration, the wire is "ripped up" by decrementing those same cells of the Cost Array.

3 Performance Comparisons

This section compares the quality and execution time of LocusRoute with three other routers.

3.1 Comparison with TimberWolf

Table 1 shows a comparison between the LocusRoute global router and the TimberWolf 4.2 [Sech85] global router for several industrial circuits. These circuits are from several sources: The standard cell benchmark suite (Primary1, Primary2, Test06 [Prea87]), Bell-Northern Research Ltd. (BNRA->BNRE), and the University of Toronto Microelectronic Development Centre (MDC). The placement for all of the circuits was done by the ALTOR standard cell placement program [Rose85, Rose88a]. Table 1 gives the number of wires in each circuit, the total density achieved by LocusRoute and Timberwolf, and the percentage fewer tracks LocusRoute achieved over Timberwolf. LocusRoute achieves significantly better total density than does the TimberWolf global router, ranging from 16% to 37% fewer tracks. The principal reason is that the TimberWolf global router is constrained to use only the minimum number of vertical hops, whereas LocusRoute uses considerably more. This is a reasonable practice in current technology because many standard cells contain "free" built-in feedthroughs. The execution times of LocusRoute and TimberWolf are comparable for most of the examples, though TimberWolf is faster by a factor of 8 and 3 respectively for circuits Test06 and Primary2. This is due to the fact that the LocusRoute algorithm increases in running time proportional to the area covered by the wire, which is much larger in these two circuits, and the inefficiency of the segment decomposition for large wires.

Circuit Name	# Wires	Total Density LocusRoute	TimberWolf	% Fewer
BNRE	420	138	179	22%
MDC	575	150	179	16%
BNRD	774	188	225	16%
Primary1	904	262	316	17%
BNRC	937	202	247	18%
BNRB	1364	320	442	27%
BNRA	1634	315	432	27%
Test06	1673	335	537	37%
Primary2	3029	563	702	20%

Table 1 - *Comparison of LocusRoute and TimberWolf*

3.2 Comparison with Maze Router

For comparison purposes a maze router [Lee61] was developed, using the same cost model as LocusRoute, that exhaustively determines the optimal solution to the two-point routing problem. It also determines a good approximation to the minimum-cost Steiner tree for multi-point wires using the approach described in [Aker72]. The maze router was carefully optimized for speed. Table 2 shows the comparison of total density and execution time for the maze router and the LocusRoute router, for all of the test circuits. The comparison is made on the basis of roughly equal numbers of vertical hops. Execution times are for four iterations over all wires on a DEC Micro Vax II.

Circuit Name	Total Density Locus	Maze	Diff	Time (Micro Vax II s) Locus	Maze	Factor
BNRE	138	129	7%	88	2378	27x
MDC	150	141	6%	178	3173	18x
BNRD	188	182	3%	167	3306	20x
Primary1	262	255	3%	325	6534	20x
BNRC	202	189	7%	363	7250	20x
BNRB	320	308	4%	599	15116	25x
BNRA	315	294	7%	769	19652	26x
Test06	335	316	6%	5137	92272	18x
Primary2	563	549	3%	3758	48295	13x

Table 2 - *Comparison of LocusRoute and Maze Router*

For all circuits the LocusRoute total density (total number of routing tracks) is no greater than 7% more than that achieved by the maze router, and in some cases is as little as 3% more. Most of this difference is due to the sub-optimality of dividing the wires up into two point nets. LocusRoute ranges from 13 to 27 times faster than the maze router. Since the purpose of this work is to use the router as an area-based cost function for a placement algorithm, we will always be willing to trade this slight loss in quality for such a large gain in speed. This will allow many more potential placements to be evaluated.

3.3 Comparison with the UTMC Highland Router

For two of our circuits, we can also compare the total routing density with the United Technologies global router used in the recent benchmark effort at the 1987 Physical Design Workshop [Prea87,Robe87]. The placements used above for circuits Primary1 and Primary2 were also routed by the UTMC router. Table 3 shows the comparison of total density for both circuits, with each router using roughly the same number of vertical hops. The total density of the UTMC router for circuit Primary1 is notably less than for the LocusRoute router. This is probably due to the fact that the UTMC router also performs neighbour exchanges and cell orientation changes on the placement in order to reduce the total number of tracks. The LocusRoute total density for circuit Primary2 is slightly less than that achieved by the UTMC router. We have no information on the execution time of the UTMC router, except that for circuits near the size of Primary2, it would take roughly 10000 Vax 11/780 seconds [Robe87] which is about three times slower than LocusRoute.

Circuit Name	# Wires	Total Density	
		LocusRoute	Highland
Primary1	904	253	194
Primary2	3029	560	562

Table 3 - *Comparison of LocusRoute and UTMC Highland Router*

3.4 Effect of Vertical Hop Approximation

As discussed in Section 2.1, the abstraction of vertical hops (representing both feedthrough cells and built-in feedthroughs), and the fact that they *overlay* active cells, causes an inaccuracy in the track counts reported here. The difference is small, however. The 904-wire Primary1 circuit global routed to 249 tracks, using 995 vertical hops under the LocusRoute algorithm. The actual, post-process track count using 10 feedthrough cells and 985 built-ins was 253, only 1.6% more tracks. For the 3029-wire Primary2 circuit with 3424 vertical hops (287 feedthroughs, 3137 built-ins) the approximate track count was 546 and the post-process count was 590, an increase of 8%.

4 Parallel Decomposition and Implementation

As mentioned in the introduction, previous parallel routers have focused on fixed hardware implementations of the maze routing algorithm [Lee61]. A more flexible approach is to use general purpose parallel processors, which can be adapted to many applications. Using the flexibility of a general purpose multiprocessor, several "axes" of parallelism can be exploited. If these axes are *orthogonal* to

each other then when used in tandem they can achieve significant speedup. Two approaches to parallelizing an algorithm are said to be orthogonal if, when used together, the resulting speedup is the product of the speedup of the individual methods. In this section several ways of parallelizing the LocusRoute router are proposed and implemented:

1. **Wire-based Parallelism.** Each processor is given an entire multi-point wire to route.

2. **Segment-based Parallelism.** Each two-point segment produced by the minimum spanning tree decomposition is routed in parallel.

3. **Permutation-based Parallelism.** Each of the four possible permutations, as discussed in Section 2.2.1, are evaluated in parallel.

4. **Route-based Parallelism.** Each of the possible two-bend routes for every permutation are evaluated in parallel.

Note that these are only *potential* axes of parallelism. It is possible to eliminate some of them as uneconomical by using statistical run-time measurements of the serial router. For example, the number of two-point segments that actually need to have all four permutations evaluated is quite small with respect to the total. Thus, permutation-based parallelism is not going to provide significant speedup. Other measurements show that the time spent evaluating the cost of two-bend routes ranges from 50 to 90 percent of the total routing time and so reasonable speedup from route-based parallelism can be expected.

The following sections gives the details of three axes of parallelism, their performance and a quantitative measure of the degradation in quality if there is some. All decompositions assume a shared-memory multiprocessor.

4.1 Wire-Based Parallelism

In Wire-Based parallelism, each multi-point wire is given to a separate processor, which runs the LocusRoute routing algorithm as described in Section 2. The Cost Array is a shared data structure to which all processors have read and write access. This is an excellent axis of parallelism: if the sharing of the Cost Array does not cause performance degradation due to memory contention, and there are enough wires to provide good load balance, then the speedup should simply be the number of wires that are routed in parallel. The resulting parallel answer, however, will not necessarily be the same as the sequential answer. The problem is that the sequential router has complete knowledge of all wires that have already been routed, by virtue of their presence in the cost array. The parallel router has less information because it doesn't see the wires that are being routed simultaneously. The more wires routed in parallel, the less information each processor has to choose good routes that avoid congestion and hence cause an increase in total density. Thus the total density will increase as the number of processors increase. The measured effect on total density is discussed below, in Section 4.1.1.

4.1.1 Wire-Based Parallel Performance

Figure 4 is a plot of the speedup versus number of processors for the 3029-wire (Primary2) example running on an sixteen-processor Encore MULTIMAX. The speedup for p processors, S_p is calculated as $\frac{T_1}{T_p}$, where T_1 is the execution time on one processor and T_p is the execution time using p processors. The Encore uses National 32032 chip sets which, in our benchmarks, timed out slightly faster than a DEC Micro Vax II.

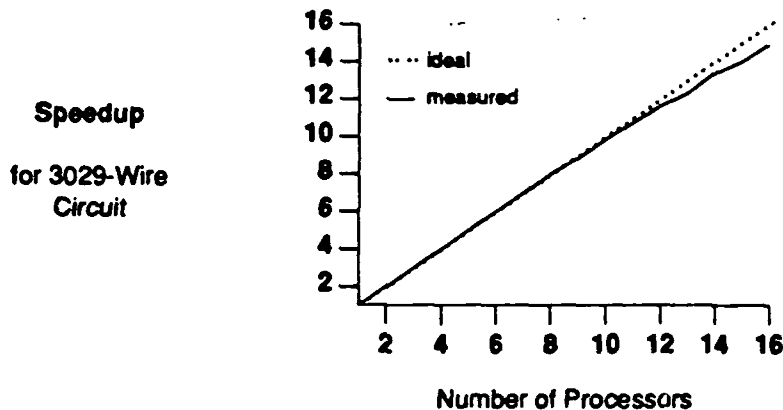


Figure 4 - *Wire-Based Speedup for Circuit Primary2*

It is clear from the figure that the wire-based approach achieves excellent speedup. Note that the execution time is only the actual routing computation time, excluding input time. For this circuit the increase in total density (between 1 and 16 processors) is negligible, and the number of vertical hops increases about 3%.

Table 4 gives the speedup using fifteen processors for the other test circuits. The speedup ranges from 10.1 for a smaller circuit to 14.1 for the largest. The speedup is less for smaller circuits because they are done in such a short time, so that the startup overhead and load balance become factors. The execution time is for four iterations over all the wires. It was discovered that very large global wires, such as TRUE or FALSE that have up to 150 pins, caused a severe degradation in speedup. This is because our system handles those nets just like any other, and the $O(n^2)$ nature of the minimum spanning tree algorithm causes load balancing problems. Since most production systems treat TRUE and FALSE signal nets differently (usually tapping directly into the power lines with special cells) these were eliminated under the assumption that they could be handled quickly that way.

Table 5 gives the density and vertical hop counts for both 1 and 15 processors using wire-based parallelism. The degradation in total density ranges between 1% to 7%. The increase in vertical hops is 6% or less. Again, in the context of using the router as a placement cost function, it is worthwhile to trade a small loss in quality for a large gain in speed, so that many more placements may be evaluated.

Circuit Name	1-Processor Time (s)	15-Processor Time (s)	15-Processor Speedup
BNRE	67	6.5	10.4
MDC	76.6	7.5	10.1
BNRD	136	11.8	11.5
Primary1	275	24.9	11.0
BNRC	196	16.9	11.6
BNRB	553	48.6	11.4
BNRA	713	54.9	13.0
Test06	5654	425	13.3
Primary2	3934	279	14.1

Table 4 - *Wire-Based Parallelism Speedup*

Circuit	Density			Vertical Hops		
Name	1-Proc	15-Proc	%Increase	1-Proc	15-Proc	%Increase
BNRE	130	137	5%	449	474	6%
MDC	134	142	6%	241	249	3%
BNRD	176	182	3%	530	574	6%
Primary1	262	271	3%	940	947	1%
BNRC	191	192	1%	725	739	2%
BNRB	307	328	7%	1904	1990	5%
BNRA	298	312	5%	2106	2198	4%
Test06	318	339	7%	3221	3309	3%
Primary2	560	593	6%	3053	3133	3%

Table 5 - *Wire-Based Parallelism Quality*

4.1.2 Gain Due to Removal of Locks

An interesting issue is whether or not each processor should lock the Cost Array as it both rips up and re-routes wires in the Cost Array. The act of ripping up a route is essentially a decrement, and re-routing is an increment on a set of cells in the Cost Array. Locking the Cost Array during these operations ensures that two simultaneous operations on the same element does not prevent one of the operations from being lost. It does, however, cause a significant performance degradation. For example, for the Primary1 circuit the speedup decreased from 8.3 to 6.4 using 15 processors when Cost Array locking was used. For the Primary2 circuit the speedup for 15 processors was reduced to 12.1 from 13.0 due to locking.

The final routing quality, however, does not decrease when locking is omitted. The reason for this is that the probability of two processors accessing the *same* Cost Array element (of which there are on the order of 10000) at the *same* instant is very low. Even if very few increment or decrement operations are lost, the effect on final quality is negligible since only a few elements would be wrong by a small amount. This was shown experimentally by performing ten runs with 15 processors on each of the above circuits, for both the locking and non-locking cases. For the two circuits table 6 gives the average running time, and the average and standard deviation of the total density and number of vertical hops. From this table it can be seen that the quality in both cases is very nearly the same. Note that in a placement context in which many more wires will be ripped up and re-routed, the effect of these small errors would be cumulative and so an occasional correction step may be necessary if locks are not used.

Circuit & Lock Type	Avg T (s)	Density		Vertical Hops	
		Avg.	SD	Avg	SD
Primary1 Locks	43.8	269	2.0	962	4.9
Primary1 NO Locks	33.7	272	3.0	964	3.4
Primary2 Locks	325	591	1.9	3126	7.5
Primary2 NO Locks	303	591	4.9	3122	4.0

Table 6 - *Speed & Quality Using and Not Using Locks*

4.2 Segment-Based Parallelism

In segment-based parallelism, each two-point segment of a wire is given to a different processor to route. This is the stage following the minimum spanning tree decomposition, but prior to the evaluation of different two-bend routes. Measurements of the sequential router showed that about 60% of the routing time was spent on wires with more than one segment. This means that a speedup of about two might be expected using three processors. Even though there are many wires that provide two or three-way parallel tasks, however, the size of those tasks are not necessarily equal. The amount of time taken by the LocusRoute router to route two points is proportional to the Manhattan distance between the two points. If, in a three-point wire, two of the points are close together and the third is far away, it will then take much longer to route one segment than the other. The processor assigned to the short segment will be idle while the longer one is being routed. This unequal load prevents a reasonable speedup. On the test circuits a speedup of about 1.1 using two processors was measured.

It is fairly clear, however, that an extra processor could be assigned to a *number* of processors that are routing different wires. It is likely that at any given time, one of them will be able to use the extra processor to route an extra segment. This technique would become essential in wire-based parallelism if the number of processors were increased much beyond sixteen. In that case, the load balance becomes a problem because wires with many segments take much longer than wires with few segments. Hence segment-based parallelism could be used as a method to balance those loads and speed up the routing of larger wires.

4.3 Route-Based Parallelism

In route-based parallelism all of the two-bend routes to be evaluated are divided among the processors. Each finds the lowest-cost path among the set of two-bend routes that it is assigned. When all processors finish, the route with the best overall cost is selected. In this case the processor loads are well balanced because the routes are all of the same length, and the number of routes is evenly divided among the processors.

Figure 5 is a plot of the speedup versus number of processors for the circuit Test06, a large circuit. It achieves a speedup of 4.6 using 8 processors.

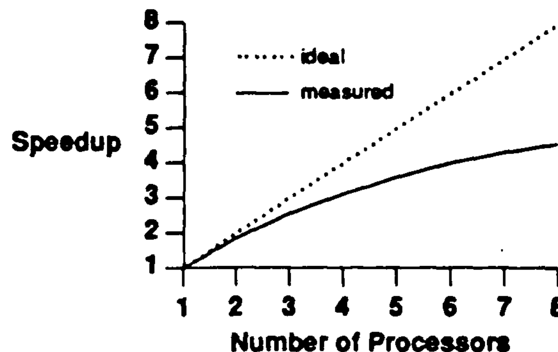


Figure 5 - *Route-Based Speedup for Test06*

Table 7 gives the best speedup achieved for all of the test circuits, ranging from 1.2 using 2 processors to 4.6 using 8 processors. The principal reason for the limitation in speedup is the sequential portion of the routing: the wire decomposition and the post-route processing that places the presence of the route into the Cost Array. On the small circuits that have lesser speedup, the sequential portion is about 50% of the total routing time, while on the larger circuits which have better speedup the sequential portion ranges from 10-15%. Another reason is that some segments have only one potential route, limiting the available parallelism.

5 Combining Two Orthogonal Axes of Parallelism

The wire and route axes of parallelism introduced above are orthogonal, and so when they are combined we can expect a multiplication of their respective speedups. In this section experiments are performed to demonstrate this effect on the Encore MULTIMAX. Using a simple model, the speedup for a larger number of processors is then predicted.

5.1 Implementation on the MULTIMAX

Because there are different kinds of tasks to be executed, the major challenge of combining the wire and route axes of parallelism is the *scheduling* of those tasks. An obvious static scheduling strategy is implied by the notion of orthogonality: for each wire that is being routed simultaneously by one processor in the wire-based approach, we now statically assign a constant number of processors to that wire to aid in the parallel execution of the route-based tasks. This situation is depicted in Figure 6. These

Circuit Name	Best Route-based Speedup (Speedup/#Processors)
BNRE	1.2/2
MDC	1.3/2
BNRD	1.3/2
Primary1	1.8/3
BNRC	1.6/3
BNRB	2.1/4
BNRA	1.9/4
Test06	4.6/8
Primary2	3.3/5

Table 7 - *Performance of Route-Based Parallelism*

extra processors are used only during the two-bend route evaluation.

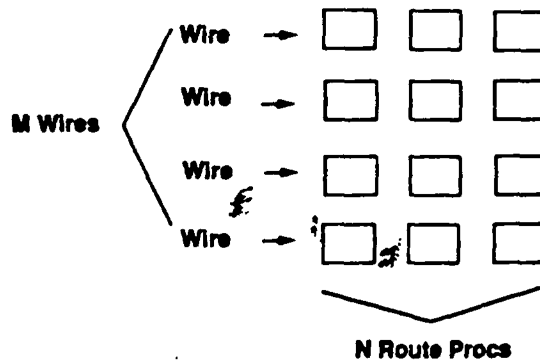


Figure 6 - *Static Scheduling Policy*

Several experiments were performed to show that the combined speedup of the wire and route-based approaches will indeed be the multiplication of the individually measured speedups. Table 8 gives the result of those experiments for the 3029-wire Primary2 circuit. For each experiment it gives the number of wires being routed in parallel (M), the number of processors assigned to each wire to do the routing tasks (N), the total number of processors ($M \times N$), the speedup predicted by multiplying the wire-based speedup using M processors and the route-based speedup using N processors, and the *measured* combined speedup. From this table it is clear that the speedups very nearly multiply, as expected. The small difference is due to increased contention for shared memory and the central bus, and the fact that two processors contend for one cache in the Encore MULTIMAX.

# Wire (M)	#Route Procs (N)	Total (M × N)	Speedup	
			Predicted	Measured
3	4	12	9.0	8.7
4	3	12	9.9	9.6
6	2	12	10.8	10.3
3	5	15	10.4	9.9
5	3	15	12.4	11.7
7	2	14	12.6	12.0

Table 8 - *Static Schedule Experiments for Circuit Primary2*

A drawback of the static scheduling policy is that it cannot assign processors where they will be of best use. If one wire has very few routes while another has many, the processors assigned to the first are not used by the second. In addition, there is a portion of the wire routing procedure that only uses one processor, so the others will be idle. A dynamic scheduling approach allows any idle processor to be used by any wire that has a need for it. This was implemented as a single task queue. Wire processors add tasks to the queue, and other processors remove and execute tasks. The *granularity* of the routing tasks in the dynamic scheme, the number of two-bend routes assigned to one processor to evaluate per task, was tuned to achieve the best speedup. The best performance was achieved when the number of tasks was several times the number of available processors, indicating that the load balance effect was more significant than the overhead of starting up a task.

Experiments have shown that the dynamic approach can both obtain the same speedup as the static approach using fewer processors, or better speedup using the same number of processors. For example, for Primary2 the static approach attained a measured speedup of 9.9 using 15 processors, while the dynamic approach achieved 10.8.

5.2 Predicting Performance on More Processors

Since we have observed that the static schedule performance of the combined approach does indeed nearly multiply the speedups attained by the individual methods, it is possible to predict the performance of that schedule on many more processors. Assume, for a given circuit that a speedup of S_w is achieved using wire-based parallelism on W processors, and a speedup of S_r is achieved using route-based parallelism on R processors. Then, because the two approaches are orthogonal, the resulting speedup when they are used together should be $S_w \times S_r$ using $W \times R$ processors. This model neglects the effect of memory contention that may occur when the number of processors is increased dramatically. Table 9 shows the best predicted speedup for the test circuits. Combined speedup ranges from 13 using 30 processors to 61 using 120 processors. The smaller circuits are routed very quickly and so it is difficult to get speedups greater than 13 due to the startup overhead. The larger circuits benefit greatly from the combination of the approaches.

Circuit	$\frac{S_w}{W}$	$\frac{S_r}{R}$	$\frac{S_w \times S_r}{W \times R}$	A_1	A_{RW}
BNRE	$\frac{10.4}{15}$	$\frac{1.2}{2}$	$\frac{12.5}{30}$	46ms	3.7ms
MDC	$\frac{10.1}{15}$	$\frac{1.3}{2}$	$\frac{13.1}{30}$	38ms	2.9ms
BNRD	$\frac{11.5}{15}$	$\frac{1.3}{2}$	$\frac{15.0}{30}$	50ms	3.3ms
Primary1	$\frac{11.0}{15}$	$\frac{1.8}{3}$	$\frac{19.8}{45}$	89ms	4.5ms
BNRC	$\frac{11.6}{15}$	$\frac{1.6}{3}$	$\frac{18.6}{45}$	59ms	3.2ms
BNRB	$\frac{11.4}{15}$	$\frac{2.1}{4}$	$\frac{24.0}{60}$	127ms	5.3ms
BNRA	$\frac{13.0}{15}$	$\frac{1.9}{4}$	$\frac{24.7}{60}$	134ms	5.4ms
Test06	$\frac{13.3}{15}$	$\frac{4.6}{8}$	$\frac{61.2}{120}$	935ms	15.2ms
Primary2	$\frac{14.1}{16}$	$\frac{3.3}{5}$	$\frac{46.5}{80}$	358ms	7.7ms

Table 9 - *Predicted Combined Speedup of Wire and Route Parallelism*

Table 9 also contains the average routing time per net on one processor, A_1, and what the the average routing time per net would be under the maximum speedup, A_{RW}. That is, $A_{RW} = \frac{A_1}{S_w \times S_r}$. The average routing times for all circuits, under the various speedups range mostly from 3 to 6ms, (with one at 15ms) and approaches our goal of one to two milliseconds per net. If more processors were used under the wire-by-wire axis, this goal could definitely be achieved.

6 Conclusions

A new global routing algorithm for standard cells and its parallel implementation has been presented. The LocusRoute algorithm uses significantly fewer tracks than the TimberWolf standard cell global router, and is comparable to a maze router and an industrial router. It is more than a factor of 10 faster than either of the two latter routers. Three axes of orthogonal parallelism were developed to speed up the LocusRoute router further. Two of the three axes that were implemented achieved significant speedup - up to 14.1 using fifteen processors and 4.6 using eight processors. They should produce combined speedups of up to 61 times.

The Locus placement environment is currently being developed, and in the future will be combined with the parallel LocusRoute global router. Our aim is to achieve smaller final area by using the global routing as a better measure of each placement.

Acknowledgements

The author is grateful to John Hennessy for the encouragement and support of this work. Thanks to Tom Blank who provided many good suggestions for an earlier version of this paper. Thanks also to Grant Martin of Bell-Northern Research for the use of company circuits and to the people involved in the standard cell benchmark effort for supplying those test circuits. Carl Sechen provided version 4.2 of TimberWolfSC.

7 References

[Adsh82]
H.G. Adshead, "Employing a Distributed Array Processor in a Dedicated Gate Array Layout System," Proc. ICCC, September 1982, pp. 411-414.

[Aker72]
S. B. Akers, "Routing," Chapter 6 of **Design Automation of Digital Systems; Theory and Techniques**, M.A. Breuer, Ed., Englewood Cliffs, NJ, Prentice-Hall, 1972.

[Blan81]
T. Blank, M. Stefik, W. VanCleemput, "A Parallel Bit Map Processor Architecture FOR DA ALGORITHMS," Proc. 18th Design Automation Conference, June 1981, pp. 837-845.

[Breu77]
M.A. Breuer, "Min-Cut Placement," Journal of Design Automation and Fault-Tolerant Computing, Oct 1977, pp 343-362.

[Breu81]
M.A Breuer, K. Shamsa, "A Hardware Router," Journal of Digital Systems, Vol IV, Issue 4, 1981, pp. 393-408.

[Hana72]
M. Hanan, J.M. Kurtzberg, "Placement Techniques," Chapter 4 of **Design Automation of Digital Systems; Theory and Techniques**, M.A. Breuer, Ed., NJ, Prentice-Hall, 1972.

[Iosu86]
A. Iosupovici, "A Class of Array Architectures for Hardware Grid Routers," IEEE Transactions on CAD, Vol. CAD-5, No. 2, April 1986, pp. 245-255.

[Kamb85]
T. Kambe, T. Okada, T. Chiba, I. Nishioka, "A Global Routing Scheme for Polycell LSI," Proc. ISCAS 1985, pp. 187-190.

[Krus56]
J.B. Kruskal, "On The Shortest Spanning Subtree of a graph and the Traveling Salesman Problem," Proc. Amer. Math. Soc, 7, 1956, pp. 48-50.

[Lee61]
C.Y. Lee, "An Algorithm for Path Connections and Its Applications," IRE Transactions on Electronic Computers, Vol EC-10, pp 346-365, 1961.

[Lore88]
M.J. Lorenzetti, D. S. Baeder, "Routing", Chapter 5 of **Physical Design Automation of VLSI Systems**, B. Preas and M. Lorenzetti Ed., Menlo Park, Benjamin/Cummings Publishing, 1988.

[Nair82]
R. Nair, S.J. Hong, S. Liles, R. Villani, "Global Wiring on a Wire Routing Machine," Proc. 19th Design Automation Conference, June 1982, pp. 224-231.

[Nair87]
R. Nair, "A Simple Yet Effective Technique for Global Wiring," IEEE Transactions on Computer-Aided Design, Vol CAD-6, No. 2, March 1987, pp. 165-172.

[Ng86]
A P-C Ng, P. Raghavan, C.D. Thompson, "A Language for Describing Rectilinear Steiner Tree Configurations," Proc. 23rd Design Automation Conference, June 1986, pp. 659-662.

[Pate85]
A.M. Patel, N.L. Soong, R.K. Korn, "Hierarchical VLSI Routing - An Approximate Routing Procedure," IEEE

Transactions on Computer-Aided Design, Vol CAD-4, No. 2, April 1985, pp. 121-126.

[Prea87]
B.T. Preas, "Benchmarks for Cell-Based Layout Systems," Proc. 24rd Design Automation Conference, June 1987, pp. 319-320.

[Robe87]
Ken Roberts used the United Technologies Standard Cell global router on the standard cell benchmark placements. Results were discussed at the 1987 DAC.

[Rose85]
J.S. Rose, W.M. Snelgrove, Z.G. Vranesic, "ALTOR: An Automatic Standard Cell Layout Program," Proc. Canadian Conference on VLSI, November 1985, pp. 168-173.

[Rose88a]
J.S. Rose, W.M. Snelgrove, Z.G. Vranesic, "Parallel Standard Cell Placement Algorithms with Quality Equivalent to Simulated Annealing," IEEE Trans. on CAD, Vol. CAD-7, No. 3, March 1988, pp. 387-396.

[Rose88b]
J.S. Rose, "LocusRoute: A Parallel Global Router for Standard Cells," Proc. 25th Design Automation Conference, June 1988, pp. 189-195.

[Rute84]
R.A. Rutenbar, T.N. Mudge, D.E. Atkins, "A Class of Cellular Architectures to Support Physical Design Automation," IEEE Trans. on CAD, Vol. CAD-3, No. 4, October 1984, pp. 264-278.

[Sech85]
C. Sechen, A. Sangiovanni-Vincentelli, "The Timberwolf Placement and Routing Package," IEEE JSSC, Vol. SC-20, No. 2, April 1985, pp 510-522. pp. 432-439.

[Won87]
Y. Won, S. Sahni, Y. El-Ziq, "A Hardware Accelerator for Maze Routing," Proc. 24th Design Automation Conference, June 1987, pp. 800-806.

[Yama85]
M. Yamada, T. Hiwatashi, T. Mitsuhashi, K. Yoshida, "A Multi-Layer Router for Standard Cell LSIs," Proceedings ISCAS 1985, 191-194.

To appear in International Journal of Parallel Programming, Volume 17, Issue #2, 1988

Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis

Anoop Gupta
Department of Computer Science, Stanford University, Stanford, CA 94305

Milind Tambe, Dirk Kalp, Charles Forgy, and Allen Newell
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213

Abstract

Until now, most results reported for parallelism in production systems (rule-based systems) have been simulation results — very few real parallel implementations exist. In this paper, we present results from our parallel implementation of OPS5 on the Encore multiprocessor. The implementation exploits very fine-grained parallelism to achieve significant speed-ups. For one of the applications, we achieve 12.4 fold speed-up using 13 processes. Our implementation is also distinct from other parallel implementations in that we parallelize a highly optimized C-based implementation of OPS5. Running on a uniprocessor, our C-based implementation is 10-20 times faster than the standard lisp implementation distributed by Carnegie Mellon University. In addition to presenting the performance numbers, the paper discusses the details of the parallel implementation — the data structures used, the amount of contention observed for shared data structures, and the techniques used to reduce such contention.

Keywords: Production Systems, Rule-based Systems, OPS5, Parallel Processing, Fine-Grained Parallelism, AI Architectures.

1. Introduction

As the technology of production systems (rule-based systems) is maturing, larger and more complex expert systems are being built both in industry and in universities. Often these large and complex systems are very slow in their execution, and this limits their utility. Researchers have been exploring many alternative ways for speeding up the execution of production systems. Some efforts have been focusing on high-performance uniprocessor implementations [3, 12], while others have been focusing on high-performance parallel implementations [2, 4, 7, 13, 10, 15, 16, 18]. This paper focuses on parallel implementations.

Until now, most results reported for parallelism in production systems have been simulation results. In fact, very few real parallel implementations exist. In this paper, we present results from our parallel implementation of OPS5 on an Encore Multimax shared-memory multiprocessor with sixteen CPUs. The implementation, called PSM-E (Production System Machine project's Encore implementation), exploits very fine-grained parallelism to achieve up to 12.4 fold speed-up for match using 13 processes. Our implementation is distinct from other parallel implementations in that we parallelize a highly optimized C-based implementation of OPS5. Running on a uniprocessor, our C-based implementation is 10-20 times faster than the lisp implementation of OPS5 distributed by Carnegie Mellon University. A consequence of parallelizing a highly-optimized implementation is that one must be

very careful about overheads, else the overheads may nullify the speed-up. One need not be as careful when parallelizing an unoptimized implementation. In this paper, we first discuss the design of an optimized implementation of OPS5, and then discuss the additions that were made for the parallel implementation. For the parallel implementation, we discuss the synchronization mechanisms that were used, the contention observed for various shared data structures, and the techniques used to reduce such contention.

The paper is organized as follows. Section 2 presents some background information about the OPS5 language, the Rete match algorithm, and the Encore Multimax multiprocessor. Section 3 gives an overview of the parallel interpreter and then goes into the implementation details describing how the rules are compiled and how various synchronization and scheduling issues are handled. Section 4 presents the results of the implementation on the Encore multiprocessor. Finally, in Section 5 we summarize the results and conclude.

2. Background

This section is divided into three parts. The first subsection describes the basics of the OPS5 production-system language — the language which we have implemented in parallel. The second subsection describes the Rete algorithm — the algorithm that forms the basis for our parallel implementation. The third subsection describes the Encore Multimax computer system — the multiprocessor on which we have done the parallel implementation.

2.1. OPS5

An OPS5 [1] production system is composed of a set of *if-then* rules called *productions* that make up the *production memory*, and a database of temporary assertions called the *working memory*. The assertions in the working memory are called *working memory elements* (wmes). Each production consists of a conjunction of *condition elements* corresponding to the *if* part of the rule (also called the *left-hand side* of the production), and a set of *actions* corresponding to the *then* part of the rule (also called the *right-hand side* of the production). The actions associated with a production can add, remove or modify working memory elements, or perform input-output. Figure 2-1 shows a production named *find-colored-block* with two condition elements in its left-hand side and one action in its right-hand side.

```
(p find-colored-block
   (goal  ^type find-block  ^color  <c>)
   (block ^id <i>  ^color <c>)
   (color ^code <c> ^name <s1>)
 -->
   (write "Found Block of Color <s1>"))
```

Figure 2-1: A sample production.

The production system *interpreter* is the underlying mechanism that determines the set of satisfied productions and controls the execution of the production system program. The interpreter executes a production system program by performing the following *recognize-act* cycle:

- **Match**: In this first phase, the left-hand sides of all productions are matched against the contents of working memory. As a result a *conflict set* is obtained, which consists of *instantiations* of all satisfied productions. An instantiation of a production is an ordered list of working memory elements that satisfies the left-hand side of the production.
- **Conflict-Resolution**: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.

- **Act**: In this third phase, the actions of the production selected in the conflict-resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, the first phase is executed again.

A working memory element is a parenthesized list consisting of a constant symbol called the *class* of the element and zero or more *attribute-value* pairs. The attributes are symbols that are preceded by the operator ^. The values are symbolic or numeric constants. For example, the following working memory element has class C1, the value 12 for attribute attr1 and the value 15 for attribute attr2.

```
(C1      ^attr1  12      ^attr2  15)
```

The condition elements in the left-hand side of a production are parenthesized lists similar to the working memory elements. They may optionally be preceded by the symbol –. Such condition elements are then called *negated* condition elements. Condition elements are interpreted as partial descriptions of working memory elements. When a condition element describes a working memory element, the working memory element is said to *match* the condition element. A production is said to be *satisfied* when: (1) For every non-negated condition element in the left-hand side of the production, there exists a working memory element that matches it; (2) For every negated condition element in the left-hand side of the production, there does not exist a working memory element that matches it.

Like a working memory element, a condition element contains a class name and a sequence of attribute-value pairs. However, the condition element is less restricted than the working memory element; while the working memory element can contain only constant symbols and numbers, the condition element can contain variables, predicate symbols, and a variety of other operators as well as constants. Variables are identifiers that begin with the character "<" and end with ">" — for example, <i> and <c> are variables. A working memory element matches a condition element if they belong to the same class and if the value of every attribute in the condition element matches the value of the corresponding attribute in the working memory element. The rules for determining whether a working memory element value matches a condition element value are: (1) If the condition element value is a constant, it matches only an identical constant. (2) If the condition element value is a variable, it will match any value. However, if a variable occurs more than once in a left-hand side, all occurrences of the variable must match identical values. (3) If the condition element value is preceded by a predicate symbol, the working memory element value must be related to the condition element value in the indicated way.

The right-hand side of a production consists of an unconditional sequence of actions which can cause input/output, and which are responsible for changes to the working memory. Three kinds of actions are provided to effect working memory changes. *Make* creates a new working memory element and adds it to working memory. *Modify* changes one or more values of an existing working memory element. *Remove* deletes an element from the working memory.

2.2. The Rete Match Algorithm

In this subsection, we describe the Rete algorithm used for performing the match-phase in the execution of production systems. The match-phase is critical because it takes 90% of the execution time and as a result it needs to be speeded up most. Rete is a highly efficient algorithm for match that is also suitable for parallel implementations. (A detailed discussion of Rete and and other match algorithms can be found in [4, 14].) The Rete algorithm gains its efficiency from two optimizations. First, it exploits the fact that only a small fraction of working memory changes each cycle by storing results of match from previous cycles and using them in subsequent cycles. Second, it exploits the similarity between condition elements of productions (both within the same production and between different productions) to reduce the number of tests that it has to perform to do match. It does so by

performing common tests only once.

The Rete algorithm uses a special kind of a data-flow network compiled from the left-hand sides of productions to perform match. The network is generated at compile time, before the production system is actually run. Figure 2-2 shows such a network for productions p1 and p2, which appear in the top part of the figure. In this figure, lines have been drawn between nodes to indicate the paths along which information flows. Information flows from the top-node down along these paths. The nodes with a single predecessor (near the top of the figure) are the ones that are concerned with individual condition elements. The nodes with two predecessors are the ones that check for consistency of variable bindings between condition elements. The terminal nodes are at the bottom of the figure. Note that when two left-hand sides require identical nodes, the algorithm shares part of the network rather than building duplicate nodes.

```
(p p1 (C1 ^attr1 <x> ^attr2 12)          (p p2 (C2 ^attr1 15 ^attr2 <y>)
      (C2 ^attr1 15  ^attr2 <x>)               (C4 ^attr1 <y>)
    - (C3 ^attr1 <x>)                      -->
  -->                                          (modify 1 ^attr1 12))
      (remove 2))
```

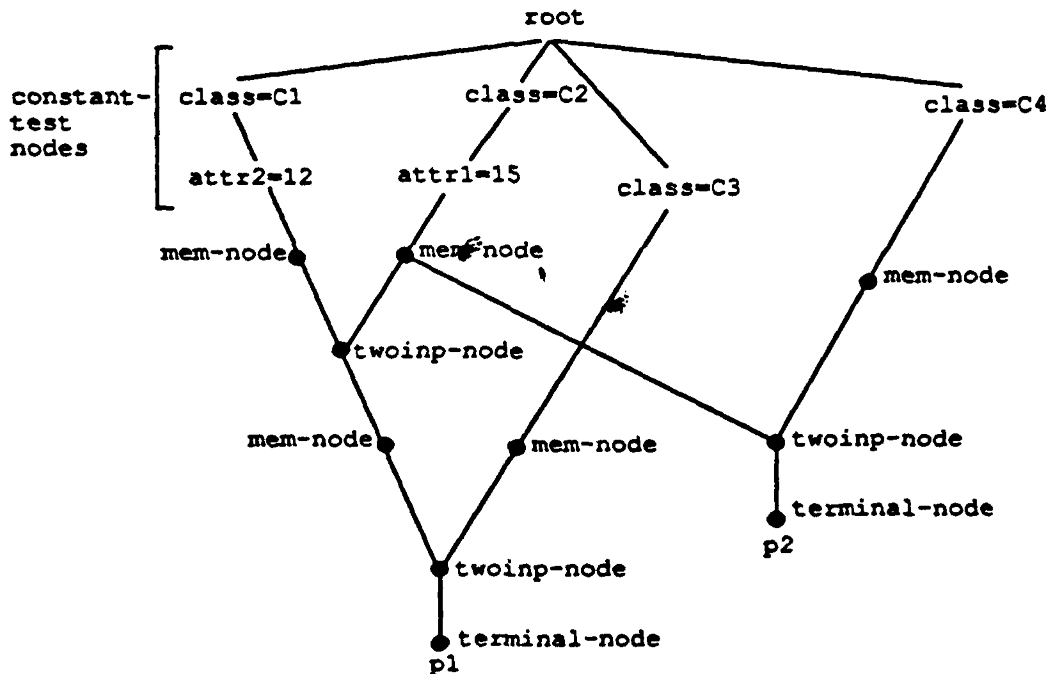


Figure 2-2: The Rete network.

To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as state within the nodes. This way, only changes made to the working memory by the most recent production firing have to be processed every cycle. Thus, the input to the Rete network consists of the changes to the working memory. These changes filter through the network updating the state stored within the network. The output of the network consists of a specification of changes to the conflict set.

The objects that are passed between nodes are called *tokens*, which consist of a *tag* and an *ordered list of working-memory elements*. The tag can be either a +, indicating that something has been added to the working

memory, or a –, indicating that something has been removed from it. No special tag for working-memory element modification is needed because a modify is treated as a delete followed by an add. The list of working-memory elements associated with a token corresponds to a sequence of those elements that the system is trying to match or has already matched against a subsequence of condition elements in the left-hand side.

The data-flow network produced by the Rete algorithm consists of four different types of nodes. These are:

1. **Constant-test nodes:** These nodes are used to test if the attributes in the condition element which have a constant value are satisfied. These nodes always appear in the top part of the network. They have only one input, and as a result, they are sometimes called *one-input* nodes.

2. **Memory nodes:** These nodes store the results of the match phase from previous cycles as state within them. The state stored in a memory node consists of a list of the tokens that match a part of the left-hand side of the associated production. For example, the right-most memory node in Figure 2-2 stores all tokens matching the second condition-element of production p2.

3. **Two-input nodes:** These nodes test for joint satisfaction of condition elements in the left-hand side of a production. Both inputs of a two-input node come from memory nodes. When a token arrives on the left input of a two-input node, it is compared to each token stored in the memory node connected to the right input. All token pairs that have consistent variable bindings are sent to the successors of the two-input node. Similar action is taken when a token arrives on the right input of a two-input node.

4. **Terminal nodes:** There is one such node associated with each production in the program, as can be seen at bottom of Figure 2-2. Whenever a token flows into a terminal node, the corresponding production is either inserted into or deleted from the conflict set.

The most commonly used interpreter for OPS5 is the Rete-based Franz Lisp interpreter. In this interpreter a significant loss in the speed is due to the interpretation overhead of nodes. In the OPS5 implementation we present in this paper, the interpretation overhead has been eliminated by compiling the network directly into machine code. While it is possible to escape to the interpreter for complex operations during match or for setting up the initial conditions for the match, the majority of the match is done without an intervening interpretation level. This has led to a speed-up of 10-20 fold over the Franz Lisp interpreter (see Table 4-4). In addition to this speed-up, our parallel implementation gets further speed-up by evaluating different node activations in the Rete network in parallel.

2.3. Encore Multimax

In this subsection, we describe the Encore Multimax shared-memory multiprocessor — the computer system on which parallel OPS5 runs. The Multimax consists of 2-20 CPUs, each of which is connected to the shared-memory through a high performance bus. The shared-memory is equally accessible to all of the processors, in that each processor sees the same latency for memory accesses.

The processors used in our Encore Multimax are National Semiconductor NS32032 chips along with NS32081 floating point coprocessors, each processor capable of approximately 0.75 million instructions per second. There are two processors packaged per board and they share 32 Kbytes of cache memory. The processor boards use a combination of write-through strategy and bus-watching logic to keep the caches on different processor boards consistent. The bus used on the Encore Multimax is called the *Nanobus*. It is a synchronous bus and it can transfer 8 bytes of new information every 80 nanoseconds, thus providing a data transfer bandwidth of 100 Mbytes/second.

The version of Encore Multimax available to us at CMU has 16 processors, 32 Mbytes of memory, and runs the MACH operating system developed at Carnegie Mellon University. The operating system provides a UNIX-like interface to the user, although the internals are different and several extensions have been made to support the underlying parallel hardware. It provides facilities to automatically distribute processes amongst the available processors and it provides facilities for multiple processes to share memory for communication and synchronization purposes. The results reported in this paper correspond to this configuration of the Encore Multimax.

3. Organization and Details of the Parallel Implementation

3.1. High-Level Structure of the Parallel Implementation

The parallel OPS5 implementation on the Encore (PSM-E) consists of one *control process* and one or more *match processes*. The number of match processes is a user specified parameter, but it is fixed for the duration of any particular run. The system is generally used in a mode where the computer contains at least as many free processors as there are processes in the matcher; this permits each process to be assigned to a distinct processor for the duration of the run (provided the operating system is reasonably clever about assigning processes to processors).

The control process is responsible for performing conflict resolution, evaluating the right-hand side of rules, handling input/output, and all the other functions of the interpreter except for performing match. It is also responsible for starting up the match processes at the beginning of the run and killing them at the end of the run. The match processes do nothing except perform the match. The match processes pipeline their operation with the control process. Thus when RHS evaluation begins, the match processes are idle. However, as soon as the first working memory change is computed, information about that change is passed to the match processes and they start to work. The control process continues evaluating the RHS, and as more changes are computed, the information is passed immediately to the match processes for them to handle as soon as they are able. When the control process finishes evaluating the RHS, it becomes idle and waits for the match processes to finish. When the last match process finishes, the control process performs conflict resolution and then begins evaluating the next RHS, thus starting the cycle over again.[1]

To perform match, the match processes use the Rete algorithm described in Section 2.2. The match processes exploit the dataflow-like nature of the Rete algorithm to achieve speed-up from parallelism. In particular, a single copy of the Rete network is held in shared memory. The match processes cooperate to pass tokens through the network and update the state stored in the memory nodes as indicated by the tokens. The match is broken into fairly small units of work called *tasks*, where a task is an independently schedulable unit of work that may be executed in parallel with other tasks. In our parallel implementation:

- All of the constant-test node activations constitute a single task. All these constant-test nodes are processed as a group, because individual constant-test node activations take only 2 machine instructions to execute (see Figure 3-1), and that is too fine a granularity.
- The memory nodes in the Rete network are coalesced with the two-input nodes that are below them. Each activation of these coalesced two-input nodes constitutes a single task. The reasons for this coalescing are discussed in [5]. As an example, the task corresponding to the left activation of a two-input node involves: (i) the addition/deletion of the incoming token to the left memory node; (ii) comparison of this token with all tokens in the opposite memory node checking for consistent variable bindings; and (iii) scheduling of matching token pairs for execution as new tasks. Note that multiple activations of the same two-input node constitute different tasks and these can be processed in parallel.
- Each individual terminal node activation constitutes a task.

In our current implementation, each task is represented by a data object called a *token*. The token in the parallel implementation is essentially the same as that used in the sequential Rete matcher (as described in Section 2.2), except that it has two extra items of information: the address of the node to which the token is to be sent, and if that node is a two-input node, an indication of whether to send it to the left or right input. The list of tokens that are

---

[1]For simplicity, we are ignoring two kinds of optimizations that are possible. First, it is possible to overlap conflict-resolution with match. Second, if *speculative* parallelism is used (we are willing to be wrong in our prediction sometimes and know how to recover from the error), it is possible to make a guess about the production that will fire next and to evaluate its right-hand side before conflict-resolution is completely finished. We choose to ignore these two optimizations for the present, because conflict-resolution and RHS evaluation are not the bottlenecks in our current implementation.

awaiting processing is held in a central data structure called a *task queue*. The individual match processes perform match by executing the following loop.

1. Remove a token from the task queue. If the queue is empty, wait until something is added.
2. Process the token. If new tokens are to be sent out, push them onto the task queue.
3. Go to step 1.

3.2. Implementation Details

When studying parallelism in production systems (or in any other application for that matter), it is important to compute the speed-ups with respect to the performance of the most efficient uniprocessor implementations. It is indeed quite easy to obtain large speed-ups with respect to inefficient implementations of the application, but such results have little practical utility. In the case of OPS5, the most efficient uniprocessor implementations are currently based on the Rete algorithm and they compile the Rete network directly into machine code and use global register allocation. Such compilation into machine code gives approximately 10-20 fold speed-up over Rete-based lisp implementations of OPS5 (see Table 4-4). For this reason, our parallel implementation of OPS5 on the Encore is also Rete-based and compiles the Rete network directly into machine code.[2] Another effect of parallelizing a highly efficient implementation versus an inefficient one is that the number of instructions executed in each parallel subtask (for the same task decomposition) is smaller in the highly efficient implementation. This is equivalent to exploiting parallelism at a finer granularity, and as a result, the issues of synchronization and scheduling are more critical.

As stated in the previous paragraph, the nodes in the Rete network are compiled directly into NS32032 machine code. Some of the operations performed by the nodes are too complex to make it reasonable to compile the necessary code in-line. For these operations, subroutine calls are compiled into the network. The subroutines themselves are coded in C and assembler. For example, a two-input node is compiled into a combination of subroutine calls for modifying and searching through the node memories plus in-line code to perform the node's variable binding tests. The OPS5 compiler uses global register allocation to make the code significantly more efficient. For example, register r6 always contains the pointer to the working-memory element currently being matched. The NS32032 assembly code generated to perform match for a simple production is shown in Figure 3-1. The code is presented here to provide a feel for the compiler and the level of optimization. For example, it shows that to evaluate a constant-test node it requires only 2 machine instructions, a compare followed by a branch. It is not essential to understand the code to understand the rest of the paper.

All communication between processes (both the match processes and the control process) takes place via shared memory. The virtual address spaces are set up so that the objects in shared memory have the same virtual address in every process. Hence processes can simply pass pointers around in essentially the same way routines within a single process can. For example, the tokens are created in shared memory, and the address of a given token is the same in every virtual address space in the system. Thus when a process places a token onto the central task queue, all it really has to do is to put the address of the token into the task queue. Figure 3-2 shows how the shared-memory is used to communicate between the various processes.

Synchronization within the program is handled explicitly by executing interlocked test-and-set instructions. The synchronization primitives provided by the operating system (for example, semaphores, barriers, signals, etc) are not used because of the large overhead associated with them. When a process finds that it is locked out of a critical

[2]Note that the argument in the beginning of this paragraph does not say that one has to use the same algorithm (as the most efficient uniprocessor one) for the parallel implementation. It just turns out in our case, that the efficient uniprocessor algorithm is also very good for parallel implementation [5].

```
!!! Rule for which code is presented below
(p p2
   (c1 ^a1 7  ^a2 <x>   ^a3 <y>)
   (c2 ^a1 <x>  ^a2 15  ^a3 <y>)
  -->
   (write  fired successfully ))


_ops_rete_root:
        movd    r6, r4                  ! Register r4 gets pointer to wme
        movd    @_curdir, @_succdir     ! Successor_direc = Current_direc
        cmpd    4(r6),@ops_symbols+4    ! Test if class = c2
        bne     .L11                    ! If test fails try next node
        cmpd    12(r6), $30             ! Test if ^a2 = 15
        bne     .L11                    ! If test fails try next node
        addr    @.L13, r0               ! Push task on to task_queue to
        bsr     _PushTaskQueue          ! begin evaluation at .L13
        br      .L11                    ! Start evaluating next node
.L13:   movd    $0, r3                  ! v----------------------------v
        xord    16(r6), r3              ! Compute hash index for
        xord    8(r6), r3               ! token
        andd    $0xfff, r3              ! ^----------------------------^
        movd    @_curdir, @_succdir     ! v----------------------------v
        movd    $0, tos                 ! Code + procedure call to add/
        bsr     ?_rmem                  ! del token to right memory node
        adjspb  $-4                     ! ^----------------------------^
        cmpqd   $0, r0                  ! Done with node activation if
        bne     @LeaveBetaTask          ! matching conjugate token found.
        cmpqd   $0, 0(_ltokHT)[r3:d]    ! Done with node activation if
        beq     @LeaveBetaTask          ! opposite mem-node empty
.Ltop0: movd    $0, r2                  ! Lev-of-node-actvn as param in r2
        bsr     _ops_lnext              ! Locate next token in opp mem
        cmpqd   $0, r5                  ! If all tokens have been examined
        beq     @LeaveBetaTask          ! then exit
        bsr     .L12                    ! Evaluate two-inp node tests
        br      .Ltop0                  ! Loop back to get next token
.L11:   cmpd    4(r6),@ops_symbols      ! Test if class = c1
        .
        .
        .
.L12:   movd    0(r4), r2               ! v----------------------------v
        movd    8(r5), r1               ! Perform tests to check if vars
        cmpd    16(r2),12(r1)           ! are consistently bound.  If tests
        bne     .L16                    ! fail, then return immediately,
        cmpd    8(r2),8(r1)             ! else push successor nodes on
        bne     .L16                    ! task queue and then return
        br      .L17                    !
.L16:   ret     $0                      ! ^----------------------------^
.L17:   addr    .L18, r0                ! Push address of successor node
        bsr     _PushTaskQueue          ! activation on to task queue
        ret     $0                      ! and return
```

Figure 3-1: Code generated for matching a production.

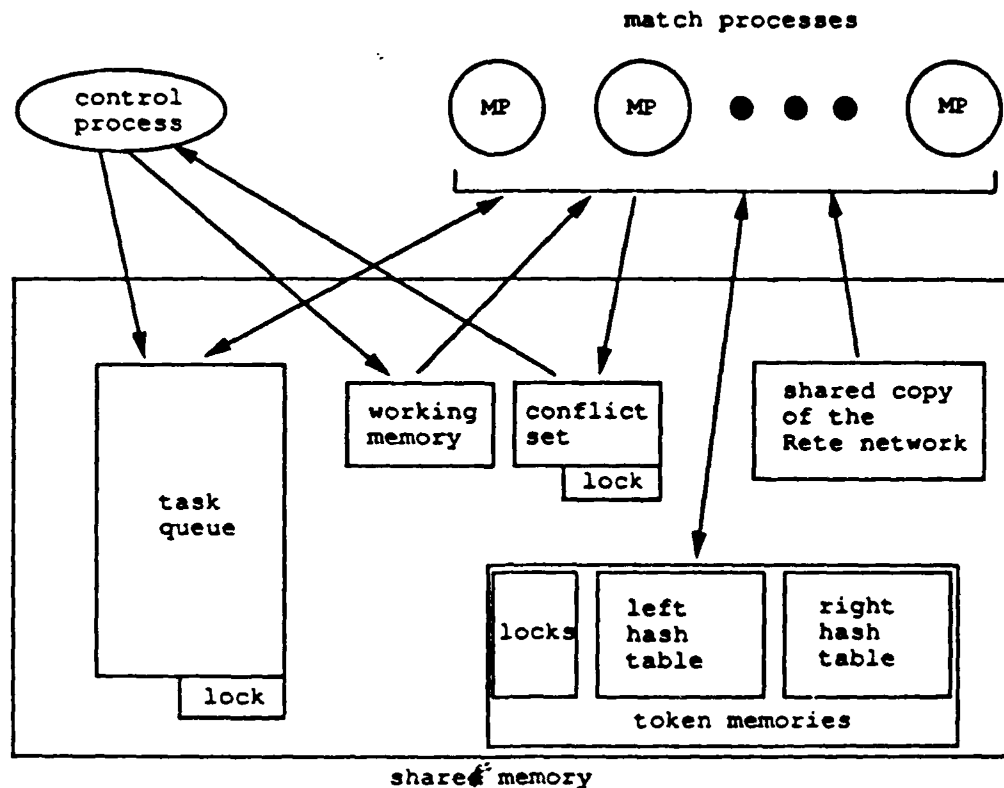


Figure 3-2: Use of shared-memory by various processes.

region it spins on the lock, waiting for a chance to enter the region. In order to minimize the amount of bus traffic generated by the spinning processes, a "test and test-and-set" synchronization mechanism is used. In this scheme, a process uses ordinary memory-read instructions to test the status of a lock until it finds that it is free; then the process uses a test-and-set interlocked instruction to re-read the lock and set it (if it is still free). Note that while the lock is busy, the process spins out of its cache and does not use the bus. This is more efficient than using only the "test-and-set" interlocked instruction for the lock. In this case, the process generates bus traffic to perform the writes while it is busy waiting.

The control process communicates with the match processes primarily through the shared task queue. Whenever the evaluation of an RHS results in a change to working memory, a token is created and marked as being destined for the root node of the network. The control process pushes these tokens onto the task queue in exactly the same way as the match processes push the tokens they create. The tokens are picked up and processed by waiting match processes. When the evaluation of an RHS begins, the match processes are idle. The first token created by the control process causes the match processes to start up. After the first token, the control process proceeds in parallel with the match processes.

Depending on the granularity of tasks (number of instructions executed per task) that are scheduled using the task queue and depending on the number of processors that are trying to access the task queue in parallel, it is quite possible that a single task queue would become a bottleneck. For this reason, Gupta [5] proposed a hardware task scheduler for scheduling the fine-grained tasks. So far we have not implemented the hardware scheduler, and in this

paper we present results only for the case when one or more software task queues are used.

After the control process finishes evaluating the RHS, it must wait for the match processes to finish before it can perform the next conflict resolution operation. A global counter, TaskCount, is used to determine when all the match processes have finished. This counter contains the sum of:

- the number of tokens that are currently on the task queue, and
- the number of tokens that are being processed by the match processes.

This count is maintained quite simply. Every time a token is put onto the task queue, the counter is incremented. Every time a match process finishes working with a token, the counter is decremented. The match phase is finished when the counter goes to zero.

Shifting our focus back to the evaluation of individual two-input node activations, we note that instead of having separate memories for each two-input node, the matcher has two large hash tables which hold all the tokens for the entire network. One hash table holds tokens for left memories of two-input nodes, and the other for right memories of two-input nodes. An alternative scheme is to have separate hash tables for each two input node, but such a scheme was considered to be wasteful of space. The hash function that is applied to the tokens takes into account:

- The values in the token which will have equality tests applied at the two-input node, and
- The unique identifier of the two-input node which stored the tokens. The unique identifier is randomized to minimize the number of hash-table collisions.

This permits the two-input nodes to locate any tokens that are likely to pass the equal-variable tests quickly. It also permits multiple activations of the same two-input node to be processed in parallel.

The processing performed by the individual node activations in the parallel implementation is similar to the processing done in the sequential matcher with two exceptions:

- Code has been added to the two-input nodes to handle conjugate token pairs.
- Sections of code that access shared resources are protected by spin locks to insure that only one process at a time can be accessing each resource.

A *conjugate pair* is a pair of tokens with opposite signs (an add token request and a delete token request), but which refer to the same working memory element or list of working memory elements. Conjugate pairs arise in the match operation for a variety of reasons, which are too complex to go into here (see [5]). They occur in both sequential and parallel implementations of Rete, but they present much greater problems in a parallel system. The reason for this is that in a parallel system it is not possible to insure that the tokens will be processed in the order in which they are generated, and consequently in some cases a token with a – (delete) flag will arrive at a two-input node before the corresponding token with the + (add) flag. The parallel matcher code handles this by saving the – tokens that arrive early on an *extra-deletes-list* without otherwise processing the token. When the corresponding + token arrives both tokens are discarded.

Many resources in a parallel system have to be protected with mutual-exclusion locks — the task queues, the count of the number of active tokens, the conflict set, etc. Most of these are relatively straight-forward to protect and a simple variation of standard spin locks is used. The exception is the locks used to control access to the token hash tables. There are several different operations that are performed on the token hash tables, for example, searching for matching tokens, adding and removing tokens, adding and removing conjugate tokens, and we would like many of these operations to proceed in parallel without having any undesirable effects. Because of the importance of the hash tables to the performance of the system, several locking schemes were implemented and tried. Two of these schemes are described here.

The first scheme, the simple one, is easy to describe and it provides a departure point for describing the second more complex one. We define a "line" as a pair of corresponding buckets (buckets with the same hash index) from the left and right hash tables along with their associated extra-deletes lists. In this scheme, each line in the hash table has a flag controlling its use.[3] The flag takes on two values: *Free* and *Taken*. When a process has to work with the hash table, it examines the flag for the line it needs. If the flag is *Free*, it sets the flag to *Taken* and proceeds to perform the necessary operations; when it finishes, it sets the flag back to *Free*. If a process finds the flag set to *Taken*, it waits until the flag is set to *Free*. Of course, the act of testing and setting the flag must be an atomic operation. This synchronization scheme works, but it is a potential bottleneck when several tokens arrive at a node about the same time, and if all of them require access to the same hash table line.

The second scheme is a complex variant of the *multiple-reader-single-writer* locking scheme. It permits several tokens to be processed in the same line at the same time, though even here, some serialization of the processing is necessary when destructive modifications to the lists of tokens are performed. This scheme requires two locks, a flag, and a counter for each line in the hash table. The flag takes on three values: *Unused*, *Left*, and *Right*, to indicate respectively that the line is not currently being processed, that it is being used to process tokens arriving from the left, or that it is being used to process tokens arriving from the right. The counter indicates how many processes are using that line in the hash table; it is needed only so that the last process to finish using the line can set the flag back to *Unused*. The first lock insures that only one process at a time can access the flag and the counter. When a process first tries to use a line in the hash table, it gets this lock, and checks the flag. If the flag indicates that tokens from the other side are being processed, the process releases the lock and tries again. If the flag allows the process to continue, it sets the flag if necessary, increments the counter, and releases the lock. For the remaining time that the process uses this line in the hash table, it leaves the flag and the counter untouched; finally, when the process finishes using the line it decrements the counter and if appropriate sets the flag to *Unused* (again, all within a section of code protected by this lock). All this is to insure that tokens from two different sides are not processed at the same time. The second lock is used to insure that only one process at a time can be modifying the token lists. Recall that the first task in processing a two-input node is to update the list of tokens stored in the memory node. To do this, the process gets the modification lock, searches the conjugate or regular token list, and it either adds the token to or deletes it from one of these lists. When it has finished, it releases the modification lock and proceeds with searching the tokens in the opposite hash-table bucket to find those that satisfy the variable binding tests.

More complex locking schemes can be devised and, in fact, were implemented and tested. One other scheme that was tried permitted more than one process to search the token lists to find tokens to delete; in this scheme the only serialization of the tasks occurred when the actual destructive modification of the token list was performed. As in all implementations, the main tradeoff to keep in mind is that in an attempt to speed-up the rare cases, one should not slow-down the normal case.

3.3. RHS Evaluation and Conflict Resolution

In our system, the rules' RHSs are compiled into a form of threaded code which is interpreted at run time [9]. Figure 3-3 shows a small piece of such threaded code. Interpreting the threaded code is slower than executing the compiled code, but since RHS evaluation is not a bottleneck to the performance, threaded code, which is simpler to compile was considered fast enough. Conflict resolution in the system is handled by code written in the C language. This code is executed by the control process.

[3]Note that any given operation on the token hash tables requires access to only a single line of the hash tables. In other words processing a single node activation never requires access to multiple hash table lines.

```
p1:  #    -- p1 --                ! Begin code for RHS of rule p1
     .double _bmake               ! Begin a make-wme action
     .double _symcon              ! v---------------------------v
     .double ops_symbols          ! Set class of wme to c1
     .double _rval                ! ^---------------------------^
     .double _tab                 ! v---------------------------v
     .double 4                    !
     .double _fixcon              ! Set 4th field of wme to 5
     .double 5                    !
     .double _rval                ! ^---------------------------^
     .double _tab                 ! v---------------------------v
     .double 3                    !
     .double _fixcon              ! Set 3rd field of wme to 10
     .double 10                   !
     .double _rval                ! ^---------------------------^
     .double _emake               ! End of make-wme action
     .double _opsret              ! End code for RHS of rule p1
```

Figure 3-3: Threaded code used to execute RHS actions.

4. Results and Analysis

In this section, we present results obtained from the execution of three production-system programs. We first present some statistics from our uniprocessor implementation, and we then present the speed-ups obtained by our parallel implementation. We also present a detailed analysis of the speed-ups observed. The three programs that we have studied are:

- Weaver [8], a VLSI routing program by Rostom Joobbani with 637 rules.
- Rubik, a program that solves the Rubik's cube by James Allen with 70 rules.
- Tourney, a program that assigns match schedules for a tournament by Bill Barabash from DEC with 17 rules.

We have chosen Weaver because it represents a fairly large program and it demonstrates that our parallel OPS5 can handle real systems. Rubik is a smaller program that demonstrates some of the strengths of our parallel implementation and the Tourney program demonstrates some of the weaknesses of our parallel implementation.

4.1. Results for the Uniprocessor Implementations of OPS5

Before we did a parallel implementation on the Encore, we initially did several uniprocessor C-based implementations of OPS5. In this subsection, we present results for two of these uniprocessor implementations, vs1 and vs2, for the Microvax-II workstation.[4] The performance results for vs1 and vs2 implementations are shown in Table 4-1. The base version is vs1, and it is characterized by the use of linear lists to store tokens in node memories, just as uniprocessor lisp implementations do.[5]

[4] The results are presented for Microvax-II and not for Encore, because the uniprocessor implementations were done on the Microvax and only one of these was later taken over to the Encore.

[5] Note that memory nodes are not shared in either vs1 or vs2 versions of OPS5, unlike in the Franzlisp version of OPS5. This optimization was not used in vs1 or vs2 because it is not possible to share memory nodes in the parallel implementations of OPS5 (see [5]), and we did not want to spend the effort just for the uniprocessor implementations.

Table 4-1: Uniprocessor versions on Microvax-II.

PROGRAM	VS1 List-based memories (sec)	VS2 Hash-based memories (sec)	Total number of WM-changes processed	Total number of node activations
Weaver	101.5	85.8	1528	371173
Rubik	235.2	96.9	8350	554051
Tourney	323.7	93.5	987	72040

The second version, vs2, uses a global hash table to store all memory-node tokens, as discussed in the previous section. If there are equality tests at the two-input node, the hash-table based scheme (i) reduces the number of tokens that have to be examined in the opposite memory to locate those that have consistent variable bindings, and (ii) for deletes, it reduces the number of tokens that have to be examined in the same memory to locate the token to be deleted. The statistics for the reduction in tokens examined in the opposite memory for the three programs are given in Table 4-2. Note the statistics are computed only for those node activations where the opposite memory is not empty. The statistics for the reduction in tokens examined in the same memory for delete requests are given in Table 4-3. As can be seen from the two tables, the savings are substantial, especially for the Tourney program. The time-saving effect of hash-based memories can be seen from numbers in Table 4-1.

Table 4-2: Number of tokens examined in opposite memory.

PROGRAM	Tokens in opp mem for left actvns		Tokens in opp mem for right actvns	
	lin mem	hash mem	lin mem	hash mem
Weaver	10.1	7.7	5.2	1.0
Rubik	31.0	3.8	1.6	1.8
Tourney	47.6	5.9	270.1	23.3

Table 4-3: Number of tokens examined in same memory for deletes.

PROGRAM	Tokens in same mem for left actvns		Tokens in same mem for right actvns	
	lin mem	hash mem	lin mem	hash mem
Weaver	6.2	3.6	7.0	5.1
Rubik	23.5	2.6	8.1	3.7
Tourney	254.4	40.1	3.8	2.9

The second last column in Table 4-1 gives the total number of wme-changes processed during the run for which

data are presented, and the last column gives the total number of node activations processed during the run (this is also equal to the number of tasks that are pushed/popped from the task queue in the parallel version). Dividing the time in column vs2 by the number of tasks, we get the average duration for which a task executes. This has implications for the amount of synchronization and scheduling overhead that may be tolerated in the parallel implementation. Doing this division we get that the average duration of a task for Weaver is 230 microseconds (or approximately 115 machine instructions, as the VAX executes about 500,000 instructions per second), for Rubik is 175 microseconds, and for Tourney is 1300 microseconds.

Finally, Table 4-4 gives the speed-up that our uniprocessor C-based implementation achieves over the widely available Franzlisp-based OPS5 implementation when running on the Microvax-II. As the table shows, we get a speed-up of about 10-20 fold over the Franzlisp based implementation. The problem in the past has been that due to lack of availability of better uniprocessor performance numbers, researchers have ended up comparing the performance of their highly optimized parallel OPS5 implementations with the slow Franzlisp-based implementation. We think that such apples to oranges comparison can be misleading.

Table 4-4: Speed-up of C-based over Franzlisp-based implementation.

PROGRAM	VS-lisp Lisp-based implemen. (sec)	VS2 Hash-based memories (sec)	Speed-up VS-lisp/VS2
Weaver	1104.0	85.8	12.9
Rubik	1175.0	96.9	12.1
Tourney	2302.0	93.5	24.6

4.2. Results for the Multiprocessor Implementation of OPS5

While the uniprocessor C-based implementations of OPS5 were done on the Microvax-II, the parallel version was done on the Encore multiprocessor. In this subsection, we present speed-up numbers for our implementation on the Encore as we vary (i) the number of task queues that are used and (ii) the locking structures used for token hash-table buckets. The speedups reported here are with respect to a parallel version of the program running on a uniprocessor. We also present a detailed analysis of the speed-ups observed.

Figure 4-1 shows results for the case when a single task queue is used and when *simple* locks (described in Section 3.2) are used with the token hash-table buckets. The figure also shows the uniprocessor times (in seconds) for the three programs. Note that the numbers along the X-axis represent the number of match processes; they do not include the control process. The speed-ups for all three programs are quite disappointing. This is especially true for Tourney, where the maximum speed-up is 2.6-fold with 5 match processes and it decreases even further as the number of match processes is increased.

There are several possible reasons for the low speed-up: (i) contention for access to the single task queue, (ii) contention for access to the hash-table buckets, (iii) low intrinsic parallelism in the programs, (iv) contention for hardware resources, and so on. We now explore the effects of removing the first two bottlenecks and provide some data on the intrinsic parallelism in the programs.

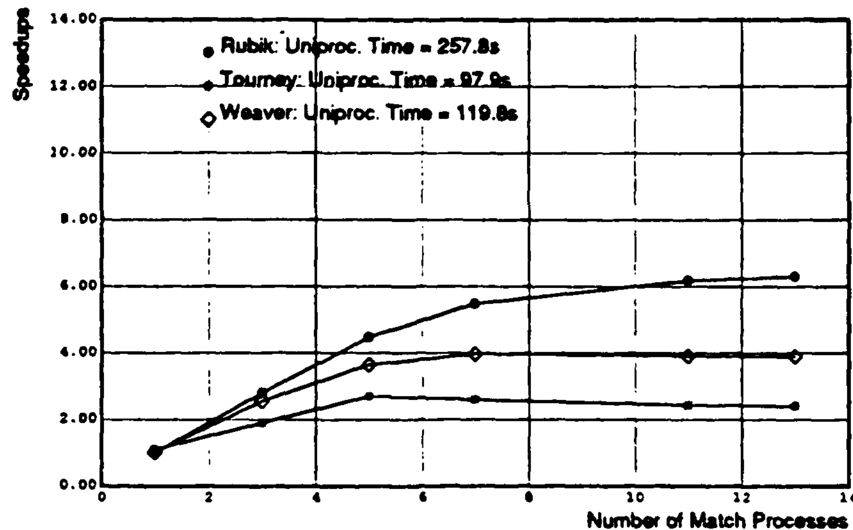


Figure 4-1: Speed-up for single task queue and simple hash-table locks.

4.2.1. Reducing Contention for the Centralized Task Queue

The contention for the single task queue can be reduced by the introduction of *multiple task queues*. Every process has its own queue, onto which it pushes and pops tasks. If it runs out of tasks then it cycles through the other processes' task queues, searching for a new task. Figure 4-2 presents the speed-up obtained when multiple task queues are used, while still using simple hash-table locks. The speed-up increases significantly for both Weaver and Rubik, indicating that the contention for pushing and popping task queues must have been a bottleneck. The speed-up for Weaver for 13 processes goes up from 3.9-fold to 8.2-fold and that for Rubik goes up from 6.3-fold to 11.4-fold. The speed-up for Tourney remains about the same at 2.4-fold.

To get more insight into these results, we instrumented the task queue to get data on contention. The results are shown in Figure 4-3. Here we plot the number of times a process spins on the task-queue lock as a function of the number of match processes. We see from the figure that as the number of processes is increased, there is indeed significant contention for the single task queue in case of Weaver and Rubik. For Tourney, there does not seem to be as much contention for the task queue, and that is why the speed-up does not increase when multiple task queues are used. Since the speed-up is still very low for Tourney (only 2.4-fold with 13 processes), in the next subsection we will explore if contention for the hash-table buckets is causing the poor speed-up.

Another question that arises when using multiple task queues is: "How many task queues should one use to maximize speed-up?". For example, when using 12 match processes, should we have 2, 4, 8, or 12 task queues. Too few task queues have the disadvantage that the contention for the task queues may still be a bottleneck. An excessive number of task queues has the disadvantage that most of the task queues will be empty, and the processes will waste time scanning several empty task queues before finding one with a task. Figure 4-4 plots the speed-up obtained for 12 match processes, as the number of task queues is increased. What the graph shows is that the optimal number of task queues varies for different programs. For Rubik, the more the task queues the better the speed-up. For Weaver, the speed-up increases up to 4 task queues, then remains the same up to 8 task queues, and then decreases slowly. For Tourney, the number of task queues really does not seem to matter. However, as a design decision, it seems that erring on the side of too many task queues is better than having too few task queues.

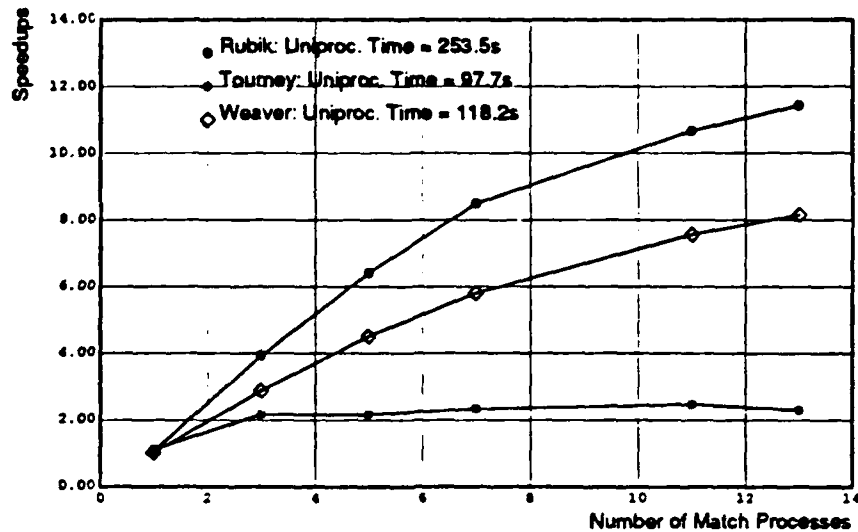


Figure 4-2: Speed-up for multiple task queues and simple hash-table locks.

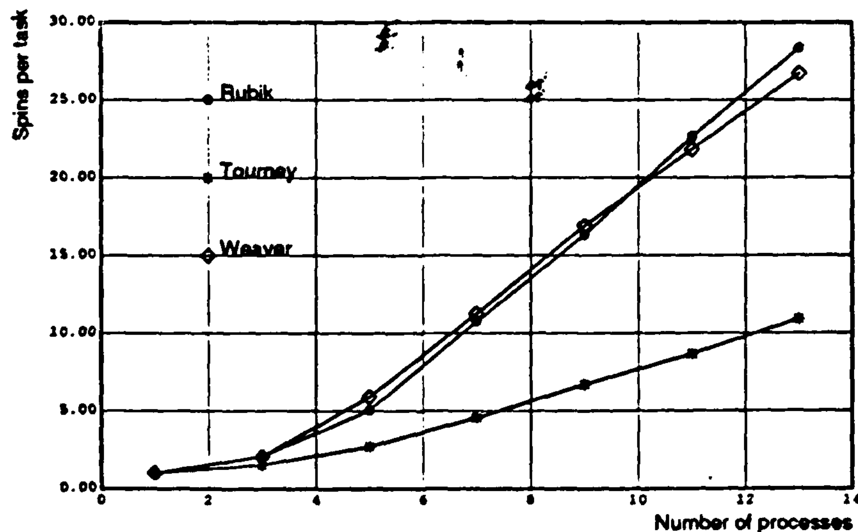


Figure 4-3: Contention for the centralized task queue. Measured by the number of times a process spins on the lock before it gets access to the task queue.

Finally, examining the speed-up for Rubik in Figure 4-2, it is interesting to note that we get 3.9-fold speed-up using only 3 match processes. This apparently anomalous behavior of the speed-up being greater than the number of match processes can be explained as follows. When the Rete network is evaluated in parallel, it is quite possible that the total number of node activations evaluated and their complexity is less than that of the sequential implementation. Of course, the final result of the match is still the same as the sequential implementation.

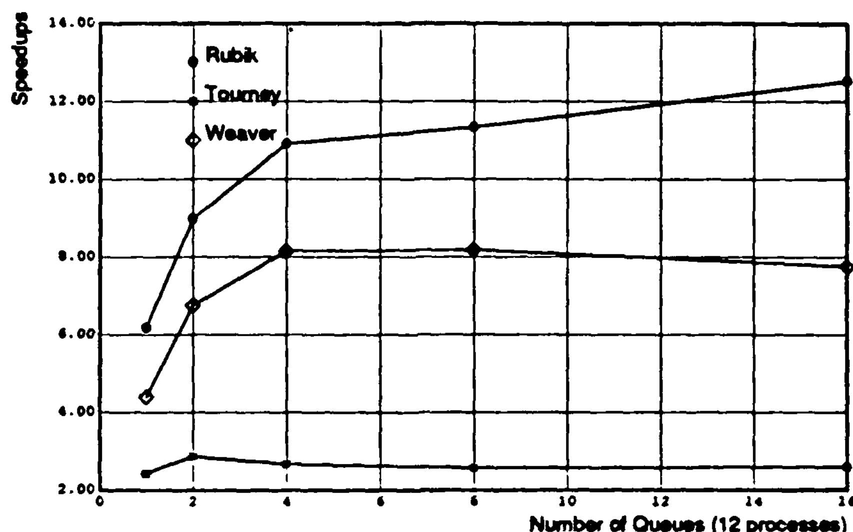


Figure 4-4: Speed-up with 12 match processes as the number of task queues is varied.

4.2.2. Reducing Contention for the Hash-Table Buckets

As discussed in Section 3.2, tokens associated with memory nodes in the Rete network are stored in two large hash tables. These hash tables are shared among all the processes. A single lock controls the access to a *line*, i.e., a pair of corresponding buckets from left and right hash tables. This lock provides a spot of contention for the various processes. The contention for a hash bucket lock can be measured by the number of times a process spins on a lock before it gets access to a line of hash table buckets. For right tokens, that activate the two-input nodes along the right inputs, the contention for the lock is very low — 1 or 2 spins per access — for all three programs, and it does not change as the number of processes is increased. This is because the right tokens are distributed evenly and most right tokens typically require very little processing [5]. The right and the left tokens do not typically contend with each other, as the right tokens are evaluated in the beginning of the cycle; while the left tokens are evaluated only later in the cycle.

For the left tokens, which activate the two input nodes from their left inputs, the contention is much higher. Figure 4-5 shows the contention observed by left tokens when the per-bucket lock used is a simple one (an ordinary spin lock), as discussed in Section 3.2. With 11 match processes, Rubik processes spin 23 times, Tourney processes spin 377 times, and Weaver processes spin 51 times on average before getting access to the hash-table bucket. While the contention for Rubik and Weaver is quite high and it is enormous for Tourney, given that each spin takes several microseconds.

In Figure 4-7 we present results for the case when multiple task queues are used and when *complex multiple-reader-single-writer* locks (described in Section 3.2) are used for controlling entry to the token hash tables. We expected the complex locks to benefit those programs that (i) generate cross-products, that is, there are multiple activations of the same two-input node from the same side that need concurrent processing, and (ii) have long lists of tokens in hash-table buckets, where the complex locks help by allowing multiple processes to read the opposite memory at the same time. However, programs for which the above two conditions are not true may slow down when complex locks are used, because of the extra overhead that they incur due to complex locks. Figure 4-6

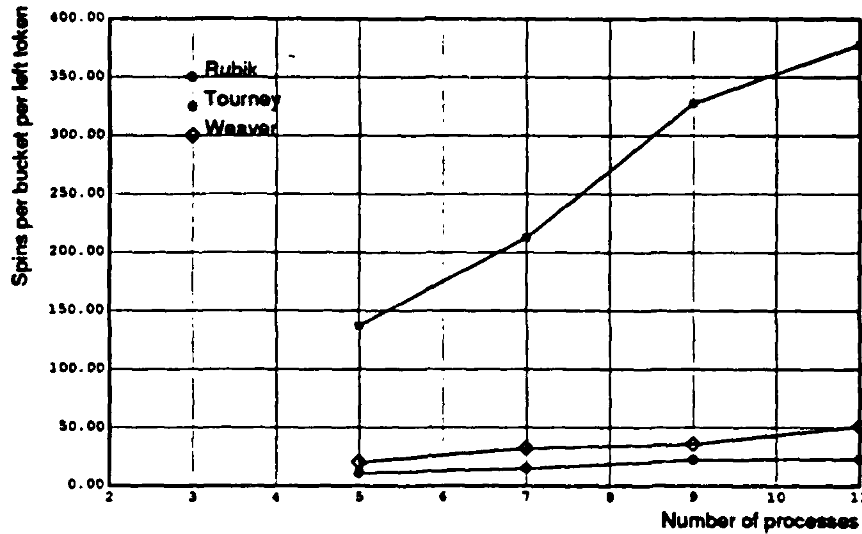


Figure 4-5: Contention for hash-table locks for left tokens. Measured by the number of times a process spins on a lock before it gets access to the hash-table bucket.

presents some results about contention when complex locks are used. Comparing with Figure 4-5, we see that the contention for the hash-table buckets decreases for all three programs, although the contention for Tourney is still very high in absolute terms. Analyzing the Tourney program in more detail, we found that the large contention for the hash-table locks is resulting from multiple node activations trying to access the same hash-table bucket. This, in turn, is the result of a few culprit productions in Tourney that have condition elements with no common variables. By modifying two such productions using domain specific knowledge, we could increase the speed-up achieved using 13 processes from 2.7-fold to 5.1-fold.

4.2.3. Other Causes for Low Speed-Up

In this subsection, we explore reasons other than contention for shared-memory objects that limit the speed-up achieved by our implementation. To this end, we examine the speed-ups obtained in individual recognize-act cycles for the programs. (Recall that the computation of an OPS5 program involves a series of recognize-act cycles.) Figure 4-8 presents the speed-ups obtained in each cycle as a function of the number of tasks (node activations) executed in that cycle[6]. These numbers are presented for Weaver, but they are representative of other OPS5 programs too. The speed-ups were measured with 11 match processes and the implementation used multiple task queues and simple hash-table locks. The 7.5-fold speedup obtained for weaver (shown in Figure 4-2), is a weighted average of the speedups for individual cycles shown in Figure 4-8.

The data points in Figure 4-8 can be divided into two regions:

- Short Cycles Region: Points in the left quarter of the graph (corresponding to cycles with less than 250 tasks), generally achieving a speed-up of 2 to 7-fold.
- Long Cycles Region: Points in the right three quarters of the graph (corresponding to cycles with 250-1000 tasks), generally achieving a speed-up of 6 to 10-fold.

[6] For presentation purposes, the cycles with more than 1000 tasks are shown as containing 1000 tasks.

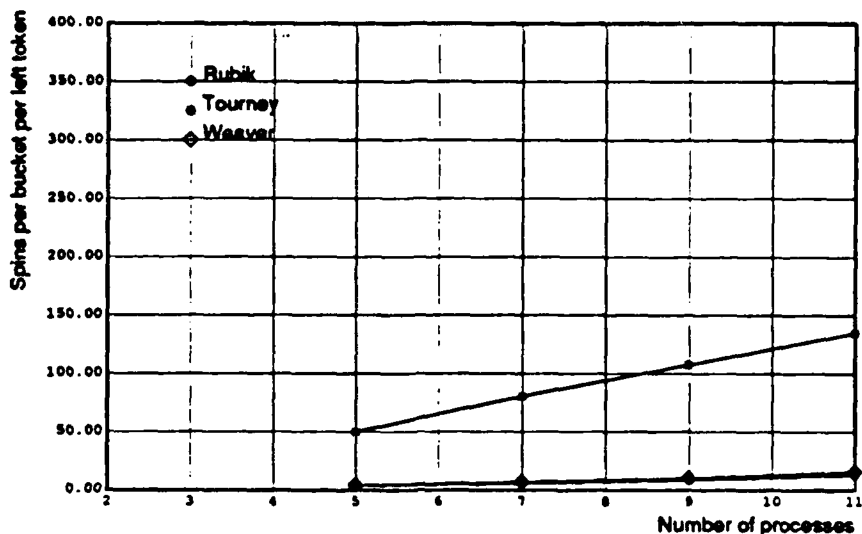


Figure 4-6: Contention for hash-table locks when multiple-reader single-writer locks are used.

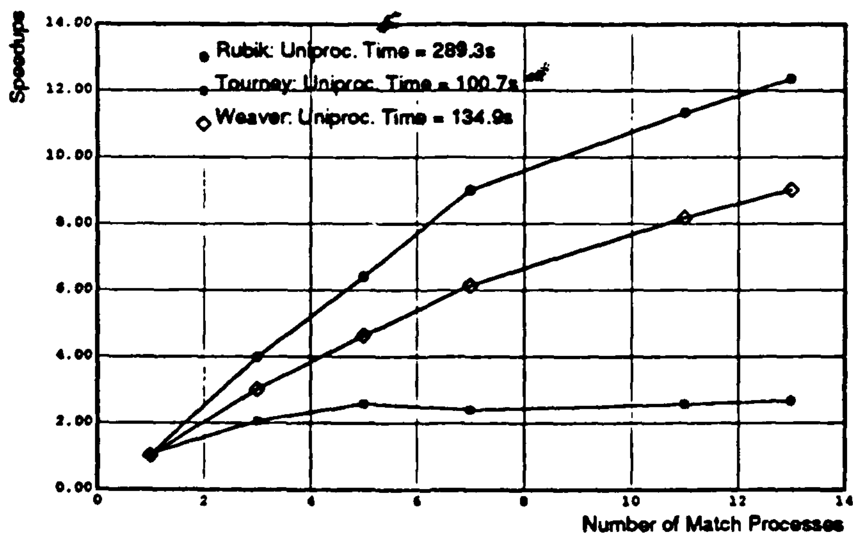


Figure 4-7: Speed-up for multiple task queues and multiple-reader-single-writer hash-table locks.

Let us first look at short cycles more closely. To understand the speed-up achieved, we plot the number of tasks in the system (which is the sum of the number of tasks waiting to be processed and those being processed) as a function of time during a cycle. Figure 4-9 shows such a plot for one of the short cycles in Weaver with about 100 tasks. The graph is plotted for 11 match processes and the time is measured in units of 100 microseconds. The graph may be interpreted as showing the number of processors that can be kept busy if infinite processors were

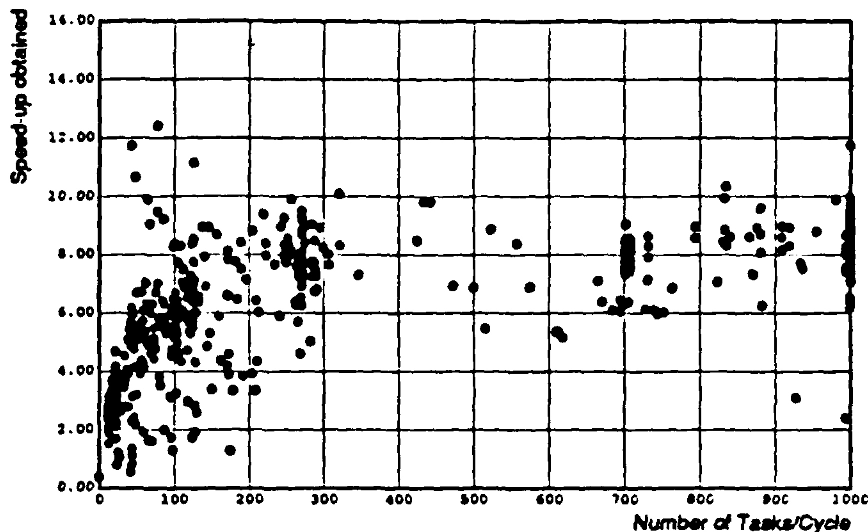


Figure 4-8: Weaver: Speed-up as a function of tasks/cycle.

present.[7] The average height of this graph (about 6.5 for this cycle) indicates the maximum speed-up that we should expect. In general, the smaller cycles tend to provide such low available parallelism.

Some additional speed-up losses occur due to the fixed overheads associated with match cycles; for example, having to check all the task queues to ensure that the match is actually finished and to inform the control process about the completion of the match. These almost fixed duration overheads affect the speed-up obtained by small cycles much more than that obtained by large cycles, as they form a larger fraction of the small cycle processing cost.

We now explore speed-up limits in long cycles. In Figure 4-10 we show a plot similar to that in Figure 4-9, except this time for a longer cycle with about 300 tasks. We see that in the early part of the graph (until time 30) the potential parallelism increases slowly, then it rises very steeply peaking at the point (62,66), then it falls rapidly (until time 65), and finally it has a slow spiky decline to the end of cycle (time 120). The portion of the graph that hurts the average speed-up most, however, is the portion from time 90 to time 120, where the system keeps processing a few tasks; each time generating only a few new tasks. This behavior is caused by the presence of chains of dependent node activations [5], which can get especially bad for productions that have a large number of condition elements. The impact of long chains on speed-ups increases with increasing number of processes. With more processes, the system can get through the earlier part of the computation (the one marked up to the first 90 time units, in Figure 4-10) faster, but it cannot get through the latter part much faster. To counter these long chains, we plan to change the Rete network organization for productions with large number of condition elements. This new network organization is called a *constrained bilinear* organization and it will allow us to reduce the dependencies between tokens (see [5, 17] for details).

[7]This interpretation is not totally correct for portions of the graph where the height of the shaded region is greater than the number of match processes used to produce the graph.

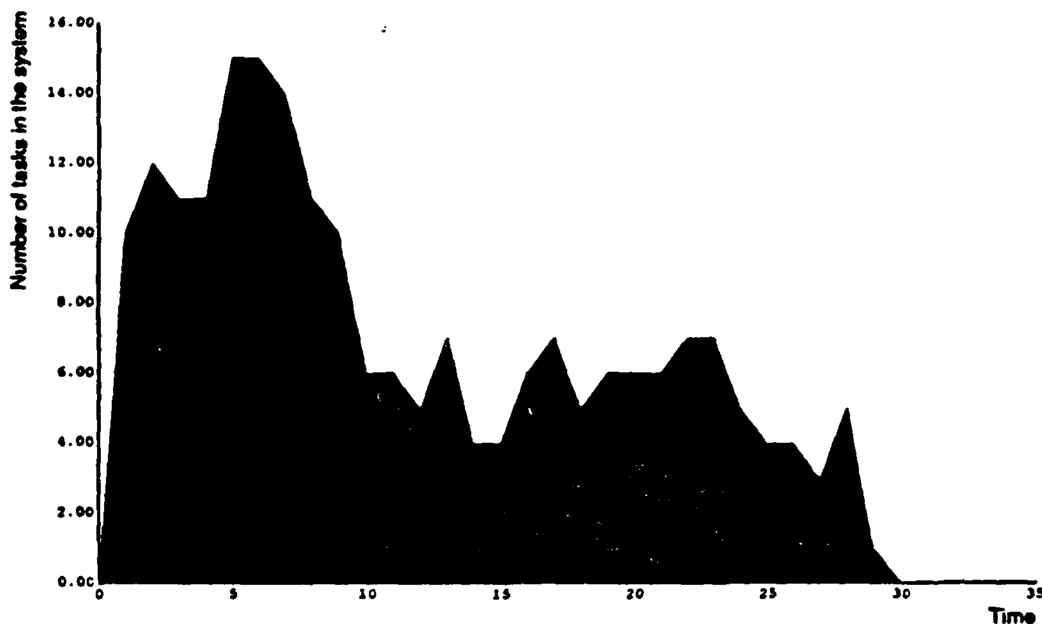


Figure 4-9: Weaver: Number of node activations available for parallel processing as a function of time during a *short* cycle. Each unit of time on the X-axis corresponds to 100µs.

5. Conclusions and Future Work

In this paper we have presented the details of a parallel implementation of OPS5 running on the Encore Multimax. The first observation is that it is important to speed-up an *optimized* sequential implementation, otherwise most of the benefits are lost. For example, speeding-up the Franzlisp implementation by 10-20 fold from parallelism would just bring us to the uniprocessor speed of the C-based implementation. Furthermore, the issues in parallelizing an optimized implementation are different from those in an unoptimized implementation, because only very limited overheads can be tolerated in an optimized implementation.

The second observation we make is that it is possible to obtain significant speed-ups for OPS5 using fine-grained parallelism on a shared-memory multiprocessor. However, this does not work for all programs. The Tourney program, because of the presence of short cycles and cross-products resisted all our attempts to obtain higher speed-up.

Our third observation is regarding the contention for shared memory objects. The average length of the individual tasks in our parallel implementation varies between 100-700 machine instructions for the three programs that we studied. In trying to exploit this fine-grained parallelism, we found that scheduling tasks using a single task queue formed a major bottleneck for the system. We found it essential to use multiple task queues (instead of a single task queue) to obtain reasonable speed-up. For the Rubik program, going from one task queue to multiple task queues increased the speed-up from 6.3-fold to 11.4-fold.

The other variation that we explored to reduce the contention for shared data structures was in the complexity of locks used for hash-based memory nodes. We used both simple spin-locks and complex multiple-reader-single-

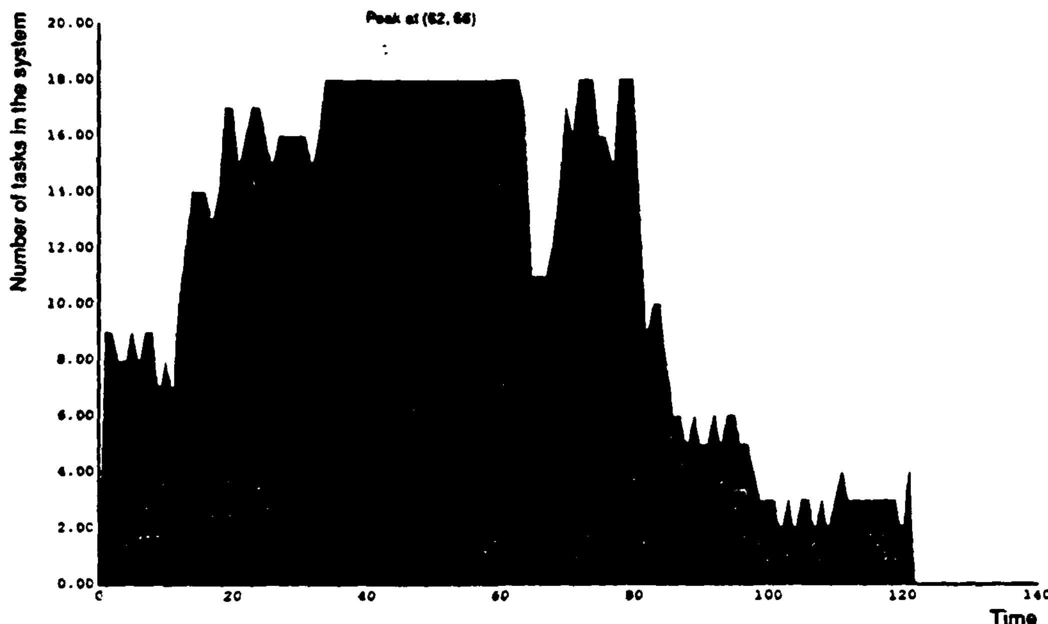


Figure 4-10: Weaver: Number of node activations available for parallel processing as a function of time during a *long* cycle.

writer locks. We observed that special note must be taken of *rare-case* versus *normal-case* execution. Trying to handle rare cases efficiently can slow down the normal case, and can result in overall poorer performance. For example, the provision of complex hash-table locks reduced the contention for the hash-table buckets, but it slowed down the overall execution speed of the Rubik program.

In the future we plan to investigate alternative computer architectures for implementing production systems; especially the message-passing architectures. Our analysis indicates that the message-passing architectures are quite suitable for implementing production systems [6]. Currently, simulations of implementing production systems on such machines are in progress.

Our other direction of investigation has been an exploration of the parallelism in Soar [11], a learning production system. The parallelism in Soar is expected to be higher than OPS5 [5]. Our current implementation of Soar on the Encore Multimax has provided good speedups in the match [17]. The next step there is to parallelize other areas of Soar besides match.

6. Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864, monitored by the Air Force Avionics Laboratory under Contract N00039-85-C-0134 and by the Encore Computer Corporation. Anoop Gupta is also supported by DARPA contract N00014-87-K-0828 and an award from the Digital Equipment Corporation.

References

[1] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin.
Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.
Addison-Wesley, 1985.

[2] P.L. Butler, J.D. Allen, and D.W. Bouldin.
Parallel Architecture for OPS5.
In *Proceedings of the Fifteenth International Symposium on Computer Architecture*, pages 452-457. 1988.

[3] Charles L. Forgy.
The OPS83 Report.
Technical Report CMU-CS-84-133, Carnegie-Mellon University, Pittsburgh, May, 1984.

[4] Anoop Gupta, Charles Forgy, Allen Newell, and Robert Wedig.
Parallel Algorithms and Architectures for Production Systems.
In *13th International Symposium on Computer Architecture.* June, 1986.

[5] Anoop Gupta.
Parallelism in Production Systems.
PhD thesis, Carnegie-Mellon University, March, 1986.
Also available from Morgan Kaufmann Publishers Inc.

[6] Anoop Gupta and Milind Tambe.
Suitability of Message Passing Computers for Implementing Production Systems.
In *National Conference on Artificial Intelligence.* AAAI-88.

[7] Bruce K. Hillyer and David E. Shaw.
Execution of OPS5 Production Systems on a Massively Parallel Machine.
Journal of Parallel and Distributed Computing 3:236-268, 1986.

[8] Rostam Joobbani and Daniel P. Siewiorek.
Weaver: A Knowledge-Based Routing Expert.
In *Design Automation Conference.* 1985.

[9] Peter M. Kogge.
An Architectural Trail to Threaded-Code Systems.
Computer March, 1982.

[10] Edward J. Krall and Patrick F. McGehearty.
A Case Study of Parallel Execution of a Rule-Based Expert System.
International Journal of Parallel Programming 15(1):5-32, 1986.

[11] Laird, J. E., Newell, A., & Rosenbloom, P. S.
Soar: An architecture for general intelligence.
Artificial Intelligence 33:1-64, 1987.

[12] Theodore F. Lehr.
The Implementation of a Production System Machine.
In *Hawaii International Conference on System Sciences.* January, 1986.

[13] Daniel P. Miranker.
TREAT: A New and Efficient Algorithm for AI Production Systems.
PhD thesis, Columbia University, 1987.

[14] Pandurang Nayak, Anoop Gupta, and Paul Rosenbloom.
Comparison of the Rete and Treat Production Matchers for SOAR.
In *National Conference on Artificial Intelligence.* AAAI-88.

[15] Kemal Oflazer.
Parallel Execution of Production Systems.
In *International Conference on Parallel Processing.* IEEE, August, 1984.

[16] Raja Ramnarayan, Gerhard Zimmerman, and Stanley Krolikoski.
PESA-1: A Parallel Architecture for OPS5 Production Systems.
In *Hawaii International Conference on System Sciences*. January, 1986.

[17] Milind Tambe, Dirk Kalp, Anoop Gupta, Charles Forgy, Brian Milnes, and Allen Newell.
Soar/PSM-E: Investigating Match Parallelism in a Learning Production System.
In *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 146-161. July, 1988.

[18] M.F.M. Tenorio and D.I. Moldovan.
Mapping Production Systems into Multiprocessors.
In *International Conference on Parallel Processing*. IEEE, 1985.

Experiences Implementing a Parallel ATMS on a Shared-Memory Multiprocessor

Edward Rothberg and Anoop Gupta
Department of Computer Science
Stanford University
Stanford. CA 94305

November 8. 1988

Abstract

The Assumption-based Truth Maintenance System (ATMS) is an important tool in AI. So far its wider use has been limited due to the enormous computational resources which it requires. We investigate the possibility of speeding it up by using a modest number of processors in parallel. We begin with a highly efficient sequential version written in C and then extend this version to allow parallel execution on the Encore Multimax. a 16 node shared-memory multiprocessor. Our parallel implementation gives speedups of between 4.4 and 6.7 using 14 processors for the ATMS trace files which we examine. We describe our experiences in implementing this shared-memory parallel version of the ATMS. present detailed results of its execution, and discuss the factors which limit the available parallelism.

1 Introduction

The Assumption-based Truth Maintenance System (ATMS) is an important tool in AI. It makes the task of designing a problem solver much easier, removing the need for the problem solver to maintain information concerning derivations which it makes. Without an ATMS, the problem solver must implicitly record which of its assumptions it currently believes to be true and what these assumptions imply. When it wishes to change its assumption set. it must also recompute the set of items which are implied. With an ATMS, the problem solver explores the problem space. informing the ATMS of the assumptions it makes, the items which it wishes to reason about, and the derivations which it makes concerning these items. The ATMS aids in the process by keeping track of which items hold under any given assumption set, thus allowing the problem solver to freely change the set of assumptions which it currently believes. A number of problem solvers have been built which use the ATMS in a number of AI subfields. The ATMS provides a convenient level of abstraction, greatly simplifying the structure of the problem solver.

So far wider use of the ATMS has been limited due to the enormous computational resources which it requires. The ATMS is often the bottleneck in the problem solving process, often having greater computational requirements than the problem solver which it is collaborating with. We investigate the possiblity of speeding up the ATMS by using a modest number of processors in parallel. We begin with a hightly efficient C-based implemention of the ATMS based on the techniques described in [1]. Through a number of modifications to the basic sequential ATMS, we obtain moderate speedup on the three example problem solver trace files which we examine.

The paper is organized as follows. Section 2 presents background information about the ATMS and introduces related terminology. Section 3 presents details of an efficient sequential implementation of the ATMS. Section 4 presents the modifications to the sequential implementation which were necessary to allow parallel execution. Section 5 presents the results of executing the basic parallel implementation. We discuss the bottlenecks encountered and introduce a number of modifications to the basic algorithm to deal with these bottlenecks. Section 6 presents the conclusions which we arrive at based on the observed results.

2 The ATMS

The ATMS serves as a companion to a problem solver, acting as a sort of "truth database". The problem solver feeds beliefs, contradictions, and implications to the ATMS. The ATMS keeps track of what is true under what assumption sets and why. In this section we illustrate how the ATMS is used and introduce the terminology with a brief example. The example problem that we solve is the 3-queens problem. It consists of finding placements for three queens on a 3 by 3 chessboard such that no queen can capture any other.

Everything which the problem solver reasons about is assigned an ATMS *node*. In the 3-queens example we use 10 nodes, one for each of the 9 squares on the chessboard and one goal node to represent the solution. Each chessboard node represents the placement of a queen on the corresponding chessboard square. Some subset of the ATMS nodes are designated to be *assumptions*. These are nodes which are presumed to be true unless there is evidence to the contrary. In the example, the 9 nodes assigned to chessboard squares are the assumptions. We assume that a queen can be placed at each square of the board. Every important derivation made by the problem solver is recorded as a *justification*:

$$x_1, x_2, \ldots \Rightarrow n$$

where $x_1, x_2, \ldots$ are the antecedent nodes and n is the consequent node. In the example, the problem solver tells the ATMS that any set of three queens placed on the board constitutes a solution. Thus, the justifications take the form:

$$position_1, position_2, position_3 \Rightarrow goal_node$$

where $position_i$ is an assumption which corresponds to a queen being on a particular square on the chessboard. An ATMS *environment* is a set of assumptions. A node n is said to hold in environment E if n can be propositionally derived from the union of E with the current set of justifications. An environment is inconsistent (called *nogood*) if the distinguished node $\perp$ (i. e. false) holds in it. In the 3-queens example, we declare any set of assumptions in which the corresponding board positions contain a capturing pair to be nogood. The answer to the 3-queens problem is the set of all consistent environments in which the goal node holds.

In the ATMS, sets of environments play an important role in keeping track of the contexts under which a given node holds. They are used extremely frequently, and consequently we need a concise representation for a them. In our representation, we can take advantage of the fact that if a node holds under environment E, then it also holds under any superset of E. We can therefore represent a set of environments by its smallest members. We choose to represent a set S of environments as a list $(E_1, E_2, \ldots)$, which we call a *minimal environment list*. It has the following properties:

- Every environment in the set S is a superset of some E_i.
- No E_i is a subset of any other.
- No E_i is nogood.

The distinction between sets of environments and sets of assumptions presents a possible source of confusion. For example, consider the environments {A, B} and {A, B, C}. Clearly {A, B, C} is a superset of {A, B}. Yet, the minimal environment list ({A, B, C}) represents a subset of the minimal environment list ({A, B}): the second contains environments which do not have assumption C in them, while the first does not. Please keep this potential source of confusion in mind when we discuss environment supersets and subsets in the remainder of this paper.

The problem solving process involves a dialogue between the problem solver and the ATMS, in which the ATMS receives a stream of requests to create new nodes, new assumptions, new justifications, and to provide information on the environments in which nodes hold. This information can be easily provided if the ATMS maintains with each node n a set of environments, in minimal environment list form, called its *label*. In addition to the minimal environment list properties, each node's label has the following two properties:

```
; an assumption for each board position
Create-Assumption "Queen at 1-1"
.
.
Create-Assumption "Queen at 3-3"

; and a node to represent a solution
Create-Node "Goal"

; all capturing pairs are inconsistent
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 1-2"
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 1-3"
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 2-1"
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 2-2"
Justify-Node "FALSE" by "Queen at 1-2" "Queen at 1-3"
.
.
Justify-Node "FALSE" by "Queen at 3-2" "Queen at 3-3"

; the goal node is implied by any set of 3 assumptions
; (the problem solver discards those sets which will
; obviously not lead to a solution)
Justify-Node "Goal" by "Queen at 1-1" "Queen at 2-3" "Queen at 3-2"
Justify-Node "Goal" by "Queen at 1-3" "Queen at 2-1" "Queen at 3-2"
```

Figure 1: A Simple Formulation of the 3-Queens Problem

- Label soundness - Node n holds in every environment in the label set.
- Label completeness — Every environment E in which n holds is a member of the label.

2.1 The Interface Between the Problem Solver and the ATMS

The four basic operations which the ATMS makes available to the problem solver are:

- Create-Node n — create a new node.
- Create-Assumption n — create a new assumption.
- Justify-Node n by $x_1 x_2 \ldots$ — add a new justification.
- Node-Query n — request the current label of node n.

The problem solving process is a collaboration between the problem solver and the ATMS. In solving the 3-queens problem. the problem solver could indiscriminately feed all of the above mentioned justifications and nogoods to the ATMS and let the ATMS sort through them. Because the ATMS has no problem-specific knowledge, though, this results in a great deal of avoidable work being done. For example, the problem solver knows that a solution to the 3-queens problem will never have 2 queens in the same column or the same row, so it could simply reject any justifications which would obviously not result in a solution without passing them on the ATMS. To keep execution time to a minimum, the problem solver must be careful about the commands it passes to the ATMS. Figure 1 illustrates how the 3-queens problem might be formulated in the ATMS framework. DeKleer discusses in [2] how the problem solver can efficiently interact with the ATMS. In this paper, however, we do not address this issue. We simply deal with how the ATMS can efficiently handle the commands which the problem solver passes to it.

3 Sequential Implementation

We now examine how the sequential ATMS is actually implemented, with emphasis on those aspects of the implemention which are relevant to parallel execution. Since we will be computing the speedups of the parallel implementation based on the execution time of the sequential implementation, we must make sure that sequential version is as efficient as possible. While it may be possible to obtain substantial speedups as compared with an inefficient sequential implementation, such results give little information about how much parallelism is available in the problem. The only way to get a true measure of how much parallelism in the problem is actually being exploited is to begin with an efficient sequential implementation.

This section is organized as follows. Section 3.1 gives a general overview of an efficient ATMS implementation. Section 3.2 describes how set operations are done on minimal environment lists. Section 3.3 presents the data structures used to represent environments, nodes, assumptions, and justifications. Section 3.4 describes the three problem solver trace files which we will examine. We compare the performance of our *sequential implementation* with the performance of an existing ATMS implementation on these three trace files. Section 3.5 describes the environment database, the data structure which is used to keep track of those environments which are consistent and those which are nogood. Section 3.6 then gives a detailed description of the steps involved in computing an environment list cross product. Finally, Section 3.7 gives the details of how the union of two environments is computed.

3.1 Implementation Overview

The overall structure of the ATMS is as follows. The problem solver places Create-Node, Create-Assumption, Justify-Node, and Node-Query messages on a shared command queue The ATMS repeatedly removes available commands from the queue. Given a command, it performs the requested action, restores node label soundness and consistency for all nodes in the inference graph, and is then ready to perform the next command.

Of the four commands which the ATMS makes available to the problem solver, only Justify-Node consumes significant amounts of time. The Create-Node command takes very little time, since at the point at which the node is created it does not participate in any justifications. The Create-Assumption command also takes little time for the same reason. In our 3-queens example, the one Create-Node and 9 Create-Assumption commands simply require the ATMS to initialize the appropriate data structures. The Node-Query command is also computationally inexpensive because of the properties of label consistency, soundness, completeness, and minimality. In order to process a Node-Query command, the ATMS simply returns the current label of the appropriate node.

The ATMS spends the vast majority of its time in processing new justifications. A new justification can cause an enormous amount of label updating and environment propagation. When a new justification

$$x_1, x_2, \ldots \Rightarrow n$$

arrives at the ATMS, the labels of node n and any nodes which depends on node n may no longer be complete. Node n may now be derivable from a new set of assumptions not currently in node n's label because of the new justification. If this is the case, node n's label must be updated. If node n's label changes, then the label of every node which depends on node n may also change. Thus any change to node n's label must be propagated to every successor of node n.

A new justification can also cause new nogood environments to be discovered, potentially causing the node label of any node in the inference graph to become inconsistent. The simplest example of this would be a justification whose consequent is the false node. Conceivably, however, any justification whose consequent node n can derive the false node can cause new nogoods to be generated. In order to restore node consistency, environments which become nogood must be removed from all node labels.

In order to handle propagation of node labels, the ATMS maintains an Update request stack. Any time a node label is changed, Update requests are placed on the request stack, one for each justification which has the modified node as an antecedent. The first step in the processing of a new justification is to push an Update request onto the request stack. The ATMS continues popping Update requests off

of the Update stack, processing the requests, and potentially pushing more requests onto the stack until the stack is empty. This corresponds to a depth first propagation of labels.

A single Update request is processed as follows:

- The set of consistent environments which derive the consequent using the new justification and the new label environments is computed. This set is the intersection of the new label environments of the one antecedent with the labels of the other antecedents of the justification.
- If the consequent is the false node, then all of these environments are recorded as nogood and removed from all node labels.
- Otherwise, these environments are compared against the existing label of the consequent.
- If they are already there, then the propagation due to this justification is complete.
- Otherwise, the consequent node label is set equal to the union of the previous label and the new set of environments.
- The changes to the consequent label, i.e. the set of environments in the label which were not present in the previous label, are propagated to all nodes which depend on the consequent. This is accomplished by creating one Update request for each justification which has the current consequent as an antecedent. The Update requests are pushed onto the request stack.

3.2 Set Operations on Minimal Environment Lists

Adding a new justification requires a number of set operations on sets of environments, including set union and set intersection. The minimal environment list representation allows us to perform these operations quickly. Given two environment sets, S and T, represented as $(E_1, E_2, \ldots)$ and $(F_1, F_2, \ldots)$ respectively, we perform set operation on them as follows:

When we wish to add a new set of environments to the label of a node, we must take the set union of the existing label with the set of new environments. The set union of S and T, in minimal form is the concatenation of the minimal forms of S and T, with all supersets removed. In other words, each E_i is checked against each F_j for subsumption. If some F_j is a subset of E_i, then E_i is not included in the union. Similarly, if E_i subsumes some F_j, then F_j is also not included. All other E_i and F_j are included.

When we wish to compute the effect of a justification on its consequent node, we must find the set intersection of all of the labels of the antecedent nodes. The set intersection of S and T is somewhat more involved than the set union. If all supersets of E_i are in S and all supersets of F_j are in T, then all environments which are supersets of both E_i and F_j are in $S \cap T$. The set of all supersets of both E_i and F_j is the set of all supersets of the union of E_i and F_j (remember that environments are sets of assumptions). For example, the intersection of the supersets of $\{A, B\}$ with the supersets of $\{B, C\}$ is the supersets of $\{A, B, C\}$, which is the union of $\{A, B\}$ with $\{B, C\}$. Thus the intersection of S with T is the set of all supersets of the pairwise unions of E_i with F_j. Thus, in minimal environment list form, this is the cross product of the minimal environment list forms of S and T, again with all supersets removed.

3.3 Data Structures

The efficiency of the ATMS is highly dependent on the data structures and algorithms used in the implementation. A straightforward ATMS implementation can literally take days [8] to solve a problem which a more sophisticated implementation solves in a few minutes. We first present the major data structures used in our ATMS implementation. The data strucutures are simply laid out here with brief descriptions: the purpose of each individual field will be made clear in later sections.

The *environment* data structure has the following fields: (1) *Present:* a bit vector representing the set of assumptions present in the environment. (2) *Constituents:* a linked list of all assumptions present in the environment. (3) *Size:* the number of assumptions present. (4) *Contra:* a flag indicating whether the environment is consistent. (5) *Where:* a linked list of all nodes which contain this environment in

Table 1: Trace file statistics.

	QPE	BUG	8-Q
Nodes	988	1705	131
Assumptions	38	62	64
Justifications	2584	4165	1192
Run time - deKleer's on Explorer I	118	182	?
- ours on MultiMax	40.44	92.08	35.61
- ours on VAX 3200	15.45	34.21	13.81

Table 2: Runtime breakdown - Our Implementation on MultiMax

	QPE	BUG	8-Q
Run time (s)	40.44	92.08	35.61
Time spent on Justifys	32.34	79.00	32.21
Time spent on file access	6.46	10.68	2.42
All other time	1.64	2.42	0.98

their labels. (6) *Orthogonal:* a bit vector representing the set of assumptions which, if added to the environment, would result in a nogood environment.

The *node* data structure has the following fields: (1) *Label:* the node's label. (2) *Assumption:* a pointer to the node's assumption fields, if the node is an assumption. Empty otherwise. (3) *Justifications:* a list of the justifications in which the node is the consequent. (4) *Consequences:* a list of justifications in which the node is an antecedent.

The *assumption* data structure has the following fields, in addition to its node fields: (1) *Binary:* a bit vector representing the set of all binary nogoods this assumption participates in. If bit j is 1 in the Binary field of assumption i, then the environment {i, j} is nogood. (2) *Nogoods:* a table of all minimal nogood environments in which the assumption belongs.

The *justification* data structure has the following fields: (1) *Antecedents:* a list of antecedent nodes. (2) *Consequent:* the consequent node.

3.4 The Trace Files

We present the results of executing three problem solver traces on our ATMS. These traces were given to us by Johan deKleer. They were generated by monitoring the interaction between an actual problem solver and an ATMS, and dumping the observed interaction into a trace file. The traces are:

- QPE, from a problem solver created by Ken Forbus [5] which solves Qualitative Physics problems.
- BUG, a trace which led to a bug in some ATMS implementation.
- 8-Q, from a problem solver which solves the 8-queens problem. This formulation of the N-Queens problem differs somewhat from the one described earlier in this paper.

Table 1 provides information on the three traces. It also provides the runtime for the three traces, both for the LISP-based ATMS implementation of deKleer [1] and for our C-based implementation. The time quoted for deKleer's ATMS is from execution on a Texas Instruments Explorer I lisp machine. The time quoted for our implementation is from execution on a single processor of an Encore Multimax multiprocessor. The Encore MultiMax is a 16 node, shared-memory multiprocessor, with an NS 32032 (0.75 MIPS) microprocessor at each node. We also include the runtime on a more widely available machine, a DEC VaxStation 3200, for reference purposes. These times include all costs involved in processing the trace files from beginning to end, including the time spent processing the ATMS commands and the time spent reading the trace files from disk.

In Table 2 we give a breakdown of where time is spent in our implementation. File access time accounts for a substantial portion of the runtime, a portion which is not relevant when measuring true

ATMS performance. We will therefore ignore file access time in evaluating parallel ATMS performance. Also, if we ignore file access time, all but an extremely small amount of the runtime is spent processing Justify-Node commands. Since they are the clear bottleneck, we will concern ourselves strictly with Justify-Node commands for the remainder of this paper. All runtimes cited in the future will measure only the amount of time spent processing Justify-Node commands.

3.5 The Environment Database

An ATMS environment has a large amount of information associated with it. The environment is either consistent or nogood. It could appear in the labels of many nodes, or it could appear in none. When the ATMS computes the union of two environments, it needs access to the information associated with the resulting environment. The information could be recomputed each time the environment is encountered, but some of the information is quite expensive to gather. In order to avoid having to recreate this information, each encountered environment is given a unique physical representation in memory. In other words, if two nodes have an environment E in their labels, they both really have pointers to the unique structure representing the environment E. The unique representation, when combined with a method for finding this representation for a given environment, allows us to do expensive checks once per environment, not once per encounter.

The method we choose for quickly finding the unique representation of a given environment is an environment hash table. Every environment which is encountered at any time in a problem execution is stored in this hash table. When a new environment is encountered, the hash table is checked to see if the environment has been encountered before. If it has not, the environment is added to the table. In this way, the ATMS can store information about environments which can be quickly retrieved if needed.

When creating new consistent or nogood environments, the ATMS also needs quick access to large sets of existing environments. For example, when a previously undiscovered environment is encountered, it must be checked for consistency. The ATMS must check the new environment against all nogood environments which are smaller than it. If the environment is subsumed by some nogood, then it is clearly also nogood. Similarly, when a new nogood is discovered, all consistent environments which are larger than this nogood must be checked for subsumption. Any consistent environment which is subsumed becomes nogood.

The data structure which seems to best serve these purposes is a pair of tables. Each table consists of an array of lists of environments, sorted by environment size. Thus to find all environments of size n, we must simply traverse the list in position n of the array. One table, the *Consistent* table, holds all consistent environments encountered. The other table, the Minimal NoGood (*MNG*) table, holds the set of nogoods which are not subsumed by any other nogood. Minimal nogoods are kept in order to keep environment consistency checks as quick as possible; an environment which is subsumed by a nogood is clearly also subsumed by any subset of that nogood.

We make two modifications to the simple MNG table for efficiency. First, we handle unary and binary nogoods as special cases. The assumption data structure has a field entitled Binary which keeps track of unary and binary minimal nogoods. If the environment $\{i, j\}$ is nogood, then bit i in the Binary field of assumption j and bit j in the Binary field of i are set. If the environment $\{i\}$ is nogood, then bit i of the Binary field of i is set. The second modification involves the Nogoods field of the assumption data structure. Any minimal nogood environment in the MNG table will also be in the Nogoods table of each assumption in the environment. These two modifications allow the ATMS to find all minimal nogoods containing a given environment extremely quickly.

The Consistent and MNG tables form what we call the *environment database*. The environment database, together with the environment hash table, makes the following frequent operations extremely fast:

- Find a particular environment, with all its associated information.
- Find all consistent environments smaller (or larger) than a given environment.
- Find all minimal nogoods which are smaller than a given environment.

Finally, the most prevalent operation in the ATMS is the subset test. The environment representation must therefore be chosen so that subset testing is extremely fast. A bit vector representation works extremely well. A one in bit i of the vector indicates the presence of assumption i in the environment. The bit vector representation allows subset testing by simply ANDing the bit vector of one environment with the complement of the bit vector of the other environment. A bit vector representation also allows fast hash function computation.

3.6 The Cross Product

When we handle an Update request, we need to compute the cross product of a number of minimal environment lists, as was described previously. Assume we wish to take the cross product of n minimal environment lists $l_1, l_2, \ldots, l_n$, with l_1 being the incremental update. We begin the cross product computation by first checking to make sure that each l_i is non-empty. If any list is empty, then the cross product is empty.

Next we loop through each list, creating m_i, the cross product of l_1 through l_i. We begin with $m_1 = l_1$, and at each iteration we will compute $m_{i+1} = m_i \times l_{i+1}$, where both m_i and m_{i+1} are in minimal environment list form. We do this by taking the union of each environment E in m_i with each environment F in l_{i+1}, using the method for finding unions to be described in the next section. The resulting list is then minimized.

We can greatly decrease the amount of time it takes to compute m_n by using the following two techniques. First, if some environment E in m_i is subsumed by some environment in the consequent of the justification which we are updating, then clearly every environment in $m_{i+1} \ldots m_n$ which is generated from E will also be subsumed by this environment. We therefore check each environment in m_i against each environment in the label of the consequent and discard those which are subsumed. Line 13 in Table 3 shows the number of environments which are discarded in this way for the three trace files.

Second, consider taking the cross product of m_i with l_{i+1}. If some environment E in m_i is subsumed by some F in l_{i+1}, then clearly E will be in m_{i+1}. Since all environments which would result from taking the union of E with some environment in l_{i+1} are supersets of E and since E is in m_{i+1}, none of the resulting environments will be present in m_{i+1}. We therefore check each E in m_i for subsumption against each F in l_{i+1}. If E is subsumed, then we can simply place it into m_{i+1}, and not take the union of E with each environment in m_{i+1}. Line 14 in Table 3 shows the number of times that this occurs for the three trace files.

If we compute the cross product, using these two techniques, the result is a minimal environment list which represents the change to the label of the consequent node n. If the consequent is not the FALSE node, we add each environment in our cross product to the label of node n. We must now restore minimality in the label by checking every environment previously in the label for subsumption against every environment just added to the label. We then propagate the cross product list, which represents the changes to the label of node n, to every justification which has node n as an antecedent.

If the consequent is the FALSE node, then our cross product list is a set of environments which were previously consistent but have just become nogood. We add them to the MNG table, and sweep through the Consistent and MNG tables looking for subsumed environments. If an environment in the Consistent table is subsumed, it is removed from the table and from the labels of all nodes which contain it (found in the Where field of the environment). If an environment in the MNG table is subsumed, it is removed from the table.

3.7 The Union of Two Environments

Computing the union of two environments is an extremely frequent and potentially extremely costly operation in the ATMS. In most ATMS problems, the vast majority of all unions result in a nogood environment (94%, 97%, and 83% for QPE, BUG, and 8-Q, respectively). Table 3 shows the empirical numbers for the three trace files. Line 1 gives the total number of unions computed, with Lines 2 and 3 giving the number of those which result in consistent and nogood environments, respectively. It is therefore to our advantage to have a quick check to see if the result of a union is a nogood. The

Table 3: Results of environment unions.

	QPE	BUG	8-Q
1. Total unions	44598	135851	16440
2. Consistent	2636	4503	2776
3. Nogood	41962	131348	13664
4. Total adds	46909	141788	16440
5. Ortho	40499	128887	0
6. Binary	973	1149	13664
7. Same	1793	4089	0
8. Exist OK	2326	3976	0
9. Exist NG	236	755	0
10. Non-binary	254	557	0
11. New env	828	2375	2776
12. Imm. Ortho	39958	127050	0
13. Old	2580	2566	0
14. Bypass	1615	2853	0

method of union computation which seems to allow the fastest recognition of nogood environments is an assumption by assumption method. That is, given two environments E_1 and E_2, we compute the union by successively adding the assumptions in E_2 into E_1, computing an intermediate environment at every step. The union function returns either a consistent environment E_3, which is the union of E_1 with E_2, or it returns nothing, indicating that the union of E_1 with E_2 is nogood. Since nogoods can never appear in node labels, they do not have to be retained. The union computation is therefore complete as soon as we know that the union will be nogood. We begin the union computation by making E_1 the larger environment, the one with more assumptions. The environments are swapped if this is not true. If both are the same size, we make the one with the larger hash function E_1. This step decreases the number of assumptions which need to be added. It also assures us that if we compute the same union more than once, the steps we do the first time will be repeated each successive time.

The next step is to begin a loop through all n members of E_2. At each iteration i of the loop, we have an environment F_i which represents the result of adding the first i assumptions from E_2 into E_1. If F_i becomes nogood at any point, we may break out of the loop and quit. We begin with $F_0 = E_1$. At the beginning of each iteration we have some consistent F_i and the ith assumption of E_2, A_i, which we wish to add to it. At the end of the iteration we either know that the union is nogood or we have some consistent F_{i+1}, which is the union of F_i with A_i. F_n is the union of E_1 with E_2. Line 4 in the table gives the total number of iterations of this loop which are performed.

Our first step within iteration i of the loop is to do a quick check to determine whether the union could possibly result in a consistent environment. We do this by doing a bitwise AND of the Present field of E_2 with the Orthogonal field of F_i (see section 3.3 for a description of these fields). If the result is non-zero, then we know that some member of E_2, when added to F_i, would yield a nogood environment. Since this nogood environment would clearly be a subset of $E_1 \cup E_2$, we then know that $E_1 \cup E_2$ is nogood and we can quit (Line 5 in the table). If the result is zero, however, it tells us nothing and we proceed.

Next we check for binary nogood subsumption of F_{i+1}. Since F_i is consistent, we only need to check F_{i+1} against those binary nogoods which contain assumption A_i. Thus we can check binary subsumption by taking the bitwise AND of the Present field of F_i with the Binary field of A_i. If the result is non-zero, then some assumption in F_i participates in a binary nogood with A_i, and thus F_{i+1} is nogood (Line 6 in the table). Since we also learn that adding A_i to F_i yields a nogood, we set the bit corresponding to A_i in the Orthogonal field of F_i. If the result is zero, again we proceed.

Next we check to see if A_i is a member of F_i. We do this by extracting the bit corresponding to A_i from the Present field of F_i. If it is set, then $F_{i+1} = F_i$ and this iteration is complete (Line 7 in the table).

Otherwise we form a partial environment structure for F_{i+1}, with all fields except the Constituents

field complete. The Present field for F_{i+1} is equal the Present of F_i with the bit for A_i set. We compute the hash function for F_{i+1}, and then check for the existence of this environment in the environment database. If it exists and is consistent, then this iteration is complete (Line 8 in the table). If it exist and is nogood, then the entire loop is complete (Line 9 in the table). In this case, we may again set the bit for A_i in the Orthogonal field of F_i. If it does not already exist, then we proceed.

If we've gotten this far, we know that our F_{i+1} will be added to the environment hash table, so we fill in the Constituents field by adding A_i to the Constituents field of F_i. We now add this environment to the table.

Next we check F_{i+1} for subsumption by a non-binary nogood. As with the binary nogood check, since we know that F_i is consistent we only need to check F_{i+1} against nogoods which contain A_i. We check F_{i+1} against every non-binary nogood smaller than it which contains A_i, which can be found in the Nogoods field of the assumption data structure for A_i. If it is subsumed by some nogood, then the loop is complete (Line 10 in the table). Again, we may set the bit corresponding to A_i in the F_i's Orthogonal field. Otherwise, we know that F_{i+1} is consistent. We add it to the Consistent table and the iteration is complete (Line 11 in the table).

While this seems like a somewhat cumbersome way of computing the union, in practice it is extremely effective in recognizing unions which will result in nogood environments quickly. Line 12 in Table 3 shows the number of unions which can be aborted after the first test against the Orthogonal vector. One can see that a simple bit vector AND successfully recognizes most unions which will result in a nogood.

This concludes our discussion of an efficient sequential implementation of the ATMS. As was discussed in section 3.4, our implementation is quite competitive with existing ATMS implementations. We use the sequential implementation which we have described as the basis of comparison for the parallel implementations which we describe in the remainder of this paper.

4 Modifications for Parallel Implementation

We now discuss the modifications which are necessary to allow the preceding algorithm to be executed in parallel. Our goal is to exploit as much parallelism as possible, but we can not afford to introduce a large amount of redundant work in doing so. When designing algorithms for massively parallel processors, it is possible and often necessary to radically change the data structures and algorithms from those which would be used on a sequential implementation. The increase in available parallelism which these changes bring about often outweighs the increased amount of work which is done. However, since we will be executing this algorithm on a modest number of processors, our speedups will suffer if the parallel implementation does a large amount of work which the sequential implementation does not do. We therefore do not stray far from the data structures and algorithms used in the efficient sequential implementation.

4.1 Division of Work

The overall structure of our parallel ATMS is quite similar to the structure of the sequential ATMS. The ATMS and the problem solver run concurrently, sharing commands and data through a shared command queue. The problem solver places Create-Node, Create-Assumption, Justify-Node, and Node-Query messages on the queue. The problem solver blocks and waits for a reply after it places a Node-Query message on the queue. A number of processors are allocated to work on the ATMS. The ATMS processors pull commands off the queue and perform the requested actions. In order to allow a greater amount of parallelism, we no longer require that node labels be made sound and complete at the completion of each command. This requirement would necessitate the synchronization of all processors after each command, an operation which would greatly constrain our ability to distribute work among the processors. We now only require that labels be made sound and complete before a Node-Query command is answered. Thus, Node-Query commands are now somewhat expensive, since they require a global synchronization. Create-Node and Assume-Node messages again require very little work to be done, and are dealt with quickly. Justify-Node messages are the source of almost all of our parallelism. Since they require by

far the most computation time, they are the commands which afford the most oppurtunity to distribute work.

In order to decrease contention for tasks, each processor has its own Update request stack. When a processor completes a task, it looks for a new task in the following places. First, it checks it's own Update request stack. If it is empty, then the processor checks the global command queue. If the next command on the command queue is a Node-Query (or if the command queue is empty) the processor becomes idle. When all processors are idle, one processor processes and removes the Node-Query command, thus unblocking the problem solver and allowing the problem solving process to proceed. We call this Algorithm A1. We later provide variations of this basic algorithm.

4.2 Locks

In our shared memory implementation, all the processors access the same data structures. We therefore need a number of mutual-exclusion locks to control simultaneous access to shared data. We begin by using straightforward locking techniques, and later modify our approach based on the observed bottlenecks.

The environment hash table is locked by bucket. Whenever a processor wants to do either an environment lookup or an environment addition, it must obtain a lock on the appropriate bucket before it may access anything in the bucket. Since there are thousands of buckets and, for now, at most 16 processors and very little time is actually spent inside the lock, contention for the hash table buckets is not a problem.

Each environment has a lock to control access to its Contra flag and its Where field. The lock is used to enforce the following conditions:

- No nogood environment may be added to a node's label.
- When an environment becomes nogood, it must be removed from the label of every node which contains it.

The above conditions are also used to avoid redundant work. When a processor wishes to change an environment's status to *nogood*, it first obtains a lock on the environment. It then checks the environment's Contra flag. If the flag is set (i.e. the environment is nogood), then some other processor must have already discovered that this node is nogood and the processor can stop; any work done with this environment would be redundant. Otherwise, the Contra flag is set and the lock is released. The environment is then removed from the label of every node in which it appears. When a processor wishes to add an environment to a node label, it obtains the environment lock and checks the Contra flag. If the flag is set, then another processor has discovered that the environment is nogood and it should therefore not be added to the node label. If the flag is not set, the environment is added to the node label and the environment lock is released. By using the lock in this way, we are assured that no node label can contain a known nogood. Since a typical ATMS application generates thousands of environments, contention at this point is usually not a problem.

Node labels are accessed and modified by many processors, thus we must provide locks to protect them. When an Update request is being processed, and a node is an antecedent to the justification, the node's label is accessed. Similarly, the label of the consequent of the justification is also accessed. Since computing the label cross product could take an enormous amount of time, we would like to avoid holding the node lock for the duration of the cross product. We therefore choose to lock the node, copy the label, and immediately release the lock. An antecedent label can simply be copied because any change to the label will be propagated to this justification. While this can create redundant work, the resulting answer will still be correct. A consequent label can be simply copied for the following reason. The only way in which an environment is removed from the label of a node is if it becomes nogood or if it is subsumed by a new label. If an environment becomes nogood, then clearly anything subsumed by it also becomes nogood. If an environment is subsumed, then clearly anything subsumed by it will also be subsumed by the new environment. In either case, it is valid to discard cross product environments which are subsumed by consequent label environments.

When revising the label of a node, the node is locked, and the environments in the current node label are checked for subsumption against the new environments and vice-versa. The node label is revised, any changes in the label are recorded, and the lock is release. Again, since there are thousands of nodes contention is usually not a serious problem.

Contention for environment and node locks can be a problem, however, when many processors are working in the same part of the inference graph. Since environments are generated entirely by propagation, it is likely that if two processors are working on tasks which resulted from the same node revision, they will encounter identical environments more often than if they were working on unrelated activations. Similarly, if two processors are working in the same part of the graph, they are more likely to want to access the same node label. In order to avoid this type of contention, it is desirable for the processors to be well distributed throughout the inference graph.

4.3 The Environment Database

We now discuss the modification necessary to allow concurrent access to the environment database. The modifications we have discussed so far have been relatively local. They have involved such changes as a lock on a bucket in a table, or a lock on a single environment or node. The environment database, however, is a very global structure. It keeps track of the consistency of all environments in the entire problem. A single change could conceivably affect every environment in the environment database. The environment database must allow the following operations:

- Determine whether a new environment is consistent.
- Add a new consistent environment.
- Add a new nogood and find all previously consistent environments which are now nogood.

It must also keep the database self-consistent while these operations are occurring. Since the ATMS spends much of its time creating new environments and checking them for consistency, we cannot tolerate a high latency on consistency checking. At the same time, however, most new environments which are encountered are nogood, so to avoid superfluous work we want a new nogood to be recorded as soon as possible.

We initially used a single global lock to control access to both the Consistent and MNG tables. When an environment needed to be checked for consistency, the processor obtains the global lock, checks the environment against the MNG table, and releases the lock. When a new consistent environment is added, the processor obtains the lock, adds the environment to the Consistent table, and releases the lock. When a new nogood is registered, the processor obtains the lock, checks all consistent environments for subsumption, changes those which are subsumed into nogoods, and releases the lock. Since the ATMS spends a substantial percentage of its time within this lock (3-15% for the three traces), this global locking approach appears somewhat suspect.

5 Results

We now present the results of executing the three problem solver traces on our parallel ATMS. Because Node-Query information was not required when the traces were originally generated, these traces do not record this command. The absence of this command does not affect the performance of the sequential ATMS significantly, since Node-Query commands take so little time to execute. In our parallel ATMS, however, the lack of these commands obviates the need for global synchronization. Thus, the results we present here are optimistic, as the synchronization is done only at the completion of the entire trace. In applications where Node-Query commands are frequent, one would expect less available parallelism.

The ATMS traces we examine seem to present abundant opportunites for parallelism. Their inference graphs are extremely large, with thousands of justifications capable of being distributed among the processors (see Table 1). The only limiting factor would appear to be the global lock on the Consistent

Table 4: Task times for algorithm A1

	QPE	BUG	8-Q
Tasks	2584	4165	1192
Ave task time (s)	0.015	0.019	0.028
Max task time (s)	2.42	35.31	0.36
Total runtime (s)	39.34	82.31	33.72

Table 5: Task times for algorithm A2

	QPE	BUG	8-Q
Tasks	18780	16576	3308
Ave task length (s)	0.002	0.005	0.010
Max task length (s)	0.86	6.88	0.34
Total runtime (s)	39.34	82.31	33.72

and MNG tables. However, if we examine Figure 3 we see that the speedup obtained for Algorithm A1 is disappointing. The speedup is greatly below what one would expect, even given the global lock. The sequential ATMS spends 3%, 15%, and 6% of its time within the lock for QPE, BUG, and 8-Q, respectively. If this were the only parallelism limitation, we would expect speedups of 7 or more. Clearly, parallelism is being limited in some other way.

The most serious bottleneck appears to be processor idle time. Figures 4 through 6 show the percentage of time each processor spends doing useful work as compared to the percentage spent waiting on locks and the percentage spent idle. Note that the speedup obtained is not equal the product of the processor utilization with the number of processors used. This is due to number of factors. First, o speedup numbers are obtained by dividing the parallel execution time by the execution time of the be sequential implementation. There are a number of overheads involved in the parallel implementa such as environment list copying and redundant checks, which can reduce the speedup when compared to a sequential implementation without these overheads. Second, the parallel ATMS does not necessarily do the same amount of work that the sequential ATMS does. For example, the parallel ATMS can process the justifications in a different order than the sequential ATMS. While the answer arrived is the same, the amount of progation necessary to get to this answer may differ. Third, there a number of hardware issues, including bus bandwidth and cache interactions, which can preclude linear speedups. These issues are not reflected in the utilization graphs which we present.

From examining Figures 4 through 6, it becomes clear that we have a problem with the distribution of work among the processors. Processors are spending a large amount of time idle, without a task to execute. What we have here is essentially a bin-packing problem. We have a certain number of tasks of varying size to execute, and we wish to divide them among a number of processors so that each processor takes approximately the same amount of time to complete them. This near equal division of tasks is normally quite possible given a large number of tasks to distribute; the large number of tasks serves to smooth out the variations in grain size. However, two factors make this untrue in Algorithm A1. First, the variation in grain size is enormous. In the BUG trace, for example, the processing of one single justification accounts for more than 40% of the run-time of the trace (see Table 4). Second, as the trace progresses the size of the inference graph increases. The amount of work required to process a single justification depends heavily on how much label propagation must be done. In the early stages of the trace, the small size of the inference graph limits the amount of propagation necessary. As the trace progresses, however, the graph becomes larger, with the potential for more necessary propagation. The grain size therefore grows as the trace progresses, and one would expect the enormous grains to be near the end of the trace. The combination of some extremely large grains with the tendency for the large grains to be towards the end of the trace combine to make it extremely likely that one processor will be stuck with a large grain while the other processors have nothing to work on.

In order to alleviate the grain size problem, we decrease the task size. Instead of each problem solver issued command being a single task, we now consider each Update request to be a task. In Algorithm A1, once the command queue becomes empty the processor simply quits. Now, in Algorithm A2, an idle

All Other Locks

Node Locks

Environment Database Locks

Idle Time

Processor Utilization

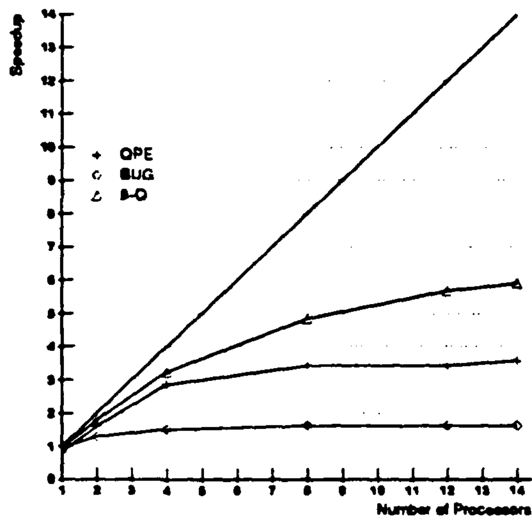


Figure 3: Speedup for Algorithm A1

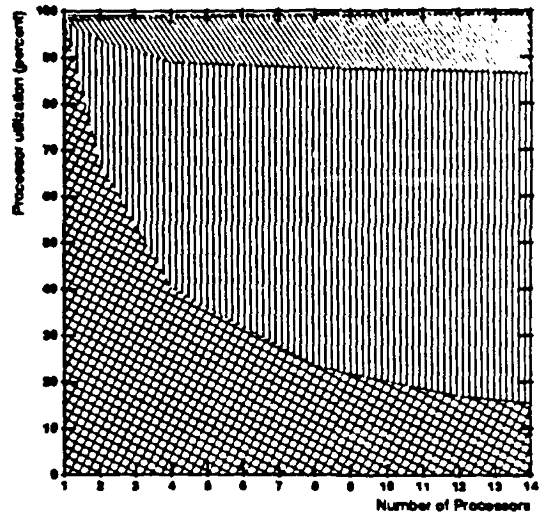


Figure 5: Processor utilization for BUG, Algorithm A1

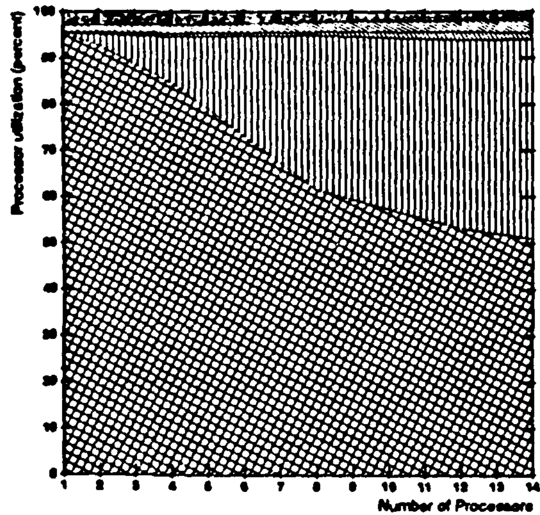


Figure 4: Processor utilization for QPE, Algorithm A1

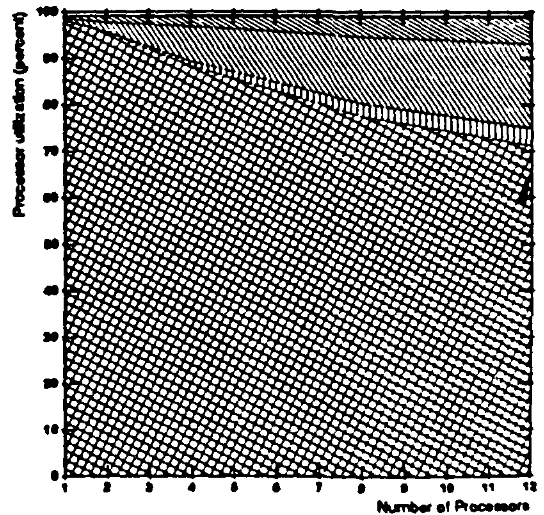


Figure 6: Processor utilization for 8-Q, Algorithm A1

processor attempts to steal an Update request from the Update stacks of the other processors. In this way, work can be distributed among the processors even after the command queue has been emptied. The only cost for this modification is the introduction of contention in the Update stacks. Comparing Tables 4 and 5, we see that by decreasing the task size we have greatly increased the number of tasks and greatly reduced both the average and maximum task size. Figures 8 through 10 show that while idle time has been greatly decreased from that of Algorithm 1, it is still substantial for the BUG trace. This is mainly because the largest task still takes 6.88 seconds, which is 8% of the total runtime. The net result of our modification (Figure 7) is that the speedup is greatly increased from that of Algorithm A1, but it is still far from ideal.

The most serious bottleneck in our parallel implementation now appears to be the environment database lock. In order to increase concurrency in the environment database, we introduce another variation on our basic algorithm. In Algorithms A1 and A2, only a single processor may access the database at one time. Our modification, which we call *Modal access*, allows a number of processors to access the table concurrently, while still maintaining the stringent consistency requirements of the environment database.

The problem in allowing concurrent access to the database comes from the potential simultaneous additions of a consistent environment and a nogood environment. In order to add the consistent environment to the database, we must know that it is not subsumed by any environment in the MNG table. To add the nogood to the database, we must remove all environments which are subsumed by it from the Consistent table. These requirements seem to place serious sequentiality constraints on modifications to the database. In order to avoid these constraints, we add to the environment database a mode of access indicator. The three access modes are:

- Mode 0 — No processor is currently accessing the database.
- Mode 1 — Only consistent environments may be added to the database.
- Mode 2 — Only nogood environments may be added to the database.

In order for a processor to add a new consistent of nogood environment, the environment database must be in the appropriate mode. If a processor needs the table to be in Mode 1, it calls a procedure called Adder(). Adder() waits until the database is in either Mode 0 or Mode 1. If the database is in Mode 0, it changes the mode indicator to Mode 1. It increments a counter of how many processors are within the database, and then proceeds. Once this processor is finished using the database, it calls the procedure ReleaseAdder(). This procedure decrements the counter, and if the counter is now zero it changes the mode indicator back to 0. The procedures Deleter() and ReleaseDeleter() are defined identically except that they move the database into Mode 2.

If a processor wishes to add a new environment to the database, it first calls Adder() to bring the database into Mode 1. It then checks the environment for consistency. If the environment passes the check, it is added to the Consistent table. Since the environment database is in Mode 1, the processor is guarenteed that no nogoods are being added. Thus if the environment passes the consistency check, the environment will remain consistent until the Mode is changed. Once the consistency check has been made, the processor is finished using the database and calls ReleaseAdder().

If a processor wishes to add a new nogood, it calls Deleter() to bring the database into Mode 2. The processor then adds the nogood to the database, and then it sweeps through the Consistent and MNG tables flushing out all subsumed environments. Since no consistent environments are being added, we can be assured that we will check all consistent environments. While it is true that nogood environments can be added at this point and we can consequently miss one which is subsumed, this situation is sufficiently rare and harmless that we do not need to be concerned with it. Remember that the sweep through the MNG table is for efficiency reasons only, and it does not affect the correctness of the algorithm. Once the sweeps through the two tables are complete, the processor calls ReleaseDeleter().

We can modify the above slightly to increase concurrency. When new nogood environments are generated, they usually come in lists. We can therefore distribute the nogoods in a single list among a number of processors. This is accomplished by keeping a global list of nogoods to be added. When a processor has a list of new nogoods to be added, it adds them to this list, calls Deleter(), and then goes

All Other Locks
Node Locks
Environment Database Locks
Idle Time
Processor Utilization

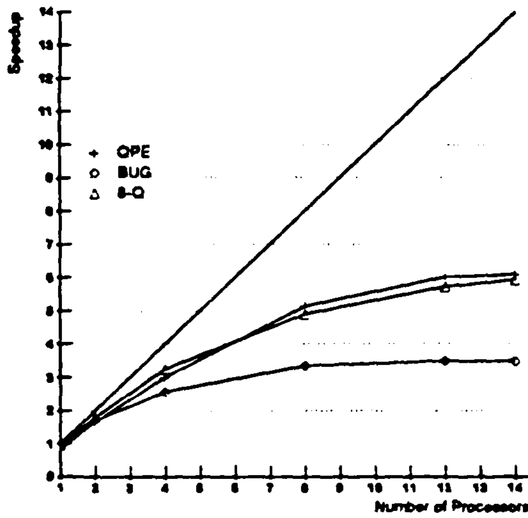


Figure 7: Speedup for Algorithm A2

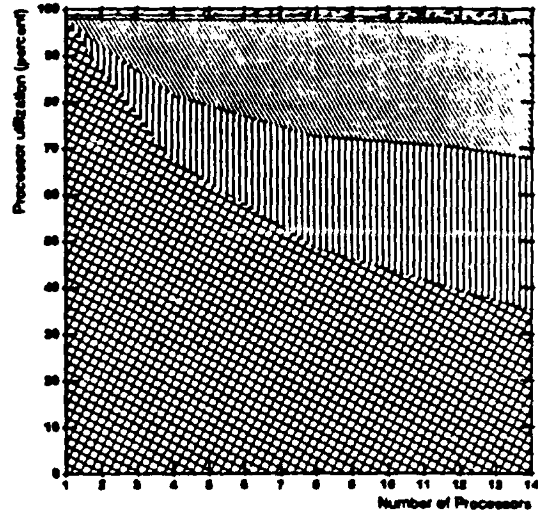


Figure 9: Processor utilization for BUG. Algorithm A2

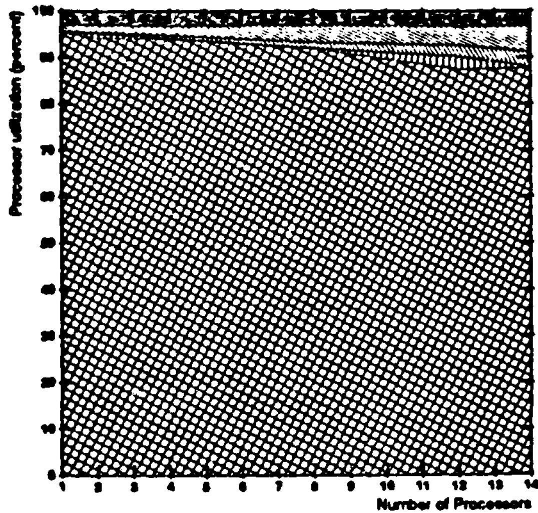


Figure 8: Processor utilization for QPE. Algorithm A2

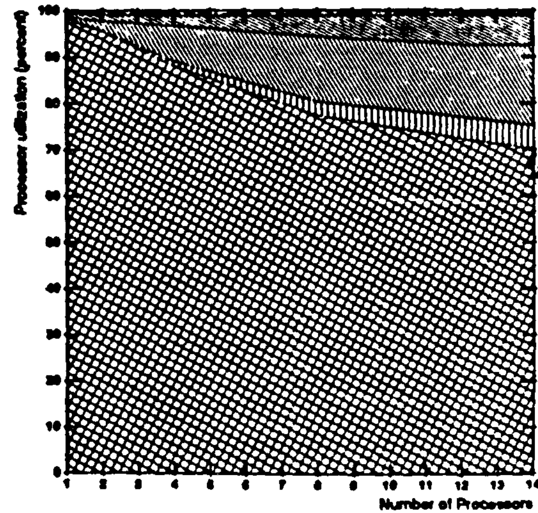


Figure 10: Processor utilization for 8-Q. Algorithm A2

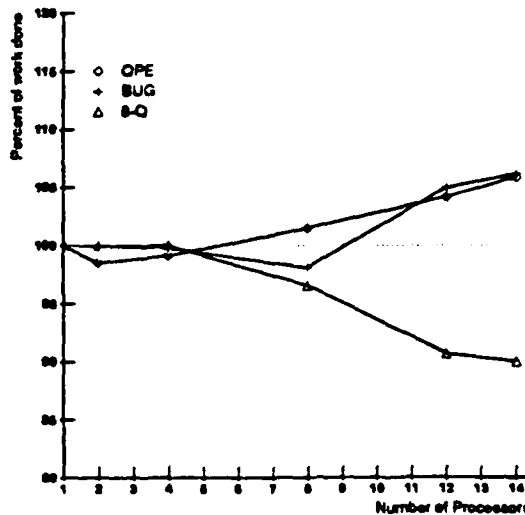


Figure 2: Amount of work done (as a percentage of that for P=1)

into a loop, pulling off nogoods from this list until the list is empty. Now, when a processor calls Adder() and finds the database to be in Mode 2, instead of simply waiting for the Mode to change, the processor also pulls nogoods off the global list and processes them.

The speedups obtained from Algorithm A3 (Figure 11) are still far from ideal. While contention for the environment database is greatly reduced, it is still substantial. We also still have a substantial speedup reduction due to processor idle time.

We have yet to examine one possible cause of reduced speedup in the parallel implementation, redundant work. In the ATMS, it is difficult to establish a measure of how much "work" is being done. There are a number of routines which are called often and take large amounts of time, yet no one routine dominates the others. One routine, the subset test routine, appears to be a reasonably accurate measure of how much work is being done. Subset testing accounts for more of the runtime of the sequential ATMS than any other routine. Also, many of the other routines which take time do large numbers of subset tests. If these routines are being called more frequently, this would be reflected in the number of subset tests done. Figure 2 gives a picture of how many subset test are done for the 3 traces. Though the subset test numbers show interesting trends as the number of processors grows larger, the differences for less than 14 processors are not significant. According to our subset measure of work, the parallel ATMS does between 90% and 106% of the work of the sequential ATMS for 14 or fewer processors.

5.1 Other Approaches

Variation in grain size is still a problem in our implementation. Furthermore, the problem would be much more severe if Node-Query commands were more frequent. One possible way to further decrease the grain size would be to split Update requests into smaller pieces. In Algorithm A3, an Update request contains a list of new environments which have been added to the label of an antecedent. In order to decrease the size of a single grain, we could split this list into many smaller lists. We could use a heuristic to determine approximating how long an Update task will take. Depending on the estimate, the list can be split so that other processors will not go idle while this task is being executed. In the extreme, Update requests can be split into single new antecedent environments.

Performing Updates with smaller lists of environments can generate a large amount of avoidable work.

All Other Locks
Node Locks
Environment Database Locks
Idle Time
Processor Utilization

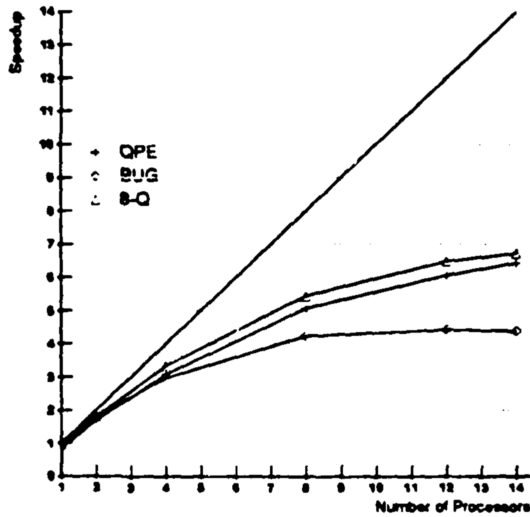


Figure 11: Speedup for Algorithm A3

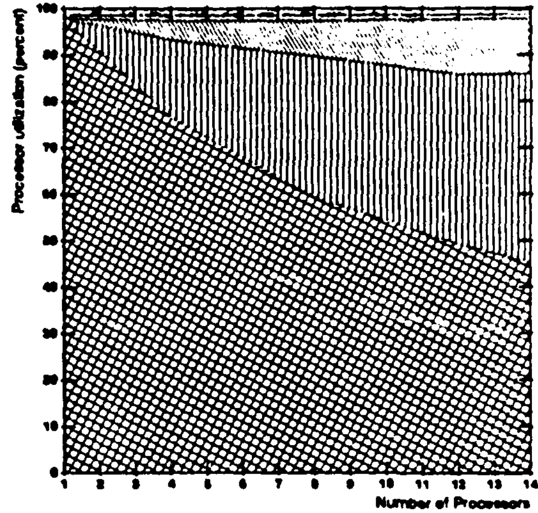


Figure 13: Processor utilization for BUG, Algorithm A3

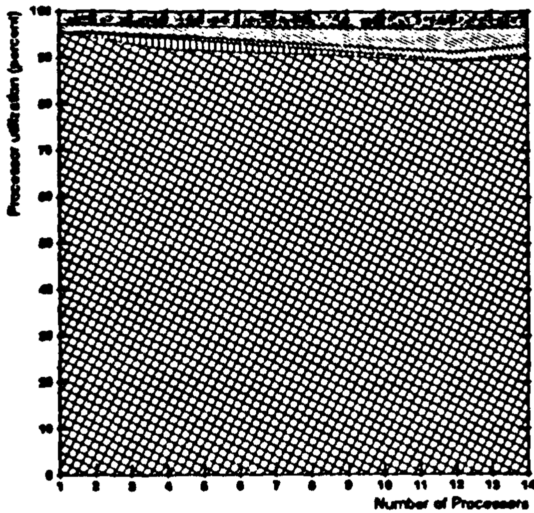


Figure 12: Processor utilization for QPE, Algorithm A3

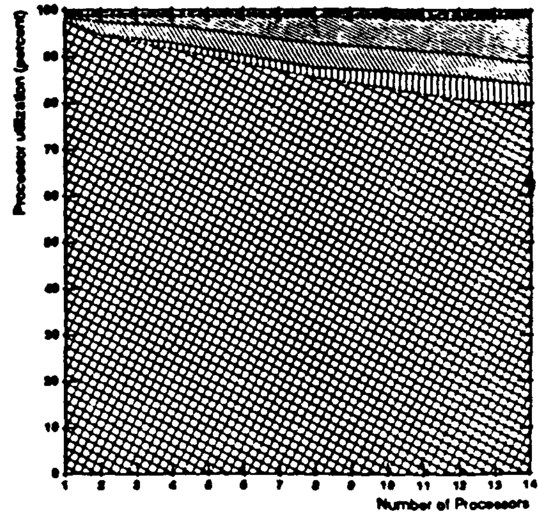


Figure 14: Processor utilization for 8-Q, Algorithm A3

however. Consider the following cross product:

$$(\{A\}, \{B\}, \{C\}, \{D\}) \times (\{D\})$$

If we simply perform the cross product, we get the list $(\{D\})$. If we split the list $(\{A\}, \{B\}, \{C\}, \{D\})$ into two parts and perform seperate cross products, however, we get $(\{A, D\}, \{B, D\})$ from the first part and $(\{D\})$ from the second. Now, instead of propagating a single list of length one to the successors of the consequent, a list of length two and a list of length one are propagated.

We can see this happening in the BUG trace file. The largest Update task in the trace arises from a justification:

$$x_1, x_2, x_3, x_4, x_5 \Rightarrow n$$

The Update request comes from x_1 with a list of 8 new environments. Nodes x_2, x_3, and x_4 all have 8 environments in their labels, and node x_5 has 1 environment in its label. The resulting cross product environment list could contain as many as 8^4 environments. It actually produces only 293 environments, because of nogood subsumption and list minimality. If the incoming new environment list of 8 environments is split into two environment lists of 4 environments each, one resulting cross products has 367 environments and the other has 55. The net effect of splitting this single Update request into two smaller requests is substantial. The sequential execution time for BUG increases from around 82.31 seconds to 119.96 seconds, an increase of 46%. While we could have all processors working on a single Update synchronize and combine their results before propagating them on, the added synchronization combined with the fact that the pieces of a split Update are not necessarily smaller than the whole Update combine to make such an action unwise.

Due to the above reasons, our initial efforts to go to a smaller grain have not resulted in much success. In order to get significantly more speedup from some ATMS instances, we need to find a natural task grain which is smaller than that of an Update request. Unfortunately, no obvious alternative presents itself.

6 Conclusions

In this paper, we have presented the details of implementing both a serial and a parallel ATMS. The results we obtained from executing the parallel implementation on an Encore MultiMax allow us to draw a number of conclusions about executing the ATMS in parallel.

- The traces we examined seemed to present abundant opportunities for parallelism. They consisted of thousands of relatively independent tasks, capable of being distributed among a number of processors. However, this apparent abundance of parallelism proved to be somewhat elusive to exploit.
- The obvious source of parallelism in the ATMS, the thousands of justifications, generated grains which varied enormously in size. In one trace, for example, a single justification accounted for 43% of the total runtime, making effective parallel distribution of grains impossible. In order to make grain sizes more uniform, we were forced to decrease grain size by treating a single justification update as a task. We also introduced the notion of modal access to the environment database in order to alleviate the sequentiality constraints imposed by the global consistentency requirements. Modal access requires that, at any one time, environments can be added in parallel or removed in parallel, but not both.
- With these modifications, we were able to obtain speedups of between 4.4 and 6.7 using 14 processors for the three trace files which we examined. Further speedups were limited by a number of factors, including still too large of a variation in task grain size, processor contention for numerous mutual-exclusion locks, and hardware contention issues.
- We further note that we examined the best case scenario, where Node-Query commands are infrequent and global synchronization is necessary only at the completion of the entire trace. While it's not clear what the average case would be, it would almost certainly present fewer opportunities for parallelism.

- By combining a highly efficient C-based implementation with a modest degree of parallelism, we have created an ATMS implementation which is significantly faster than currently available LISP-based implementations.

- We believe that in order to acheive near-linear speedups, parallelism in the ATMS must be exploited at a finer grain than that used in the three algorithms presented here.

While in this paper we have explored how ATMS parallelism can be exploited on a shared-memory multiprocessor, a related question is how it can be exploited on other types of parallel machine architectures. Michael Dixon and Johan deKleer [3] have studied the implementation of the ATMS on the Connection Machine, a massively parallel processor with between 16K and 64K processors [7]. Their implementation has shown promise in the tests which they have tried, but it remains to be seen whether it will offer a dramatic speed advantage for a wide range of problem solver domains.

In the future, we plan to investigate the tradeoffs between using a shared-memory architecture versus a message passing or Connection Machine architecture for exploiting parallelism in the ATMS. We plan to investigate how the grain size can be reduced without introducing an enormous amount of extra work. We also hope to integrate our parallel ATMS with LISP-based problem solvers, allowing the exchange of commands and data through inter-process communication.

7 Acknowledgements

We would like to thank Johan deKleer and Ken Forbus for providing us with problem solver trace files. We would like to thank Hiroshi Okuno for his assistance in the initial stages of this research. This research is supported by DARPA contract N00014-87-K-0828. Edward Rothberg is also supported by an Office of Naval Research graduate fellowship. Anoop Gupta is also supported by a faculty award from Digital Equipment Corporation.

References

[1] deKleer. J., "An Assumption-based Truth Maintenance System", *Artificial Intelligence*. **28**. 1986

[2] deKleer. J., "Problem Solving with the ATMS", *Artificial Intelligence*, **28**, 1986.

[3] Dixon. M. and deKleer. J., "Massively Parallel Assumption-based Truth Maintenance", *Proceedings of the AAAI*, 1988.

[4] Filman. R.E. "Reasoning With Worlds and Truth Maintenance in a Knowledge Based System". *Communications of the ACM*, **31**, 1988.

[5] Forbus. K., "The Qualitative Process Engine". University of Illinois Technical Report No. UIUCDCS-R-86-1288. December, 1986.

[6] Gupta. A., Forgy. C., Kalp. D., Newell, A. and Tambe. M., "Results of Parallel Implementation of OPS5", *Proceedings of the ICCP*. 1988.

[7] Hillis. D., *The Connection Machine*, MIT Press, Cambridge. Massachusetts. 1985.

[8] Okuno. H., "An Efficient Parallel Execution of the ATMS", to appear.

Shared Memory vs. Message Passing Architectures: An Application Based Study

Margaret Martonosi and Anoop Gupta
(Draft: Nov 9, 1988, 7:30 pm)
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

The diminishing differences between the hardware structure of shared memory and message passing parallel computers mandate a new evaluation of the tradeoffs these architectures impose on the development and performance of applications. In a message passing computer, some message traffic is used to perform interprocessor updates which maintain consistency between the various processors' data. We consider this traffic to be analogous to global bus traffic needed in a shared memory computer for hardware cache consistency. Using LocusRoute, a global router for standard cells, as a case study, we investigate the level of traffic required to maintain consistency of data with each of the two architectures. By explicitly varying the frequency of interprocessor updates, the level of traffic in the message passing approach can be reduced to as little as 1% of the traffic in the shared memory approach while still obtaining solution quality within 10% of the quality given by the shared memory version. We show that exploiting locality, in the way wires to be routed are assigned to processors, can further lower this message traffic by as much as 67%. However, the degree to which locality can be exploited may be limited by the opposing requirement that the application be load balanced, as well as by limited locality in the data set.

1 Introduction

In recent years, there has been much debate about the relative merits of shared memory and message passing parallel architectures. The previously large distinctions between the two approaches, however, are now diminishing. The main drawbacks of the early message passing computers, such as the NCUBE/ten [11] were the high network latency and the large message reception overhead. These characteristics forced programmers to exploit only large grain parallelism. With the development of new message passing computers, such as the Ametek Series 2010 and the Message Driven Processor [3,9], things are rapidly changing. Using specialized routing chips and the technique of wormhole routing [6], the network latencies have been reduced by 2-3 orders of magnitude. Similarly, using dedicated hardware to copy messages to and from the network and using innovative memory mapping techniques [10,18], the message reception overhead has been cut down by 1-2 orders of magnitude. These reductions enable current machines to exploit parallelism at a fairly fine grain. Furthermore, it is now possible to approximate aspects of the shared-memory model, since sending a message to a remote processor requesting an update of some global data is no longer an unreasonably long operation.

On the other hand, we see that efforts to scale shared memory machines led them to resemble message passing computers in several ways. Traditional shared memory architectures used a shared global bus to memory [5] and could only accommodate a limited number of processors before the global bus connecting processors and memory became saturated. Because of the limited scalability of these architectures, designers of shared memory machines are now turning to architectures using processor clusters with directory-based cache coherence schemes between the clusters [19,1], hierarchical architectures with more than one level of shared buses such as the Encore UltraMax [13,4], or architectures with multistage processor interconnection networks

Submitted for Publication.

like the BBN Butterfly [12] and IBM RP3 [16]. In directory based approaches, consistency operations between clusters are performed on a point-to-point basis, with invalidations going only to the clusters that need them. Also, the latency of non-local communication can be as much as an order of magnitude larger than that of a local access in all of these machines. These two characteristics, point to point communication and high non-local reference latency, increase the cost of non-local traffic in a shared memory approach, and force the programmer of these shared memory machines to consider more carefully the effect of locality.

While the gap between the two architectures is narrowing, there are still fundamental differences between the two which force tradeoffs between the two architectures. The shared memory architecture considered in this paper has a single global address space with hardware to guarantee the consistency of data in the processor caches. In this case, cache coherence protocols enforce consistency of data [2]. In the message passing architectures, each processor has a separate address space, and exchanges of information occur by sending messages on the network. One type of message passed on the network is for updating distributed data held by all the processors to periodically make it consistent with the other processors' data. This message traffic for updates in the message passing case is analogous to the cache coherence traffic used in the shared memory case. In the message passing case, this traffic is explicitly controlled by the application programmer, who decides when and how to perform updates between processors. In the shared memory case, the traffic is implicitly controlled by the cache consistency hardware, which relieves the programmer from the process of maintaining data consistency.

In many cases, however, the level of consistency enforced by the shared memory computer may be more than is needed by a particular application. In such cases the message passing computer may be superior, because it allows the application programmer to control the degree of consistency explicitly. In this paper, we explore several such tradeoffs between shared-memory and message passing architectures using the LocusRoute [17] application as a case study. (LocusRoute is a commercial quality routing program for standard cells and is now being widely used at Stanford for parallel processing studies.) Our results comparing network traffic show that the message passing version of the program generates only 1% of the traffic that the shared memory version does, while the degradation in the quality of the routing is less than 10%. Another issue this paper addresses is the sensitivity of the network traffic to the exploitation of locality in the data set. In our study, we exploit locality by assigning wires which are physically close to each other in the circuit, to the same processor. Specifically, we show that message passing architectures can reduce their network traffic by more than 50% by exploiting locality, while also improving solution quality. The effect on shared memory architectures, while not so large, is also significant. Some other issues regarding exploiting of locality and execution time are also discussed.

The rest of the paper is structured as follows. The next section gives information about the LocusRoute application and the simulation tools used to collect data for the architectural comparison. Section 3 describes the changes made to the original shared memory LocusRoute to convert it to a message passing style. Section 4 presents our results on network traffic for the two architectures under varying assumptions, and Section 5 shows the reduction of network traffic made possible by exploiting locality. Section 6 presents conclusions based on this data, and suggests further areas to explore.

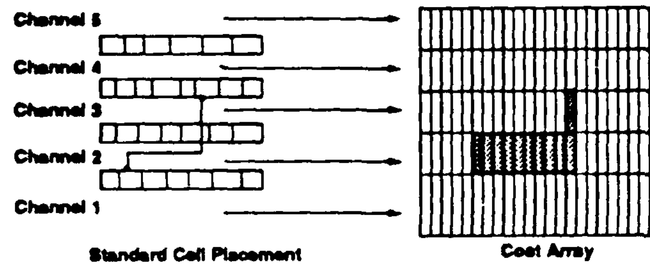


Figure 1: Standard cell placement and corresponding cost array.

2 Applications, Tools and Methodology

To understand the architectural comparisons being made. one must understand the application the data is based on. and the tools used to make the measurements. Our starting point was the version of LocusRoute written for a shared memory machine. This code was converted. as described in Section 3. to a message passing style. Because there was no message passing computer available to run the code. we used CBS. a simulator for parallel message passing machines. With CBS. detailed statistics on execution time and network behavior are readily available. To make network traffic comparisons between the message passing version and the shared memory version. another program. also described below. was written to estimate the amount of bus traffic required by the shared memory approach.

2.1 LocusRoute

LocusRoute [17] is an industrial quality router for VLSI standard cells developed by Jonathan Rose at Stanford University. LocusRoute routes the wires of a given standard cell placement. while attempting to minimize the overall circuit area. To do this. it maintains a global data structure known as the **Cost Array**. The vertical dimension of the array is the number of routing channels in the circuit. and the horizontal dimension of the array is the number of routing grids. The **Cost Array** keeps a record of the number of wires running through each sector of the circuit. Each wire is routed along the path with the minimal sum of the cost array entries. Figure 1 shows a standard cell circuit and one of its wires. with the corresponding **Cost Array**. The highlighted portions of the cost array will be incremented if this route is chosen.

In addition to producing the routed circuit, LocusRoute also computes a measure of the solution's quality. Quality, also referred to in this paper as the *circuit height*. is computed as follows. For each channel. the number of wires using the channel will vary across the width of the circuit. The number of routing tracks required by the channel is the maximum number of wires running through the channel at any point. The *circuit height* is the total number of routing tracks required for all channels.

The **Cost Array** is the central data structure for the LocusRoute application. and it accounts for almost all of the shared data references made by LocusRoute. Therefore. studying the reference patterns to the cost array will provide an excellent approximation to the application's memory reference behavior as a whole. Examining the references to the cost array for one wire. we see that LocusRoute starts with a series of reads to explore possible routes for the wire.

LocusRoute reads every location of the cost array along the paths being considered. This is followed by a smaller stream of writes, as the cost array is updated along the final path of the wire. This basic sequence of reads and writes occurs for each wire, with several processors routing wires in parallel. Performing several iterations of routing improves the final solution quality, but, before rerouting a wire for an iteration after the first one, the processor must "rip up" the old routing of the wire by decrementing the cost array locations in its path. These rip up operations are the second type of writes performed on the cost array.

Consistency of the cost array is an important issue in this paper, and one with serious implications on the amount of traffic, so it is important to understand how, and to what extent, consistency is maintained in the shared memory version of LocusRoute. To avoid the performance bottleneck a lock would impose, accesses to the cost array are not locked. This implies that simultaneous operations on the same element of the cost array may result in one of the operations being lost. As previously stated, LocusRoute is a optimization problem, and can tolerate a certain amount of inconsistency. With the number of processors the shared memory version currently uses (up to 16), the probability of simulataneous writes is very low, and experiments indicate that the quality is not degraded. Except for this, consistency in the cost array is maintained at the hardware level, by the cache coherency hardware.

When running the experiments, two benchmark circuits were used. The first circuit, bnrE has 420 wires, a size of 10 channels by 341 routing grids, and represents an actual standard cell circuit developed at Bell-Northern Research Ltd. The second circuit, MDC, has 573 wires with a size of 12 channels by 386 routing grids, and was designed at the University of Toronto Microelectronic Development Centre.

2.2 CBS: A Message Passing Architecture Simulator

Execution of LocusRoute on a message passing computer was simulated using a program called CBS. CBS [15] is a C++ program written by Andreas Nowatzyk at Carnegie Mellon University which simulates the behavior of a k-ary n-dimensional hypercube machine (with a total of k^n processors). For the experiments described here, CBS simulated a machine with deterministic wormhole routing using the E-cube routing algorithm [14,21], and with the dimension n, always equal to two (mesh interconnection). The use of wormhole routing minimizes the effect of the distance between destination and source, making the assignment of processes to processing elements less critical. Research by Dally [7,8] indicates that low-dimensional networks have greater channel bandwidth, and better hot-spot throughput than do high-dimensional networks. These two features give the simulated machine low-latency, high-bandwidth communication performance which makes it competitive with the shared memory machine.

CBS uses a detailed simulation model to produce its network statistics. CBS simulates the behavior of the processor interconnection network at the level of individual flow control units (in this case, single bytes) flowing between processors. There are unidirectional channels connecting a processor to its North and East neighbors. This means that a packet must travel all the way around the network to talk to its West neighbor. The network performance is specified with two parameters: *t_delay* and *c_delay*. T_delay is the time required for 1 byte to travel one hop on the network, and c_delay, the time required for the entire packet to be copied down from the processor node to the message network, or up from the message network to the destination processor node. Assuming no delays due to contention, the total time required for a packet of length L to travel D hops on the network is $2c_delay + t_delay(D + L)$. To simulate the execution time of the node processors, a *delay* statement is provided which blocks the running processor for the number of time units specified. Timings obtained from the Encore microsecond clock

were used as arguments for the delay statement.

For the purpose of concreteness, we chose to set the performance parameters to model the behavior of the Ametek Series 2010 Message Passing Multicomputer [18,10]. A packet of length L travelling D hops on the Ametek requires $CopyTime + HopTime(2D + L)$. CopyTime is the time required to copy the message from the network to the node processor's address space, which depends on the message length. This can be performed at about 50MB/s. Assuming an average message length of 200 bytes, we chose CopyTime to equal 4000 ns, so c_delay was set to half of that, or 2000 ns. [1] HopTime for the Ametek is defined as the time it takes for one byte of a packet to advance one hop, assuming that the route has already been established. This is stated to be 50 ns. (Establishing the route is slower, so the head of a packet requires two HopTimes to advance one hop.) To make the Ametek packet latency equation conform to the CBS form, we factored out the 2 to get: $CopyTime + 2HopTime(D + L/2)$. Now we can set t_delay equal to $2HopTime$ or 100 ns. When the simulation is run, the number of bytes in a packet is always cut in half, so that the L/2 term in the Ametek equation matches the L term in the CBS equation. Also, since the Ametek 2010's processing elements are about five times faster than the Multimax's processing elements, we divide the times obtained on the Encore processing elements by a factor of five before using them as arguments to the delay statement.

2.3 Shared Memory Traffic Evaluation

While for the message passing implementation, the network traffic is directly given by CBS, there is no simple way of estimating traffic for the shared memory implementation. In order to make comparisons of the traffic required for message passing and shared memory approaches, we need a method for measuring the traffic generated by LocusRoute on a shared memory machine. Although we have the capability to directly trace all memory references [20,22], these direct methods require a large amount of memory, which limits the portion of the program that can be traced to about 1 wire per processor. As will be described in Section 3, updates in the message passing approach can occur at time intervals greater than the total time being monitored by these detailed trace methods, making traffic comparison difficult. Instead we have chosen to modify the shared memory version of LocusRoute to record information about the memory reference stream over the whole execution time and use this information to estimate total traffic.

Before explaining the traffic estimating program, we will first explain in detail the type of machine to be simulated. This work considers a shared memory multiprocessor with a single global bus that all processors use to access memory. Each processor has a private cache memory, and consistency is maintained by dedicated cache coherence hardware using a Write Back with Invalidate scheme. The traffic being measured in the shared memory version is the traffic on the shared global bus. [2]

With the above machine model in mind, we now describe a method for estimating the bus traffic. The uniprocessor version of LocusRoute is modified to record memory reference information. Data is printed to a trace file whenever one of four types of events occurs. The four event types are described below:

1. **Wire event:** Whenever routing of a new wire is begun, the time and wire name are recorded as a *wire event* in the trace file.

[1] C_delay is a parameter set at the time the simulator is compiled. Therefore, one cannot compute c_delay dynamically.

[2] Our traffic estimating program can simulate non-bus-oriented architectures as well, but since we are only considering 16 processing elements, we present data only for the bus-based scheme.

2. **Iteration event:** When a new iteration is begun, the time at which it was begun is recorded as an *iteration event* in the trace file. These first two events are needed by the traffic evaluator for interleaving the execution among several processors.

3. **Read event:** LocusRoute is structured so that a processor reads data from the cost array when it evaluates possible routes for a wire segment. For each route considered, a processor reads all the locations of the cost array along the path of the route. At the beginning of each of these read sequences, a *read event* is recorded in the trace file, with the time, and the locations affected by the reads.

4. **Write event:** A processor writes data at two times: (i) at the end of exploring alternative routes, and (ii) when it does a rip-up before starting to explore routes. Both of these are recorded as write events in the trace file along with the time at which they are begun, and the locations affected by the writes.

All events are assumed to be atomic: one time is recorded in the trace file at the beginning of the read or write sequence, and all reads or writes associated with that event are assumed to occur at that time.

The trace file described above records the relevant memory access information about a *uniprocessor* run of LocusRoute. The events recorded in the trace file give enough information to interleave execution among more processors, so that *multiprocessor* traffic data can be estimated using the steps described here. First, the simulator decides on an assignment of wires to processors, using one of the heuristics from Section 3. At this point, each event in the trace is associated with a certain wire, and each wire has now been assigned to a processor, so all the events are now associated with the processor executing them. Events for each processor can be interleaved using the times recorded in the trace file. To simulate the traffic, events are pulled off the interleaved event queue and handled in order. The cache coherence protocol implemented is Write Back with Invalidate Scheme [2]. The first write to any cache line results in a bus operation which causes all other caches to invalidate that line if it is present. Subsequent reads or writes by that processor do not result in any bus traffic. A read or write by another processor to that cache line causes that processor to become the owner, and forces the previous owner to invalidate the line from its cache.

Like any such measurement system, ours has its inaccuracies, which we will list here. First, the system simulates infinite caches. Lines are only written back to memory for coherency reasons, never simply for replacement. Because the data structure we are studying is smaller than most multiprocessor caches (about 8000 bytes), this is not a major flaw. Second, we assume that read and write *events*, which are conglomerations of several read or writes, occur atomically. In actual execution, these operations occur as sequences of operations in tight (single instruction) loops. The atomic assumption only leads to inaccuracy when a read event and a write event, or two write events, that should be occurring simultaneously become serialized by the simulator. If these events were occurring in the same area of the cost array, then their simultaneous execution would lead to multiple invalidations and refetches. If they are serialized, at most one invalidation and refetch will occur. This will cause the simulator to slightly underestimate the total traffic. This is not a major effect, because write operations are relatively infrequent, so simultaneous reads and writes to the same cache line are highly improbable. Both of the inaccuracies tend to slightly underestimate the total traffic so that the numbers given in Section 5 may be considered lower bounds on the actual traffic.

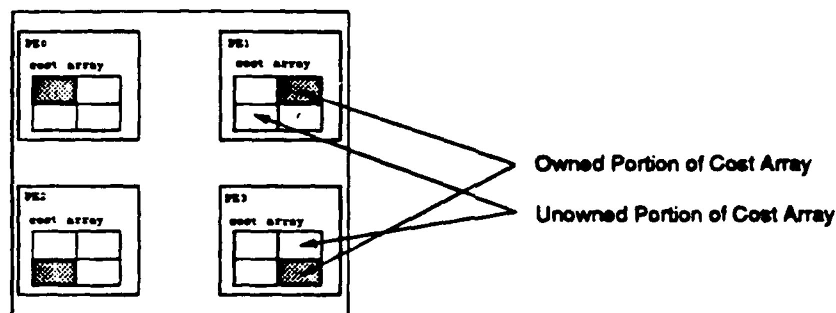


Figure 2: Division of the cost array among processors.

3 Implementing LocusRoute for a Message Passing Machine

Finally, before we go on to discuss the results, we need to specify how we mapped LocusRoute to a message passing architecture. Because the message passing machine has distributed memory, implementing LocusRoute required changes in the distribution and updating of information between processors. To reduce message traffic on the network, a static method of assigning wires to processors was used, rather than allowing processors to send requests out when they need a wire. These changes are described below.

3.1 Distribution of Data Structures

The most important data structure in the program, as previously stated, is the Cost Array. In the shared memory version, all processors have access to a single copy of the cost array. The message passing architecture forces the programmer to decide how the cost array should be handled in a machine with distributed memory. We chose to divide the cost array into sections, with each processor being the owner of one section. Each processor is, however, allowed to have a view of the whole cost array. The portion that is owned by that processor will be as consistent as possible, while the other portions of the cost array are less consistent, but still usable. Consider, for example, a four processor case. Figure 2 shows each processor's cost array, with the portion that it owns highlighted. Although, the unowned portions of each processor's cost array may not be accurate, the processor is still allowed to make use of them. Thus, if there are 4 processors, the bottom left processing element will own the bottom left fourth of the cost array, but it will also have a copy of the rest of the cost array which it can use. In all future discussion, the processor which owns a certain region of the cost array will be called the *owner processor* for that region, and the region itself will be called the *owned region*.

3.2 Maintaining Cost Array Consistency

No circuit is perfectly local, that is, wires assigned to one processing element will extend into regions owned by other processing elements. Consequently, using the scheme discussed above cost array updates between processors are needed. Many different methods of performing these updates are possible; one can experiment with the frequency of updates, as well as how the updates are initiated. The update frequency was allowed to vary, with updates occurring at time intervals on the order of the time it takes to route one wire. The decision of which update

strategy to use depends heavily on the underlying computer architecture. We have considered two main types of updates, and variations on these. The two types of updates considered are *sender initiated* updates, and *receiver initiated* updates, as well as a mixture of these two.

With sender initiated updates, the processor which determines that an update is necessary is the one to send out the data. With receiver initiated updates, the processor which determines that an update is necessary sends a request packet to another processor, and the destination processor then sends back the requested update data. The architectural dependencies in these schemes should be clear. For receiver initiated updates to be useful, the latency of the network as well as the message reception time, must be low, so that the requesting processor spends a minimal amount of time idle, waiting for the requested data. On the other hand, our results shown in Section 4 indicate that sender initiated updates tend to send out more bytes than receiver initiated, and therefore place a greater premium on high network bandwidth.

3.3 Wire Assignment

One advantage of the shared memory architecture is that the wires to be routed can be easily allocated to processors dynamically, using a distributed loop. [3] In the message passing architecture, dynamic wire allocation requires message transactions on the network. In our implementation, processors only retrieve messages queued for them at the end of each wire routed. Assuming that the processor receiving the task requests is also routing wires, the potential wait for a wire task is large, because in this case, a processor may have to wait for an entire wire to be routed before the wire assignment processor even receives the message. With this in mind, a static method of wire assignment was used. Because of the division of cost array into owned regions, the algorithm benefits from a method of wire assignment that attempts to assign wires to the processor that owns the region they run through. We use a very simple heuristic developed to achieve this goal. We assign wires to the owner processor of the lower leftmost pin of the wire.

To control the amount of locality exploited, a parameter *ThresholdCost* is provided. If a wire's "cost", a function of its length, is less than the threshold, it will be assigned using the heuristic described above. Otherwise, it is held in a pool of unassigned wires, and is assigned to a processor at the end of the wire assignment phase. The processor it is assigned to is the processor whose total wire cost is the current minimum. With this method, a high value of ThresholdCost results in a wire assignment that is based primarily on locality, while a low value of ThresholdCost results in a wire assignment that is based mostly on load balancing.

4 Traffic in Shared Memory and Message Passing Architectures

In this section, we present results on the traffic generated by shared memory and message passing architectures. We think this is a useful measure because in both architectures overly high network traffic results in a performance penalty. For example, since the message passing architecture forces the cost array to be distributed across the processors, periodic update messages are needed to keep the processors' views of the cost array consistent. There is computational overhead associated with sending and receiving these messages, so one would like to update as infrequently

[3] Associated with a distributed loop is a locked index variable. To get the next wire to be routed, a processor obtains the lock, reads and increments the index of the next wire to be routed, and releases the lock.

as possible. Similarly, in the shared memory architecture, hardware cache consistency protocols cause extra global bus traffic due to cache line invalidations and any subsequent refetches that may be needed. These operations cause the processor to stall, and also represent a performance overhead. Although the degree to which network traffic translates to performance overhead will be different for the two architectures, we contend that a comparison of the network traffic between the two architectures is itself useful.

This section will show that traffic for the shared memory architecture is a strong function of the cache line size, while traffic in the message passing architecture is explicitly controlled by the programmer. This explicit control allows the traffic to be reduced by more than two orders of magnitude compared to the shared memory traffic, and the program still gives comparable solution quality.

4.1 Traffic in the Shared Memory Architecture

Here we consider traffic in the shared memory approach. Traffic in the shared memory approach is made up of 3 parts. First, the processor's very first access to a location always results in a miss, and brings the line into the cache. Second, the first write to a clean location causes a word write on the shared bus. The other processors see this write and invalidate that cache line if it is in their cache. Third, once a line has been invalidated by a cache, it may need the line again. This leads to refetches of data from memory. Traffic in the shared memory architecture is, clearly, a function of the cache coherence protocol used, and the line size of the cache. For all the results given here, the coherence protocol used was a Write Back with Invalidate scheme [2]. The line size of the cache was allowed to vary.

Increases in the cache line size can have the effect of either increasing or decreasing traffic. There are two factors which will increase traffic with increasing line size. First, with an increased line size, data items that will never be used are more likely to be brought into the cache. This will increase the traffic on the bus. Second, increasing the line size means there will be more data in the cache (under the infinite cache assumption) and this means that processors are more likely to interfere with each other. With more data in the caches, processors are more likely to force invalidations in other caches. These invalidations, as well as the subsequent refetches, also cause the traffic to increase. On the other hand, it is possible for a longer cache line to cause a traffic decrease as well. If there are several shared data items stored relatively close to each other, then a single invalidation of a long cache line could cause them to all be invalidated in one operation. This can save bytes over the case of several individual invalidations, and cause the traffic to decrease. Because this last situation happens infrequently, its effect is minor compared to the first two. Thus, we expect that increasing the cache line size will lead to an increase in the number of bytes transferred.

Table 1 shows the shared memory bus traffic as a function of the cache line size. As predicted, the data clearly shows that the traffic increases significantly as the line size increases. For example, in the MDC circuit, a cache line size of 4 bytes causes the total traffic to be 932,976 bytes while a larger cache line size of 32 bytes causes the traffic to increase sharply to 5,840,280, more than five times as much.

Solution quality and execution time are not available for the wire assignments shown in Table 1 because the actual shared memory implementation uses a dynamic wire assignment. The simulator does not actually route wires, so it cannot output solution quality or execution time values. However, for comparison with the message passing figures presented later in the paper, the shared memory version of LocusRoute running on an Encore Multimax with 16

Table 1: Traffic as a function of cache line size in shared memory version.

Circuit	Cache Line Size	Bytes Transferred
bnrE	4	769.788
	8	1.306.152
	16	2.429.764
	32	4.712.540
MDC	4	932.976
	8	1.596.940
	16	3.043.941
	32	5.840.280

processors can route the bnrE circuit in a time of 5.59 seconds with a height of 136. For MDC. the solution quality is 144 in a time of 5.78 seconds.

4.2 Traffic in the Message Passing Architecture

Now that we have presented data from the shared memory architecture, we move on to the data from the message passing architecture. comparing the two as we go. Traffic in the message passing approach is determined by the programmer. subject to the constraint of acceptable solution quality. The programmer controls the size of messages, as well as their frequency. The results given in Table 2 show the traffic required by the message passing approach with varying update strategies. All the results given are for 16 processors. Because of the explicit tradeoffs in the message passing approach between network traffic. solution quality. and execution time. one cannot discuss the amount of network traffic required without also discussing the update strategies used and the resulting solution quality and execution time. Table 2 shows data for three different update strategies (discussed in Section 3.2). Recall that, in a sender initiated strategy. it is the sender of the update that determines when the update should be sent. In the receiver initiated strategy. the processor wishing to receive an update sends a request to another processor, who then returns the data. The third strategy is a mixture of the other two. In this third mixed approach, sender initiated updates occur with the same frequency as in the purely sender initiated strategy, and receiver initiated updates occur with the same frequency as in the purely receiver initiated approach.

4.2.1 Network Traffic

There are two points to be made about the message passing network traffic. First, there is a large difference between bytes transferred for sender initiated and receiver initiated. Intuitively, one would expect the receiver initiated approach to be more efficient in terms of network traffic. because data is only sent when it is specifically requested. In contrast, in the sender initiated approach, data is sent periodically, regardless of whether the destination processor is routing wires in that area and needs that data or not. Consequently, one would expect a larger number of bytes to be necessary to get the same quality as receiver initiated. The data. shown in Table 2 bears out this intuition. Receiver initiated transfers use anywhere from 44% to 71% fewer bytes than sender initiated to produce similar quality results. Because sending and receiving

Table 2: Traffic in the message passing version.

Circuit	Update method	Circuit Height	Execution Time	Bytes Transferred
bnrE	Sender Initiated	145	1.603	156468
bnrE	Receiver Initiated	150	1.210	87572
bnrE	Mixed	146	1.519	245270
MDC	Sender Initiated	150	2.171	236304
MDC	Receiver Initiated	156	1.635	85646
MDC	Mixed	153	2.208	324914

messages has a computational overhead. the savings in network traffic translate to time savings as well. For all the trials given in Table 2, the receiver initiated method is about 20% faster than the corresponding run using the sender initiated method.

The second point to note about the network traffic on the message passing architecture is that. even with the less efficient sender initiated method. the message passing network traffic is more than an order of magnitude less than that for shared memory. The huge difference may. at first glance, be surprising. but it can be explained by two factors. First. the updates being performed in the message passing version occur. at most. once per wire. In the data shown in Table 2. they occur approximately every two wires. Second. these updates can be thought of as constituting a very loose form of coherence protocol. There are several differences between this protocol and the strict one implemented on the shared memory system. Because updates occur no more frequently than once per wire. the write performed at the wire rip up stage is handled at the same time as the write performed at the wire routing stage. Because much of the wire's path will remain the same after rerouting. these two writes will often cancel each other. and many of the locations will not need to be updated at all. [4] This removes many of the write operations. a significant accomplishment since writes are the cause for over 80% of the bytes transferred in the shared memory version.

4.2.2 Solution Quality

A discussion of network traffic in the message passing approach is incomplete without also discussing the solution quality and execution time required. The first point we note is that the solution quality. that is the height of the circuit, has degraded slightly from the quality given by the shared memory version, but is still acceptable. Recall that the solution quality for the shared memory approach was 136 for bnrE and 144 for MDC.

Some quality degradation can be expected in the message passing approach, because less information is available to each processor as it is routing. For example. the cache coherence hardware on the shared memory machine guarantees a perfectly consistent view for all processors at all times. The only inconsistency in the shared memory approach comes from not locking the cost array. and as previously explained, this has no noticeable effect on the quality. By contrast. in the message passing approach. updates occur only after the processors have each routed

[4] In the message passing implementation, a *delta array* is maintained which records changes made to the cost array. If no changes are made to a location. or the changes cancel each other. updates for that location need not be sent.

one or more wires. This leads to a much larger level of inconsistency in the processors' cost arrays. In the worst case, the solution quality in the message passing approach has degraded by 10% from the shared memory solution quality. LocusRoute is intended to be used with a standard cell placement program, so that once the placement program has decided on a placement, LocusRoute performs the routing for that placement. LocusRoute returns the circuit height as a measure of that particular *placement's* quality. Thus, in this situation, slight declines in the quality of the routing can be tolerated. However, if LocusRoute is used to produce the final routing for a circuit that is to be mass produced, this increased area could become quite significant. For these cases, this characteristic of the message passing implementation of LocusRoute could make it undesirable.

Section 5 will discuss how the quality of the solution can be improved somewhat by exploiting locality in the wires. However, another obvious way to improve the solution quality is by increasing the frequency of updates. The best combination of execution time and solution quality obtained for bnrE was a solution quality of 136 with an execution time of 1.7 seconds and 843,542 bytes transferred. Note that the quality of this run equals the quality given by the shared memory version. The bytes transferred, while much larger than any of those in Table 2 is still about a factor of two less than those measured for the shared memory version. Results such as this indicate the robustness of the LocusRoute algorithm to inconsistencies in the cost array. For LocusRoute, and other applications like it, hardware cache consistency seems to impose a large cost on the execution of the program, without giving compensating benefits.

4.2.3 Execution Time

Now, we turn to the execution time. Execution time for the message passing version of LocusRoute depends heavily on the wire assignment strategy *for two reasons*. If the wires are not assigned in a way that will balance the load on the processors, execution time will suffer greatly. This is discussed in Section 5.3.2. On the other hand, if the wire assignment does not exploit locality well, more update packets will need to be sent to get the same quality, and the extra processing time spent sending and receiving messages will show up in the final execution time of the run. In general, the execution times given in Table 2 are much faster than those for the shared memory runs. However, recall that CBS is simulating processors which are five times faster than the processors used when timing the shared memory version. A rough comparison can be obtained by multiplying the execution times of the message passing version by five. The fastest execution time obtained overall is 0.97 seconds for the receiver initiated scheme. Multiplying by five, the execution time becomes 4.85 seconds which is still 13% faster than the execution time from the shared memory case. The quality obtained in the message passing run being considered was only 7% worse than the shared memory quality. Note that simple multiplication by a factor of five when comparing the execution times favors the shared memory architecture. This is because if the processors in the shared memory machine really were five times faster, there would be more contention on the bus, and the overall performance would not improve by a factor of five.

5 Effect of Locality

The previous section compared the network traffic required by LocusRoute in the shared memory and message passing architectures. Here we examine the effectiveness of exploiting locality to reduce this traffic in both architectures. Locality, here, is a measure of how often a processor is

routing wires within its owned region or regions close by. (A quantitative measure is described in Section 7.3.1.) Both architectures benefit differently from exploiting locality. Message passing architectures benefit from locality because the need for message traffic to produce a certain level of solution quality is reduced. This is because improving the locality of the wires routed by each processor means that each processor will have a better view of the part of the cost array it is routing in, and fewer updates will be needed. Shared memory architectures benefit from locality through better cache behavior. Specifically, shared memory architectures benefit because of two factors—better spatial locality, and less interference between processors causing cache coherence traffic.

In the past, locality has not played a major part in the design of shared memory parallel programs. In a traditional global bus shared memory computer, all memory is equally accessible to all processors. However, as hierarchical approaches are used to scale shared memory multicomputers, this is no longer true. The current trend towards hierarchical shared memory machines, in which a local reference can be more than an order of magnitude faster than a non-local reference implies that locality must become an important part of future program design. In this section we present data that indicates that implementations taking advantage of locality can reduce the total network traffic by as much as 67%.

5.1 Effect of Locality in the Shared Memory Architecture

In this subsection, we study the effects of locality on the interconnection network traffic generated by a shared-memory architecture. Table 3 shows the amount of network traffic generated as a function of differing amounts of locality exploited in the application. The extreme non-local case is taken to be the round robin wire assignment to processors, and the extreme local case (ThresholdCost = infinity) is taken to be the one where each wire is assigned to the processor whose owned region contains its lower left pin. We also consider two cases with intermediate locality.

Table 3: Effect of locality in shared memory version.

Circuit	Allocation strategy	Total Wires	Wires Held for round robin asmt.	Bytes Transferred	Reduction from round robin (%)
bnrE	round robin	420	420	1,306,152	—
	ThresholdCost = 30		209	1,299,580	0.5
	ThresholdCost = 1000		25	1,275,492	2.3
	ThresholdCost = inf		0	1,219,576	6.6
MDC	round robin	573	573	1,596,940	—
	ThresholdCost = 30		263	1,608,520	-0.7
	ThresholdCost = 1000		38	1,593,104	0.2
	ThresholdCost = inf		0	1,516,468	5.0

For bnrE with 8 byte cache lines, total global bus traffic can be reduced 6.6% by taking advantage of locality in the assignment of wires. It is clear that locality is not producing the significant benefit we had expected. The reason for this is that cache lines contain too many cost array entries. Since each cost array entry is one byte, a cache line holds eight cost array

entries. This means that for each byte accessed, seven of its neighbors are brought in as well. The increase in traffic brought about by interference between the caches maintaining coherence is almost as big as the decrease in traffic due to locality. To check this theory, we increase the size of the cost array entries to 4 byte integers, so an 8 byte cache line will hold only two of them. In this case, the total traffic for a round robin wire assignment is higher— up to 1,799,984 bytes, but the percent gain from exploiting locality is higher as well. For 4 byte array entries with an 8 byte cache line, changing to the infinite ThresholdCost wire assignment reduces the traffic by 15.6%. The reason for this bigger reduction is the following. With a cache line that holds more array entries, the probability of interference between processors causing invalidations is higher. Intuitively with these 1 byte cache entries, it is harder for a processor to be active in only a small section of the cost array, because every time an array entry is accessed, eight entries are fetched into the cache. If each processor has many "extra" entries in its cache, the cache coherence traffic will reduce the benefit possible from exploiting locality. Obviously, we are not concluding that all variables be made as large as possible, so that they interfere less with each other. Rather, the conclusion is that care must be used when allocating memory for write-shared data items. The notion, stemming from uniprocessor caches, that dense data will display better locality, and therefore better caching behavior, is no longer true when speaking of multiprocessor cache coherent systems. In multiprocessor cache coherent systems, the benefits of data density must be traded off with the penalty of increased coherency traffic.

5.2 Effect of Locality in the Message Passing Approach

Having examined the traffic in the shared memory case, we move to the message passing case, where the effect of locality is much more significant. Data in Table 4 shows the effect of various wire assignment strategies on the quality of the routed circuit, the execution time, and the number of bytes transferred.

One can see that in general, wire assignments which do not take advantage of locality, such as round robin, result in poorer solution quality than those that do, such as assignments made with infinite ThresholdCost. The average quality improvement due to locality over the cases shown in Table 4 is about 4%. While small, this improvement is quite significant, because the results given are all quite close to optimal anyway, so even a small percentage improvement is difficult to achieve. This data indicates that it is best to have a single processor route the wires in one area, because that processor will have more accurate information about the cost array in that area.

Having seen that exploiting locality improves the solution quality, the next question is, what is the effect of locality on the number of bytes transferred? This depends heavily on the type of update strategy used. In the sender initiated scheme, updates are sent out for any owned region in the sender's array that has changed, so the only reduction in traffic will be due to changes being made in fewer and smaller regions of the cost array. The change in bytes transferred for sender initiated updates from a round robin assignment to a local assignment with infinite ThresholdCost is 11.4%. The receiver initiated scheme will be more sensitive to locality, because in this strategy, low locality results in frequent interprocessor data requests. A processor only requests an update if it has routed in a certain owned region a specific number of times. If, by exploiting locality, we reduce the frequency with which update requests need to be made, we can dramatically reduce the message traffic. The data bears out this prediction, with a traffic reduction of more than a factor of two in both circuits, when going from a round robin assignment policy to a local one. Because the mixed strategy is made up partly of receiver initiated requests, which are highly sensitive to locality and partly of sender initiated requests,

Table 4: Effect of locality in the message passing version.

Circuit	Update method	Wire Assignment	Circuit Height	Exec. Time	Bytes Transferred	Reduc. from rnd-rbn (%)
bnrE	Sender Initiated	round robin	145	1.603	156468	—
		Thr_Cost = 30	138	1.467	149562	4.4
		Thr_Cost = 1000	139	1.647	141206	9.8
		Thr_Cost = inf	135	5.073	138636	11.4
bnrE	Receiver Initiated	round robin	150	1.210	87572	—
		Thr_Cost = 30	145	0.970	76334	12.8
		Thr_Cost = 1000	139	1.217	51582	41.1
		Thr_Cost = inf	140	4.043	39858	54.5
bnrE	Mixed	round robin	146	1.519	245270	—
		Thr_Cost = 30	141	1.411	218568	10.9
		Thr_Cost = 1000	142	1.601	179372	26.9
		Thr_Cost = inf	140	4.864	169992	30.7
MDC	Sender Initiated	round robin	150	2.171	236304	—
		Thr_Cost = 30	149	1.789	231310	2.1
		Thr_Cost = 1000	147	1.909	225912	4.4
		Thr_Cost = inf	148	5.323	223004	5.6
MDC	Receiver Initiated	round robin	156	1.635	85646	—
		Thr_Cost = 30	155	1.203	70236	18.0
		Thr_Cost = 1000	153	1.192	50308	41.3
		Thr_Cost = inf	152	4.479	28132	67.2
MDC	Mixed	round robin	153	2.208	324914	—
		Thr_Cost = 30	158	1.707	291428	10.3
		Thr_Cost = 1000	150	1.850	249578	23.2
		Thr_Cost = inf	150	5.429	236804	27.1

which are not very sensitive to locality, the reduction in network traffic is between the two other cases. The benefit gained by exploiting locality in the mixed sender/receiver case is larger than in sender initiated, and smaller than in receiver initiated.

Of course, locality also has an effect on the execution time of the application. As one improves the locality of the application, fewer update messages are needed, and the time the processors spend sending and receiving messages is reduced. This has a direct effect on the execution time. Unfortunately, there is another opposing effect as well. If all wires are assigned to processors strictly on the basis of locality, it is likely that the resulting wire assignment will not be load balanced. This effect will be discussed in Section 5.3.2.

5.3 Limitations on Exploiting Locality

The previous subsections have shown that exploiting locality to reduce execution time and increase quality clearly has some benefit. Unfortunately, there are several factors which limit the amount to be gained by taking advantage of locality in a problem. First, the standard cell

circuits themselves have only a limited amount of locality. If the wires to be routed are long enough to pass through the owned regions of several processors. there is an unavoidable amount of interprocessor communication that will take place to perform the necessary updates. Further. as the number of processors increases. the region size will decrease. and the locality will decrease as well. Second. fully exploiting the locality that does exist can often interfere with the load balancing of the processors. If there are many wires within a single processor's region. then considering locality alone. that processor should route them all. However, this may cause that processor to have a disproportionate amount of work. resulting in a load imbalance and poor performance. These two issues are treated in the sections below.

5.3.1 Limited Circuit Locality

To determine an upper bound on the degree to which LocusRoute can take advantage of locality. we developed a measure of the amount of locality in the standard cell circuits being used. Using this measure. we found that the degree to which locality can be exploited is, in part. limited by the circuits themselves. This section will first describe the method of measuring circuit locality. and then analyse the results for the two benchmark circuits.

The measure is computed in the following steps. First. wires are assigned to processors using one of the methods already described. Next. for each processor. the program computes the number of wire segments to be routed in each region of the cost array. and the distance in hops from the region where the segment lies to the processor performing the routing. Locality is considered to be the average distance between the routing processor and the processor that owns a region. weighted by the number of wire segments routed at this distance. Thus. a low number means good locality. For example. a locality measure of 0 indicates that all segments were routed by the region owner. giving perfect locality. Increases in the locality measure indicate that the average segment is being routed at a distance further from the owner. Note that when a wire assignment with infinite ThresholdCost is used for this calculation. one end of the wire is guaranteed to be in the owned region of the processor doing the routing. In this case. the degree of locality is mainly a measure of the length of the wire. compared to the size of the individual cost array regions because it measures how many hops the other end of the wire is from the lower left end. which is certainly in the region of the routing processor.

Computing the locality for the two test circuits. using several wire allocation strategies. gave the results shown in Table 5. These results are computed assuming 16 processors. The results indicate that the amount of locality to be exploited in these circuits is limited. For the bnrE circuit, using a round robin assignment, the average segment gets routed by a processor 1.96 hops away from the processor that owns the region the segment lies in. When the wire assignment method is changed to one with ThresholdCost equal to infinity, the average segment is routed by a processor only 1.24 hops away. As the number of processors is increased. the locality in the circuit will decrease because the size of each owned region, formed by splitting the cost array into equal chunks, will decrease. This limited locality in the circuits indicates that the message passing approach may require substantially more message traffic with very large numbers of processors than it currently does, and that the solution quality will be degraded as well.

5.3.2 Tradeoff between Locality and Load Balancing

The requirement that a parallel program be load balanced is also a limitation on the benefit possible from methods which exploit locality. To some extent. a circuit with good locality will

Table 5: Circuit locality.

Circuit	Allocation strategy	Total Wires	Wires Held for Round_robin asmt.	Measure of Locality
bnrE	round robin	420	420	1.96
	ThresholdCost = 30		209	1.77
	ThresholdCost = 1000		25	1.30
	ThresholdCost = inf		0	1.24
MDC	round robin	573	573	2.05
	ThresholdCost = 30		263	1.58
	ThresholdCost = 1000		38	1.04
	ThresholdCost = inf		0	0.99

require fewer updates, and therefore, less time to execute. However, the effect of a load imbalance can outweigh the subtle effect of the difference in update time. Therefore, in terms of execution time, the optimal point is neither a fully load balanced circuit, where the update time becomes significant, nor a fully local circuit, where the load imbalances become significant, but rather a point between the two.

The data from Table 4 shows this quite clearly, with the optimal execution time in almost every case being the ThresholdCost = 30 wire assignment. The most obvious example of the negative effect of locality on load balancing is exhibited in the move from the point with infinite ThresholdCost to the point with ThresholdCost equal to 1000. For example in the MDC circuit, this change in the way the last 38 wires are assigned gives as much as a 73% execution time reduction. However, wire assignments which use the most locality generally give the best solution quality. When processors route in localized regions, each has a fairly consistent view of the area it is routing in. Ultimately, this is a more effective way to produce good solution quality than nonlocalized routing with periodic updates.

6 Conclusions

The goals of this research were to re-evaluate the tradeoffs between shared memory and message passing architectures in light of the new features becoming prevalent in the two architectures. Specifically, we studied the level of traffic required by each approach to maintain consistency, and the effect of exploiting locality on this traffic. Although this study provides data from a single application, we feel that it is representative of a class of applications which do not require the strict consistency enforced by hardware cache coherence schemes.

We show that implementing the LocusRoute application on a message passing machine can result in a dramatic decrease in the amount of interconnection network traffic, with only a small negative effect on the solution quality. This is especially impressive because LocusRoute has been touted as an excellent application for a shared memory architecture. However, this dramatic improvement did have a cost. The explicit control afforded, and in fact required, by the message passing architecture requires significantly larger programming effort. The decisions of how to partition the cost array among the processors, how to initiate updates, how frequently updates should occur, and how to assign wires to processors, all involve complex tradeoffs and

much programming.

We further show that exploiting locality in the message passing case can have a positive effect on all three of the factors studied in this paper: solution quality, network traffic, and to a lesser extent, execution time. What, then, is the cost of exploiting it? The answer, once again, is that exploiting locality currently requires more programmer effort than using a simpler method which does not exploit it.

In this paper, we also studied the shared memory approach, examining the traffic necessary to maintain cache consistency. We found that the bus traffic in a shared memory architecture can be as much as two orders of magnitude larger than the network traffic in a shared memory approach. In an absolute sense, however, the amount of traffic is not excessive. For this reason, in applications where the improved solution quality given by the shared memory implementation is important, it appears to be the correct choice. Perhaps the most compelling benefit, however, of the shared memory architecture is the easy and natural programming environment it provides. The hardware cache coherence enforced enables programs to be developed and debugged more quickly and easily.

We also presented data indicating that traditional bus-based shared memory machines are not extremely sensitive to exploiting locality. In the LocusRoute application, this is in part due to the difficulty of singling out the array elements needed by each processor. In general, even in the most local wire assignment method, interference between processors is still a problem. Sharing of cost array information leads to a large number of invalidations and refetches. However, future machines relying on hierarchies to scale the total number of processors, are expected to be more sensitive to locality. As these architectures become available, more research will be needed to automatically detect and exploit locality in parallel programs.

7 Acknowledgements

We would like to thank Jonathan Rose for his patience in explaining the LocusRoute application to us, and Andreas Nowatzyk for his prompt, helpful replies to questions about CBS. Margaret Martonosi is supported by a fellowship from the National Science Foundation. Anoop Gupta is supported by DARPA contract N00014-87-K-0828 and by a faculty award from Digital Equipment Corporation.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz.
Scalable Directory Schemes for Cache Coherence.
In *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988.

[2] James Archibald and Jean-Loup Baer.
Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model.
ACM Transactions on Computer Systems, 4(4):273–298, November 1986.

[3] William C. Athas and Charles L. Seitz.
Multicomputers: Message-Passing Concurrent Computers.
IEEE Computer, 21(8):9–24, August 1988.

[4] David Cheriton, Anoop Gupta, Patrick Boyle, and Hendrik Goosen.
The VMP Multiprocessor: Initial Experience, Refinements, and Performance Evaluation.
In *Proc. Fifteenth Annual Symposium on Computer Architecture*, 1988.

[5] Encore Computer Corp.
Multimax Technical Summary.
1986.

[6] W. J. Dally and C. L. Seitz.
Deadlock-Free Message Routing in Multiprocessor Interconnection Networks.
IEEE Trans. Computers, 36(5):547–553, May 1987.

[7] William J. Dally.
A VLSI Architecture for Concurrent Data Structures.
Kluwer Publishers, 1987.

[8] William J. Dally.
Wire Efficient VLSI Multiprocessor Communication Networks.
In *Stanford Conference on Advanced Research in VLSI*, pages 391–415, 1987.

[9] William J. Dally, Linda Chao, et al.
Architecture of a Message-Driven Processor.
In *Proc. 14th Annual International Symposium on Computer Architecture*, June 1987.

[10] Ametek Computer Research Division.
Series 2010 System General Description Issue 3.
1988.

[11] J.P. Hayes et al.
A Microprocessor-based Hypercube Supercomputer.
IEEE Micro, 6(5):6–17, October 1986.

[12] BBN Laboratories Inc.
Butterfly Parallel Processor Overview.
1986.
BBN Report No. 6148.

[13] Andrew W. Wilson Jr.
Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors.
In *Proc. 14th Annual International Symposium on Computer Architecture*, pages 244–251, June 1987.

[14] C. R. Lang Jr.
The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture.

Dept. of Computer Science. California Institute of Technology. Technical Report 5014. May 1982.

[15] Andreas Nowatzyk.
CBS: A Message Passing Cube Simulator.
Unpublished report. 1988.

[16] G.F. Pfister, W.C. Brantley, et al.
The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture.
In *Proc. International Conference on Parallel Processing*. 1985.

[17] Jonathan Rose.
LocusRoute: A Parallel Global Router for Standard Cells.
In *Design Automation Conference*, pages 189–195. June 1988.

[18] Charles L. Seitz, William C. Athas, et al.
The Architecture and Programming of the Ametek Series 2010 Multicomputer.
In *Hypercube Concurrent Computers and Applications*. ??? 1988.

[19] Richard Simoni.
Implementing a Directory-Based Cache Consistency Protocol.
Unpublished report. July 1988.

[20] Richard L. Sites and Anant Agarwal.
Multiprocessor Cache Analysis using ATUM.
In *Proc. 15th Annual International Symposium on Computer Architecture*. June 1988.

[21] H. Sullivan and T.R. Bashkow.
A Large Scale Homogeneous Machine.
In *Proc. Fourth Annual Symposium on Computer Architecture*. pages 105–124. 1977.

[22] Wolf-Dietrich Weber and Anoop Gupta.
Analysis of cache invalidation patterns in multiprocessors.
To be published in Third Intl. Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.

Experiences Implementing a Parallel ATMS on a Shared-Memory Multiprocessor

Edward Rothberg and Anoop Gupta
Department of Computer Science
Stanford University
Stanford. CA 94305

November 8. 1988

Abstract

The Assumption-based Truth Maintenance System (ATMS) is an important tool in AI. So far its wider use has been limited due to the enormous computational resources which it requires. We investigate the possibility of speeding it up by using a modest number of processors in parallel. We begin with a highly efficient sequential version written in C and then extend this version to allow parallel execution on the Encore Multimax. a 16 node shared-memory multiprocessor. Our parallel implementation gives speedups of between 4.4 and 6.7 using 14 processors for the ATMS trace files which we examine. We describe our experiences in implementing this shared-memory parallel version of the ATMS. present detailed results of its execution. and discuss the factors which limit the available parallelism.

1 Introduction

The Assumption-based Truth Maintenance System (ATMS) is an important tool in AI. It makes the task of designing a problem solver much easier, removing the need for the problem solver to maintain information concerning derivations which it makes. Without an ATMS, the problem solver must implicitly record which of its assumptions it currently believes to be true and what these assumptions imply. When it wishes to change its assumption set. it must also recompute the set of items which are implied. With an ATMS, the problem solver explores the problem space, informing the ATMS of the assumptions it makes, the items which it wishes to reason about, and the derivations which it makes concerning these items. The ATMS aids in the process by keeping track of which items hold under any given assumption set, thus allowing the problem solver to freely change the set of assumptions which it currently believes. A number of problem solvers have been built which use the ATMS in a number of AI subfields. The ATMS provides a convenient level of abstraction, greatly simplifying the structure of the problem solver.

So far wider use of the ATMS has been limited due to the enormous computational resources which it requires. The ATMS is often the bottleneck in the problem solving process, often having greater computational requirements than the problem solver which it is collaborating with. We investigate the possiblity of speeding up the ATMS by using a modest number of processors in parallel. We begin with a hightly efficient C-based implemention of the ATMS based on the techniques described in [1]. Through a number of modifications to the basic sequential ATMS, we obtain moderate speedup on the three example problem solver trace files which we examine.

The paper is organized as follows. Section 2 presents background information about the ATMS and introduces related terminology. Section 3 presents details of an efficient sequential implementation of the ATMS. Section 4 presents the modifications to the sequential implementation which were necessary to allow parallel execution. Section 5 presents the results of executing the basic parallel implementation. We discuss the bottlenecks encountered and introduce a number of modifications to the basic algorithm to deal with these bottlenecks. Section 6 presents the conclusions which we arrive at based on the observed results.

2 The ATMS

The ATMS serves as a companion to a problem solver, acting as a sort of "truth database". The problem solver feeds beliefs, contradictions, and implications to the ATMS. The ATMS keeps track of what is true under what assumption sets and why. In this section we illustrate how the ATMS is used and introduce the terminology with a brief example. The example problem that we solve is the 3-queens problem. It consists of finding placements for three queens on a 3 by 3 chessboard such that no queen can capture any other.

Everything which the problem solver reasons about is assigned an ATMS *node*. In the 3-queens example we use 10 nodes, one for each of the 9 squares on the chessboard and one goal node to represent the solution. Each chessboard node represents the placement of a queen on the corresponding chessboard square. Some subset of the ATMS nodes are designated to be *assumptions*. These are nodes which are presumed to be true unless there is evidence to the contrary. In the example, the 9 nodes assigned to chessboard squares are the assumptions. We assume that a queen can be placed at each square of the board. Every important derivation made by the problem solver is recorded as a *justification*:

$$x_1, x_2, \ldots \Rightarrow n$$

where $x_1, x_2, \ldots$ are the antecedent nodes and n is the consequent node. In the example, the problem solver tells the ATMS that any set of three queens placed on the board constitutes a solution. Thus, the justifications take the form:

$$position_1, position_2, position_3 \Rightarrow goal_node$$

where $position_i$ is an assumption which corresponds to a queen being on a particular square on the chessboard. An ATMS *environment* is a set of assumptions. A node n is said to hold in environment E if n can be propositionally derived from the union of E with the current set of justifications. An environment is inconsistent (called *nogood*) if the distinguished node $\perp$ (i. e. false) holds in it. In the 3-queens example, we declare any set of assumptions in which the corresponding board positions contain a capturing pair to be nogood. The answer to the 3-queens problem is the set of all consistent environments in which the goal node holds.

In the ATMS, sets of environments play an important role in keeping track of the contexts under which a given node holds. They are used extremely frequently, and consequently we need a concise representation for a them. In our representation, we can take advantage of the fact that if a node holds under environment E, then it also holds under any superset of E. We can therefore represent a set of environments by its smallest members. We choose to represent a set S of environments as a list $(E_1, E_2, \ldots)$, which we call a ***minimal environment list***. It has the following properties:

- Every environment in the set S is a superset of some E_i.
- No E_i is a subset of any other.
- No E_i is nogood.

The distinction between sets of environments and sets of assumptions presents a possible source of confusion. For example, consider the environments {A, B} and {A, B, C}. Clearly {A, B, C} is a superset of {A, B}. Yet, the minimal environment list ({A, B, C}) represents a subset of the minimal environment list ({A, B}): the second contains environments which do not have assumption C in them, while the first does not. Please keep this potential source of confusion in mind when we discuss environment supersets and subsets in the remainder of this paper.

The problem solving process involves a dialogue between the problem solver and the ATMS, in which the ATMS receives a stream of requests to create new nodes, new assumptions, new justifications, and to provide information on the environments in which nodes hold. This information can be easily provided if the ATMS maintains with each node n a set of environments, in minimal environment list form, called its *label*. In addition to the minimal environment list properties, each node's label has the following two properties:

```
; an assumption for each board position
Create-Assumption "Queen at 1-1"
.
.
Create-Assumption "Queen at 3-3"

; and a node to represent a solution
Create-Node "Goal"

; all capturing pairs are inconsistent
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 1-2"
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 1-3"
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 2-1"
Justify-Node "FALSE" by "Queen at 1-1" "Queen at 2-2"
Justify-Node "FALSE" by "Queen at 1-2" "Queen at 1-3"
.
.
Justify-Node "FALSE" by "Queen at 3-2" "Queen at 3-3"

; the goal node is implied by any set of 3 assumptions
; (the problem solver discards those sets which will
; obviously not lead to a solution)
Justify-Node "Goal" by "Queen at 1-1" "Queen at 2-3" "Queen at 3-2"
Justify-Node "Goal" by "Queen at 1-3" "Queen at 2-1" "Queen at 3-2"
```

Figure 1: A Simple Formulation of the 3-Queens Problem

- Label soundness - Node n holds in every environment in the label set.
- Label completeness — Every environment E in which n holds is a member of the label.

2.1 The Interface Between the Problem Solver and the ATMS

The four basic operations which the ATMS makes available to the problem solver are:

- Create-Node n — create a new node.
- Create-Assumption n — create a new assumption.
- Justify-Node n by $x_1 x_2 \ldots$ — add a new justification.
- Node-Query n — request the current label of node n.

The problem solving process is a collaboration between the problem solver and the ATMS. In solving the 3-queens problem, the problem solver could indiscriminately feed all of the above mentioned justifications and nogoods to the ATMS and let the ATMS sort through them. Because the ATMS has no problem-specific knowledge, though, this results in a great deal of avoidable work being done. For example, the problem solver knows that a solution to the 3-queens problem will never have 2 queens in the same column or the same row, so it could simply reject any justifications which would obviously not result in a solution without passing them on the ATMS. To keep execution time to a minimum, the problem solver must be careful about the commands it passes to the ATMS. Figure 1 illustrates how the 3-queens problem might be formulated in the ATMS framework. DeKleer discusses in [2] how the problem solver can efficiently interact with the ATMS. In this paper, however, we do not address this issue. We simply deal with how the ATMS can efficiently handle the commands which the problem solver passes to it.

3 Sequential Implementation

We now examine how the sequential ATMS is actually implemented, with emphasis on those aspects of the implemention which are relevant to parallel execution. Since we will be computing the speedups of the parallel implementation based on the execution time of the sequential implementation, we must make sure that sequential version is as efficient as possible. While it may be possible to obtain substantial speedups as compared with an inefficient sequential implementation, such results give little information about how much parallelism is available in the problem. The only way to get a true measure of how much parallelism in the problem is actually being exploited is to begin with an efficient sequential implementation.

This section is organized as follows. Section 3.1 gives a general overview of an efficient ATMS implementation. Section 3.2 describes how set operations are done on minimal environment lists. Section 3.3 presents the data structures used to represent environments, nodes, assumptions, and justifications. Section 3.4 describes the three problem solver trace files which we will examine. We compare the performance of our sequential implementation with the performance of an existing ATMS implementation on these three trace files. Section 3.5 describes the environment database, the data structure which is used to keep track of those environments which are consistent and those which are nogood. Section 3.6 then gives a detailed description of the steps involved in computing an environment list cross product. Finally, Section 3.7 gives the details of how the union of two environments is computed.

3.1 Implementation Overview

The overall structure of the ATMS is as follows. The problem solver places Create-Node, Create-Assumption, Justify-Node, and Node-Query messages on a shared command queue The ATMS repeatedly removes available commands from the queue. Given a command, it performs the requested action, restores node label soundness and consistency for all nodes in the inference graph, and is then ready to perform the next command.

Of the four commands which the ATMS makes available to the problem solver, only Justify-Node consumes significant amounts of time. The Create-Node command takes very little time, since at the point at which the node is created it does not participate in any justifications. The Create-Assumption command also takes little time for the same reason. In our 3-queens example, the one Create-Node and 9 Create-Assumption commands simply require the ATMS to initialize the appropriate data structures. The Node-Query command is also computationally inexpensive because of the properties of label consistency, soundness, completeness, and minimality. In order to process a Node-Query command, the ATMS simply returns the current label of the appropriate node.

The ATMS spends the vast majority of its time in processing new justifications. A new justification can cause an enormous amount of label updating and environment propagation. When a new justification

$$x_1, x_2, \ldots \Rightarrow n$$

arrives at the ATMS, the labels of node n and any nodes which depends on node n may no longer be complete. Node n may now be derivable from a new set of assumptions not currently in node n's label because of the new justification. If this is the case, node n's label must be updated. If node n's label changes, then the label of every node which depends on node n may also change. Thus any change to node n's label must be propagated to every successor of node n.

A new justification can also cause new nogood environments to be discovered, potentially causing the node label of any node in the inference graph to become inconsistent. The simplest example of this would be a justification whose consequent is the false node. Conceivably, however, any justification whose consequent node n can derive the false node can cause new nogoods to be generated. In order to restore node consistency, environments which become nogood must be removed from all node labels.

In order to handle propagation of node labels, the ATMS maintains an Update request stack. Any time a node label is changed, Update requests are placed on the request stack, one for each justification which has the modified node as an antecedent. The first step in the processing of a new justification is to push an Update request onto the request stack. The ATMS continues popping Update requests off

of the Update stack, processing the requests, and potentially pushing more requests onto the stack until the stack is empty. This corresponds to a depth first propagation of labels.

A single Update request is processed as follows:

- The set of consistent environments which derive the consequent using the new justification and the new label environments is computed. This set is the intersection of the new label environments of the one antecedent with the labels of the other antecedents of the justification.
- If the consequent is the false node, then all of these environments are recorded as nogood and removed from all node labels.
- Otherwise, these environments are compared against the existing label of the consequent.
- If they are already there, then the propagation due to this justification is complete.
- Otherwise, the consequent node label is set equal to the union of the previous label and the new set of environments.
- The changes to the consequent label, i.e. the set of environments in the label which were not present in the previous label, are propagated to all nodes which depend on the consequent. This is accomplished by creating one Update request for each justification which has the current consequent as an antecedent. The Update requests are pushed onto the request stack.

3.2 Set Operations on Minimal Environment Lists

Adding a new justification requires a number of set operations on sets of environments, including set union and set intersection. The minimal environment list representation allows us to perform these operations quickly. Given two environment sets, S and T, represented as $(E_1, E_2, \ldots)$ and $(F_1, F_2, \ldots)$ respectively, we perform set operation on them as follows:

When we wish to add a new set of environments to the label of a node, we must take the set union of the existing label with the set of new environments. The set union of S and T, in minimal form is the concatenation of the minimal forms of S and T, with all supersets removed. In other words, each E_i is checked against each F_j for subsumption. If some F_j is a subset of E_i, then E_i is not included in the union. Similarly, if E_i subsumes some F_j, then F_j is also not included. All other E_i and F_j are included.

When we wish to compute the effect of a justification on its consequent node, we must find the set intersection of all of the labels of the antecedent nodes. The set intersection of S and T is somewhat more involved than the set union. If all supersets of E_i are in S and all supersets of F_j are in T, then all environments which are supersets of both E_i and F_j are in $S \cap T$. The set of all supersets of both E_i and F_j is the set of all supersets of the union of E_i and F_j (remember that environments are sets of assumptions). For example, the intersection of the supersets of $\{A, B\}$ with the supersets of $\{B, C\}$ is the supersets of $\{A, B, C\}$, which is the union of $\{A, B\}$ with $\{B, C\}$. Thus the intersection of S with T is the set of all supersets of the pairwise unions of E_i with F_j. Thus, in minimal environment list form, this is the cross product of the minimal environment list forms of S and T, again with all supersets removed.

3.3 Data Structures

The efficiency of the ATMS is highly dependent on the data structures and algorithms used in the implementation. A straightforward ATMS implementation can literally take days [8] to solve a problem which a more sophisticated implementation solves in a few minutes. We first present the major data structures used in our ATMS implementation. The data strucutures are simply laid out here with brief descriptions: the purpose of each individual field will be made clear in later sections.

The *environment* data structure has the following fields: (1) *Present:* a bit vector representing the set of assumptions present in the environment. (2) *Constituents:* a linked list of all assumptions present in the environment. (3) *Size:* the number of assumptions present. (4) *Contra:* a flag indicating whether the environment is consistent. (5) *Where:* a linked list of all nodes which contain this environment in

Table 1: Trace file statistics.

	QPE	BUG	8-Q
Nodes	988	1705	131
Assumptions	38	62	64
Justifications	2584	4165	1192
Run time - deKleer's on Explorer I	118	182	?
- ours on MultiMax	40.44	92.08	35.61
- ours on VAX 3200	15.45	34.21	13.81

Table 2: Runtime breakdown - Our Implementation on MultiMax

	QPE	BUG	8-Q
Run time (s)	40.44	92.08	35.61
Time spent on Justifys	32.34	79.00	32.21
Time spent on file access	6.46	10.68	2.42
All other time	1.64	2.42	0.98

their labels. (6) *Orthogonal:* a bit vector representing the set of assumptions which, if added to the environment, would result in a nogood environment.

The *node* data structure has the following fields: (1) *Label:* the node's label. (2) *Assumption:* a pointer to the node's assumption fields, if the node is an assumption. Empty otherwise. (3) *Justifications:* a list of the justifications in which the node is the consequent. (4) *Consequences:* a list of justifications in which the node is an antecedent.

The *assumption* data structure has the following fields, in addition to its node fields: (1) *Binary:* a bit vector representing the set of all binary nogoods this assumption participates in. If bit j is 1 in the Binary field of assumption i, then the environment $\{i, j\}$ is nogood. (2) *Nogoods:* a table of all minimal nogood environments in which the assumption belongs.

The *justification* data structure has the following fields: (1) *Antecedents:* a list of antecedent nodes. (2) *Consequent:* the consequent node.

3.4 The Trace Files

We present the results of executing three problem solver traces on our ATMS. These traces were given to us by Johan deKleer. They were generated by monitoring the interaction between an actual problem solver and an ATMS, and dumping the observed interaction into a trace file. The traces are:

- QPE, from a problem solver created by Ken Forbus [5] which solves Qualitative Physics problems.
- BUG, a trace which led to a bug in some ATMS implementation.
- 8-Q, from a problem solver which solves the 8-queens problem. This formulation of the N-Queens problem differs somewhat from the one described earlier in this paper.

Table 1 provides information on the three traces. It also provides the runtime for the three traces, both for the LISP-based ATMS implementation of deKleer [1] and for our C-based implementation. The time quoted for deKleer's ATMS is from execution on a Texas Instruments Explorer I lisp machine. The time quoted for our implementation is from execution on a single processor of an Encore Multimax multiprocessor. The Encore MultiMax is a 16 node, shared-memory multiprocessor, with an NS 32032 (0.75 MIPS) microprocessor at each node. We also include the runtime on a more widely available machine, a DEC VaxStation 3200, for reference purposes. These times include all costs involved in processing the trace files from beginning to end, including the time spent processing the ATMS commands and the time spent reading the trace files from disk.

In Table 2 we give a breakdown of where time is spent in our implementation. File access time accounts for a substantial portion of the runtime, a portion which is not relevant when measuring true

ATMS performance. We will therefore ignore file access time in evaluating parallel ATMS performance. Also, if we ignore file access time, all but an extremely small amount of the runtime is spent processing Justify-Node commands. Since they are the clear bottleneck, we will concern ourselves strictly with Justify-Node commands for the remainder of this paper. All runtimes cited in the future will measure only the amount of time spent processing Justify-Node commands.

3.5 The Environment Database

An ATMS environment has a large amount of information associated with it. The environment is either consistent or nogood. It could appear in the labels of many nodes, or it could appear in none. When the ATMS computes the union of two environments, it needs access to the information associated with the resulting environment. The information could be recomputed each time the environment is encountered, but some of the information is quite expensive to gather. In order to avoid having to recreate this information, each encountered environment is given a unique physical representation in memory. In other words, if two nodes have an environment E in their labels, they both really have pointers to the unique structure representing the environment E. The unique representation, when combined with a method for finding this representation for a given environment, allows us to do expensive checks once per environment, not once per encounter.

The method we choose for quickly finding the unique representation of a given environment is an environment hash table. Every environment which is encountered at any time in a problem execution is stored in this hash table. When a new environment is encountered, the hash table is checked to see if the environment has been encountered before. If it has not, the environment is added to the table. In this way, the ATMS can store information about environments which can be quickly retrieved if needed.

When creating new consistent or nogood environments, the ATMS also needs quick access to large sets of existing environments. For example, when a previously undiscovered environment is encountered, it must be checked for consistency. The ATMS must check the new environment against all nogood environments which are smaller than it. If the environment is subsumed by some nogood, then it is clearly also nogood. Similarly, when a new nogood is discovered, all consistent environments which are larger than this nogood must be checked for subsumption. Any consistent environment which is subsumed becomes nogood.

The data structure which seems to best serve these purposes is a pair of tables. Each table consists of an array of lists of environments, sorted by environment size. Thus to find all environments of size n, we must simply traverse the list in position n of the array. One table, the *Consistent* table, holds all consistent environments encountered. The other table, the Minimal NoGood (*MNG*) table, holds the set of nogoods which are not subsumed by any other nogood. Minimal nogoods are kept in order to keep environment consistency checks as quick as possible; an environment which is subsumed by a nogood is clearly also subsumed by any subset of that nogood.

We make two modifications to the simple MNG table for efficiency. First, we handle unary and binary nogoods as special cases. The assumption data structure has a field entitled Binary which keeps track of unary and binary minimal nogoods. If the environment $\{i, j\}$ is nogood, then bit i in the Binary field of assumption j and bit j in the Binary field of i are set. If the environment $\{i\}$ is nogood, then bit i of the Binary field of i is set. The second modification involves the Nogoods field of the assumption data structure. Any minimal nogood environment in the MNG table will also be in the Nogoods table of each assumption in the environment. These two modifications allow the ATMS to find all minimal nogoods containing a given environment extremely quickly.

The Consistent and MNG tables form what we call the *environment database*. The environment database, together with the environment hash table, makes the following frequent operations extremely fast:

- Find a particular environment, with all its associated information.
- Find all consistent environments smaller (or larger) than a given environment.
- Find all minimal nogoods which are smaller than a given environment.

Finally, the most prevalent operation in the ATMS is the subset test. The environment representation must therefore be chosen so that subset testing is extremely fast. A bit vector representation works extremely well. A one in bit i of the vector indicates the presence of assumption i in the environment. The bit vector representation allows subset testing by simply ANDing the bit vector of one environment with the complement of the bit vector of the other environment. A bit vector representation also allows fast hash function computation.

3.6 The Cross Product

When we handle an Update request, we need to compute the cross product of a number of minimal environment lists, as was described previously. Assume we wish to take the cross product of n minimal environment lists $l_1, l_2, \ldots, l_n$, with l_1 being the incremental update. We begin the cross product computation by first checking to make sure that each l_i is non-empty. If any list is empty, then the cross product is empty.

Next we loop through each list, creating m_i, the cross product of l_1 through l_i. We begin with $m_1 = l_1$, and at each iteration we will compute $m_{i+1} = m_i \times l_{i+1}$, where both m_i and m_{i+1} are in minimal environment list form. We do this by taking the union of each environment E in m_i with each environment F in l_{i+1}, using the method for finding unions to be described in the next section. The resulting list is then minimized.

We can greatly decrease the amount of time it takes to compute m_n by using the following two techniques. First, if some environment E in m_i is subsumed by some environment in the consequent of the justification which we are updating, then clearly every environment in $m_{i+1} \ldots m_n$ which is generated from E will also be subsumed by this environment. We therefore check each environment in m_i against each environment in the label of the consequent and discard those which are subsumed. Line 13 in Table 3 shows the number of environments which are discarded in this way for the three trace files.

Second, consider taking the cross product of m_i with l_{i+1}. If some environment E in m_i is subsumed by some F in l_{i+1}, then clearly E will be in m_{i+1}. Since all environments which would result from taking the union of E with some environment in l_{i+1} are supersets of E and since E is in m_{i+1}, none of the resulting environments will be present in m_{i+1}. We therefore check each E in m_i for subsumption against each F in l_{i+1}. If E is subsumed, then we can simply place it into m_{i+1}, and not take the union of E with each environment in m_{i+1}. Line 14 in Table 3 shows the number of times that this occurs for the three trace files.

If we compute the cross product, using these two techniques, the result is a minimal environment list which represents the change to the label of the consequent node n. If the consequent is not the FALSE node, we add each environment in our cross product to the label of node n. We must now restore minimality in the label by checking every environment previously in the label for subsumption against every environment just added to the label. We then propagate the cross product list, which represents the changes to the label of node n, to every justification which has node n as an antecedent.

If the consequent is the FALSE node, then our cross product list is a set of environments which were previously consistent but have just become nogood. We add them to the MNG table, and sweep through the Consistent and MNG tables looking for subsumed environments. If an environment in the Consistent table is subsumed, it is removed from the table and from the labels of all nodes which contain it (found in the Where field of the environment). If an environment in the MNG table is subsumed, it is removed from the table.

3.7 The Union of Two Environments

Computing the union of two environments is an extremely frequent and potentially extremely costly operation in the ATMS. In most ATMS problems, the vast majority of all unions result in a nogood environment (94%, 97%, and 83% for QPE, BUG, and 8-Q, respectively). Table 3 shows the empirical numbers for the three trace files. Line 1 gives the total number of unions computed, with Lines 2 and 3 giving the number of those which result in consistent and nogood environments, respectively. It is therefore to our advantage to have a quick check to see if the result of a union is a nogood. The

Table 3: Results of environment unions.

	QPE	BUG	8-Q
1. Total unions	44598	135851	16440
2. Consistent	2636	4503	2776
3. Nogood	41962	131348	13664
4. Total adds	46909	141788	16440
5. Ortho	40499	128887	0
6. Binary	973	1149	13664
7. Same	1793	4089	0
8. Exist OK	2326	3976	0
9. Exist NG	236	755	0
10. Non-binary	254	557	0
11. New env	828	2375	2776
12. Imm. Ortho	39958	127050	0
13. Old	2580	2566	0
14. Bypass	1615	2853	0

method of union computation which seems to allow the fastest recognition of nogood environments is an assumption by assumption method. That is, given two environments E_1 and E_2, we compute the union by successively adding the assumptions in E_2 into E_1, computing an intermediate environment at every step. The union function returns either a consistent environment E_3, which is the union of E_1 with E_2, or it returns nothing, indicating that the union of E_1 with E_2 is nogood. Since nogoods can never appear in node labels, they do not have to be retained. The union computation is therefore complete as soon as we know that the union will be nogood. We begin the union computation by making E_1 the larger environment, the one with more assumptions. The environments are swapped if this is not true. If both are the same size, we make the one with the larger hash function E_1. This step decreases the number of assumptions which need to be added. It also assures us that if we compute the same union more than once, the steps we do the first time will be repeated each successive time.

The next step is to begin a loop through all n members of E_2. At each iteration i of the loop, we have an environment F_i which represents the result of adding the first i assumptions from E_2 into E_1. If F_i becomes nogood at any point, we may break out of the loop and quit. We begin with $F_0 = E_1$. At the beginning of each iteration we have some consistent F_i and the ith assumption of E_2, A_i, which we wish to add to it. At the end of the iteration we either know that the union is nogood or we have some consistent F_{i+1}, which is the union of F_i with A_i. F_n is the union of E_1 with E_2. Line 4 in the table gives the total number of iterations of this loop which are performed.

Our first step within iteration i of the loop is to do a quick check to determine whether the union could possibly result in a consistent environment. We do this by doing a bitwise AND of the Present field of E_2 with the Orthogonal field of F_i (see section 3.3 for a description of these fields). If the result is non-zero, then we know that some member of E_2, when added to F_i, would yield a nogood environment. Since this nogood environment would clearly be a subset of $E_1 \cup E_2$, we then know that $E_1 \cup E_2$ is nogood and we can quit (Line 5 in the table). If the result is zero, however, it tells us nothing and we proceed.

Next we check for binary nogood subsumption of F_{i+1}. Since F_i is consistent, we only need to check F_{i+1} against those binary nogoods which contain assumption A_i. Thus we can check binary subsumption by taking the bitwise AND of the Present field of F_i with the Binary field of A_i. If the result is non-zero, then some assumption in F_i participates in a binary nogood with A_i, and thus F_{i+1} is nogood (Line 6 in the table). Since we also learn that adding A_i to F_i yields a nogood, we set the bit corresponding to A_i in the Orthogonal field of F_i. If the result is zero, again we proceed.

Next we check to see if A_i is a member of F_i. We do this by extracting the bit corresponding to A_i from the Present field of F_i. If it is set, then $F_{i+1} = F_i$ and this iteration is complete (Line 7 in the table).

Otherwise we form a partial environment structure for F_{i+1}, with all fields except the Constituents

field complete. The Present field for F_{i+1} is equal the Present of F_i with the bit for A_i set. We compute the hash function for F_{i+1}, and then check for the existence of this environment in the environment database. If it exists and is consistent, then this iteration is complete (Line 8 in the table). If it exist and is nogood, then the entire loop is complete (Line 9 in the table). In this case, we may again set the bit for A_i in the Orthogonal field of F_i. If it does not already exist, then we proceed.

If we've gotten this far, we know that our F_{i+1} will be added to the environment hash table, so we fill in the Constituents field by adding A_i to the Constituents field of F_i. We now add this environment to the table.

Next we check F_{i+1} for subsumption by a non-binary nogood. As with the binary nogood check, since we know that F_i is consistent we only need to check F_{i+1} against nogoods which contain A_i. We check F_{i+1} against every non-binary nogood smaller than it which contains A_i, which can be found in the Nogoods field of the assumption data structure for A_i. If it is subsumed by some nogood, then the loop is complete (Line 10 in the table). Again, we may set the bit corresponding to A_i in the F_i's Orthogonal field. Otherwise, we know that F_{i+1} is consistent. We add it to the Consistent table and the iteration is complete (Line 11 in the table).

While this seems like a somewhat cumbersome way of computing the union, in practice it is extremely effective in recognizing unions which will result in nogood environments quickly. Line 12 in Table 3 shows the number of unions which can be aborted after the first test against the Orthogonal vector. One can see that a simple bit vector AND successfully recognizes most unions which will result in a nogood.

This concludes our discussion of an efficient sequential implementation of the ATMS. As was discussed in section 3.4, our implementation is quite competitive with existing ATMS implementations. We use the sequential implementation which we have described as the basis of comparison for the parallel implementations which we describe in the remainder of this paper.

4 Modifications for Parallel Implementation

We now discuss the modifications which are necessary to allow the preceding algorithm to be executed in parallel. Our goal is to exploit as much parallelism as possible, but we can not afford to introduce a large amount of redundant work in doing so. When designing algorithms for massively parallel processors, it is possible and often necessary to radically change the data structures and algorithms from those which would be used on a sequential implementation. The increase in available parallelism which these changes bring about often outweighs the increased amount of work which is done. However, since we will be executing this algorithm on a modest number of processors, our speedups will suffer if the parallel implementation does a large amount of work which the sequential implementation does not do. We therefore do not stray far from the data structures and algorithms used in the efficient sequential implementation.

4.1 Division of Work

The overall structure of our parallel ATMS is quite similar to the structure of the sequential ATMS. The ATMS and the problem solver run concurrently, sharing commands and data through a shared command queue. The problem solver places Create-Node, Create-Assumption, Justify-Node, and Node-Query messages on the queue. The problem solver blocks and waits for a reply after it places a Node-Query message on the queue. A number of processors are allocated to work on the ATMS. The ATMS processors pull commands off the queue and perform the requested actions. In order to allow a greater amount of parallelism, we no longer require that node labels be made sound and complete at the completion of each command. This requirement would necessitate the synchronization of all processors after each command, an operation which would greatly constrain our ability to distribute work among the processors. We now only require that labels be made sound and complete before a Node-Query command is answered. Thus, Node-Query commands are now somewhat expensive, since they require a global synchronization. Create-Node and Assume-Node messages again require very little work to be done, and are dealt with quickly. Justify-Node messages are the source of almost all of our parallelism. Since they require by

far the most computation time. they are the commands which afford the most oppurtunity to distribute work.

In order to decrease contention for tasks. each processor has its own Update request stack. When a processor completes a task. it looks for a new task in the following places. First. it checks it's own Update request stack. If it is empty. then the processor checks the global command queue. If the next command on the command queue is a Node-Query (or if the command queue is empty) the processor becomes idle. When all processors are idle. one processor processes and removes the Node-Query command. thus unblocking the problem solver and allowing the problem solving process to proceed. We call this Algorithm A1. We later provide variations of this basic algorithm.

4.2 Locks

In our shared memory implementation. all the processors access the same data structures. We therefore need a number of mutual-exclusion locks to control simultaneous access to shared data. We begin by using straightforward locking techniques, and later modify our approach based on the observed bottlenecks.

The environment hash table is locked by bucket. Whenever a processor wants to do either an environment lookup or an environment addition. it must obtain a lock on the appropriate bucket before it may access anything in the bucket. Since there are thousands of buckets and. for now. at most 16 processors and very little time is actually spent inside the lock, contention for the hash table buckets is not a problem.

Each environment has a lock to control access to its Contra flag and its Where field. The lock is used to enforce the following conditions:

- No nogood environment may be added to a node's label.
- When an environment becomes nogood. it must be removed from the label of every node which contains it.

The above conditions are also used to avoid redundant work. When a processor wishes to change an environment's status to *nogood*. it first obtains a lock on the environment. It then checks the environment's Contra flag. If the flag is set (i.e. the environment is nogood), then some other processor must have already discovered that this node is nogood and the processor can stop; any work done with this environment would be redundant. Otherwise. the Contra flag is set and the lock is released. The environment is then removed from the label of every node in which it appears. When a processor wishes to add an environment to a node label. it obtains the environment lock and checks the Contra flag. If the flag is set, then another processor has discovered that the environment is nogood and it should therefore not be added to the node label. If the flag is not set. the environment is added to the node label and the environment lock is released. By using the lock in this way, we are assured that no node label can contain a known nogood. Since a typical ATMS application generates thousands of environments. contention at this point is usually not a problem.

Node labels are accessed and modified by many processors, thus we must provide locks to protect them. When an Update request is being processed, and a node is an antecedent to the justification, the node's label is accessed. Similarly, the label of the consequent of the justification is also accessed. Since computing the label cross product could take an enormous amount of time, we would like to avoid holding the node lock for the duration of the cross product. We therefore choose to lock the node. copy the label, and immediately release the lock. An antecedent label can simply be copied because any change to the label will be propagated to this justification. While this can create redundant work. the resulting answer will still be correct. A consequent label can be simply copied for the following reason. The only way in which an environment is removed from the label of a node is if it becomes nogood or if it is subsumed by a new label. If an environment becomes nogood. then clearly anything subsumed by it also becomes nogood. If an environment is subsumed. then clearly anything subsumed by it will also be subsumed by the new environment. In either case. it is valid to discard cross product environments which are subsumed by consequent label environments.

When revising the label of a node, the node is locked, and the environments in the current node label are checked for subsumption against the new environments and vice-versa. The node label is revised, any changes in the label are recorded, and the lock is release. Again, since there are thousands of nodes contention is usually not a serious problem.

Contention for environment and node locks can be a problem, however, when many processors are working in the same part of the inference graph. Since environments are generated entirely by propagation, it is likely that if two processors are working on tasks which resulted from the same node revision, they will encounter identical environments more often than if they were working on unrelated activations. Similarly, if two processors are working in the same part of the graph, they are more likely to want to access the same node label. In order to avoid this type of contention, it is desirable for the processors to be well distributed throughout the inference graph.

4.3 The Environment Database

We now discuss the modification necessary to allow concurrent access to the environment database. The modifications we have discussed so far have been relatively local. They have involved such changes as a lock on a bucket in a table, or a lock on a single environment or node. The environment database, however, is a very global structure. It keeps track of the consistency of all environments in the entire problem. A single change could conceivably affect every environment in the environment database. The environment database must allow the following operations:

- Determine whether a new environment is consistent.
- Add a new consistent environment.
- Add a new nogood and find all previously consistent environments which are now nogood.

It must also keep the database self-consistent while these operations are occurring. Since the ATMS spends much of its time creating new environments and checking them for consistency, we cannot tolerate a high latency on consistency checking. At the same time, however, most new environments which are encountered are nogood, so to avoid superfluous work we want a new nogood to be recorded as soon as possible.

We initially used a single global lock to control access to both the Consistent and MNG tables. When an environment needed to be checked for consistency, the processor obtains the global lock, checks the environment against the MNG table, and releases the lock. When a new consistent environment is added, the processor obtains the lock, adds the environment to the Consistent table, and releases the lock. When a new nogood is registered, the processor obtains the lock, checks all consistent environments for subsumption, changes those which are subsumed into nogoods, and releases the lock. Since the ATMS spends a substantial percentage of its time within this lock (3-15% for the three traces), this global locking approach appears somewhat suspect.

5 Results

We now present the results of executing the three problem solver traces on our parallel ATMS. Because Node-Query information was not required when the traces were originally generated, these traces do not record this command. The absence of this command does not affect the performance of the sequential ATMS significantly, since Node-Query commands take so little time to execute. In our parallel ATMS, however, the lack of these commands obviates the need for global synchronization. Thus, the results we present here are optimistic, as the synchronization is done only at the completion of the entire trace. In applications where Node-Query commands are frequent, one would expect less available parallelism.

The ATMS traces we examine seem to present abundant opportunites for parallelism. Their inference graphs are extremely large, with thousands of justifications capable of being distributed among the processors (see Table 1). The only limiting factor would appear to be the global lock on the Consistent

Table 4: Task times for algorithm A1

	QPE	BUG	8-Q
Tasks	2584	4165	1192
Ave task time (s)	0.015	0.019	0.028
Max task time (s)	2.42	35.31	0.36
Total runtime (s)	39.34	82.31	33.72

Table 5: Task times for algorithm A2

	QPE	BUG	8-Q
Tasks	18780	16576	3308
Ave task length (s)	0.002	0.005	0.010
Max task length (s)	0.86	6.88	0.34
Total runtime (s)	39.34	82.31	33.72

and MNG tables. However, if we examine Figure 3 we see that the speedup obtained for Algorithm A1 is disappointing. The speedup is greatly below what one would expect, even given the global lock. The sequential ATMS spends 3%, 15%, and 6% of its time within the lock for QPE, BUG, and 8-Q, respectively. If this were the only parallelism limitation, we would expect speedups of 7 or more. Clearly, parallelism is being limited in some other way.

The most serious bottleneck appears to be processor idle time. Figures 4 through 6 show the percentage of time each processor spends doing useful work as compared to the percentage spent waiting on locks and the percentage spent idle. Note that the speedup obtained is not equal the product of the processor utilization with the number of processors used. This is due to number of factors. First, our speedup numbers are obtained by dividing the parallel execution time by the execution time of the best sequential implementation. There are a number of overheads involved in the parallel implementation, such as environment list copying and redundant checks, which can reduce the speedup when compared to a sequential implementation without these overheads. Second, the parallel ATMS does not necessarily do the same amount of work that the sequential ATMS does. For example, the parallel ATMS can process the justifications in a different order than the sequential ATMS. While the answer arrived is the same, the amount of progation necessary to get to this answer may differ. Third, there a number of hardware issues, including bus bandwidth and cache interactions, which can preclude linear speedups. These issues are not reflected in the utilization graphs which we present.

From examining Figures 4 through 6, it becomes clear that we have a problem with the distribution of work among the processors. Processors are spending a large amount of time idle, without a task to execute. What we have here is essentially a bin-packing problem. We have a certain number of tasks of varying size to execute, and we wish to divide them among a number of processors so that each processor takes approximately the same amount of time to complete them. This near equal division of tasks is normally quite possible given a large number of tasks to distribute; the large number of tasks serves to smooth out the variations in grain size. However, two factors make this untrue in Algorithm A1, First, the variation in grain size is enormous. In the BUG trace, for example, the processing of one single justification accounts for more than 40% of the run-time of the trace (see Table 4). Second, as the trace progresses the size of the inference graph increases. The amount of work required to process a single justification depends heavily on how much label propagation must be done. In the early stages of the trace, the small size of the inference graph limits the amount of propagation necessary. As the trace progresses, however, the graph becomes larger, with the potential for more necessary propagation. The grain size therefore grows as the trace progresses, and one would expect the enormous grains to be near the end of the trace. The combination of some extremely large grains with the tendency for the large grains to be towards the end of the trace combine to make it extremely likely that one processor will be stuck with a large grain while the other processors have nothing to work on.

In order to alleviate the grain size problem, we decrease the task size. Instead of each problem solver issued command being a single task, we now consider each Update request to be a task. In Algorithm A1, once the command queue becomes empty the processor simply quits. Now, in Algorithm A2, an idle

All Other Locks
Node Locks
Environment Database Locks
Idle Time
Processor Utilization

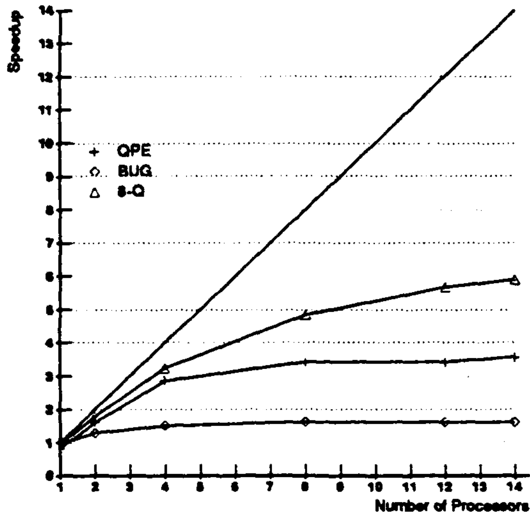


Figure 3: Speedup for Algorithm A1

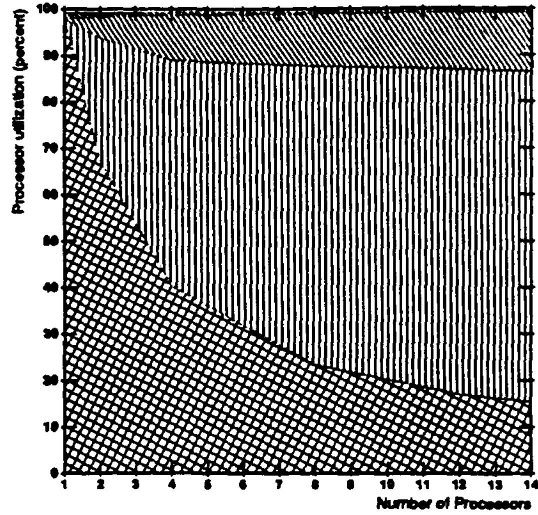


Figure 5: Processor utilization for BUG, Algorithm A1

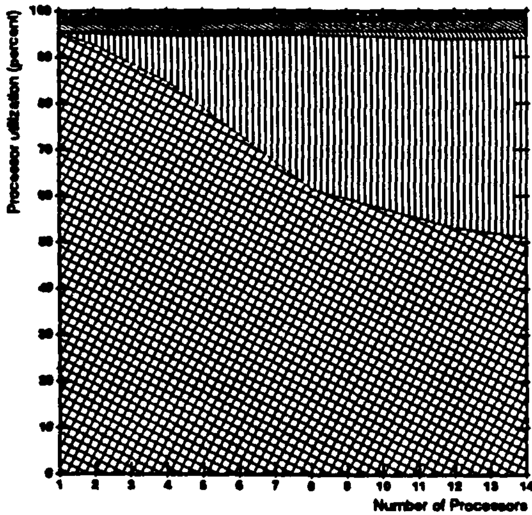


Figure 4: Processor utilization for QPE, Algorithm A1

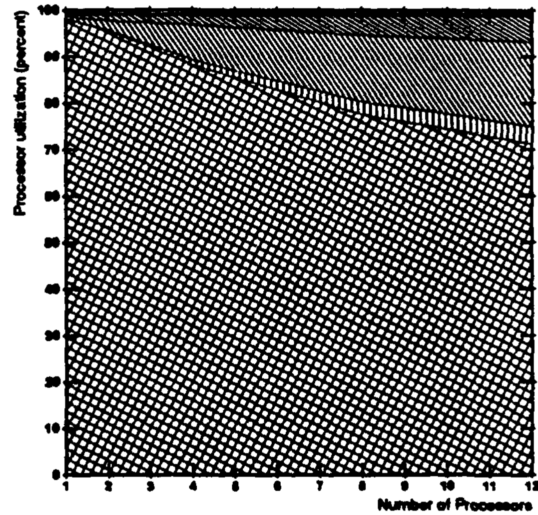


Figure 6: Processor utilization for 8-Q, Algorithm A1

processor attempts to steal an Update request from the Update stacks of the other processors. In this way, work can be distributed among the processors even after the command queue has been emptied. The only cost for this modification is the introduction of contention in the Update stacks. Comparing Tables 4 and 5, we see that by decreasing the task size we have greatly increased the number of tasks and greatly reduced both the average and maximum task size. Figures 8 through 10 show that while idle time has been greatly decreased from that of Algorithm 1, it is still substantial for the BUG trace. This is mainly because the largest task still takes 6.88 seconds, which is 8% of the total runtime. The net result of our modification (Figure 7) is that the speedup is greatly increased from that of Algorithm A1, but it is still far from ideal.

The most serious bottleneck in our parallel implementation now appears to be the environment database lock. In order to increase concurrency in the environment database, we introduce another variation on our basic algorithm. In Algorithms A1 and A2, only a single processor may access the database at one time. Our modification, which we call *Modal access*, allows a number of processors to access the table concurrently, while still maintaining the stringent consistency requirements of the environment database.

The problem in allowing concurrent access to the database comes from the potential simultaneous additions of a consistent environment and a nogood environment. In order to add the consistent environment to the database, we must know that it is not subsumed by any environment in the MNG table. To add the nogood to the database, we must remove all environments which are subsumed by it from the Consistent table. These requirements seem to place serious sequentiality constraints on modifications to the database. In order to avoid these constraints, we add to the environment database a mode of access indicator. The three access modes are:

- Mode 0 — No processor is currently accessing the database.
- Mode 1 — Only consistent environments may be added to the database.
- Mode 2 — Only nogood environments may be added to the database.

In order for a processor to add a new consistent of nogood environment, the environment database must be in the appropriate mode. If a processor needs the table to be in Mode 1, it calls a procedure called Adder(). Adder() waits until the database is in either Mode 0 or Mode 1. If the database is in Mode 0, it changes the mode indicator to Mode 1. It increments a counter of how many processors are within the database, and then proceeds. Once this processor is finished using the database, it calls the procedure ReleaseAdder(). This procedure decrements the counter, and if the counter is now zero it changes the mode indicator back to 0. The procedures Deleter() and ReleaseDeleter() are defined identically except that they move the database into Mode 2.

If a processor wishes to add a new environment to the database, it first calls Adder() to bring the database into Mode 1. It then checks the environment for consistency. If the environment passes the check, it is added to the Consistent table. Since the environment database is in Mode 1, the processor is guarenteed that no nogoods are being added. Thus if the environment passes the consistency check, the environment will remain consistent until the Mode is changed. Once the consistency check has been made, the processor is finished using the database and calls ReleaseAdder().

If a processor wishes to add a new nogood, it calls Deleter() to bring the database into Mode 2. The processor then adds the nogood to the database, and then it sweeps through the Consistent and MNG tables flushing out all subsumed environments. Since no consistent environments are being added, we can be assured that we will check all consistent environments. While it is true that nogood environments can be added at this point and we can consequently miss one which is subsumed, this situation is sufficiently rare and harmless that we do not need to be concerned with it. Remember that the sweep through the MNG table is for efficiency reasons only, and it does not affect the correctness of the algorithm. Once the sweeps through the two tables are complete, the processor calls ReleaseDeleter().

We can modify the above slightly to increase concurrency. When new nogood environments are generated, they usually come in lists. We can therefore distribute the nogoods in a single list among a number of processors. This is accomplished by keeping a global list of nogoods to be added. When a processor has a list of new nogoods to be added, it adds them to this list, calls Deleter(), and then goes

All Other Locks
Node Locks
Environment Database Locks
Idle Time
Processor Utilization

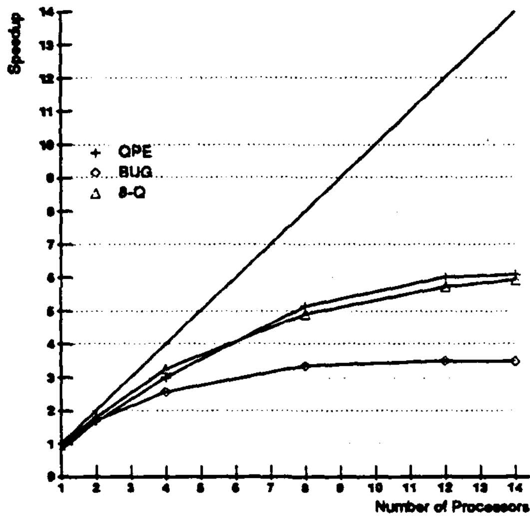


Figure 7: Speedup for Algorithm A2

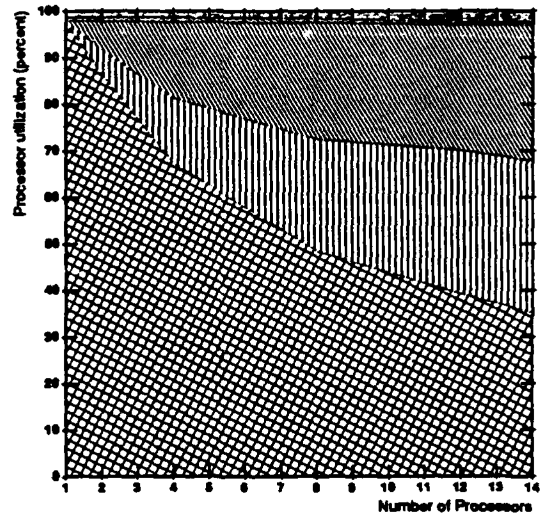


Figure 9: Processor utilization for BUG, Algorithm A2

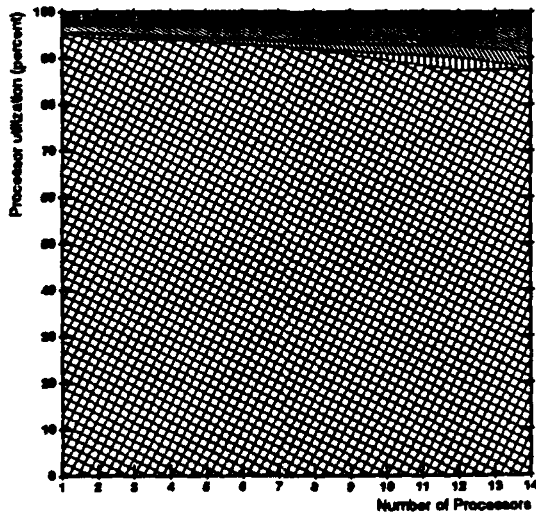


Figure 8: Processor utilization for QPE, Algorithm A2

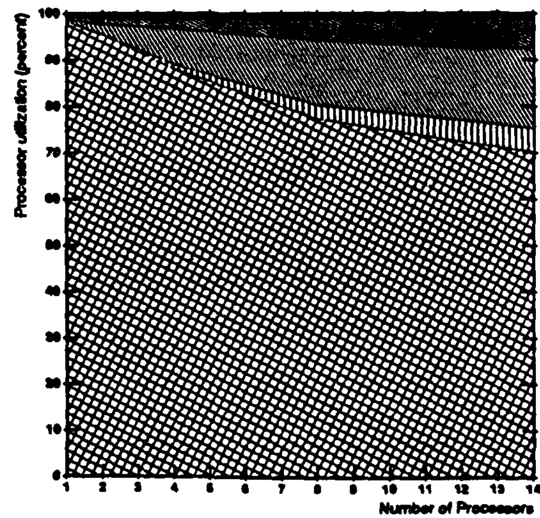


Figure 10: Processor utilization for 8-Q, Algorithm A2

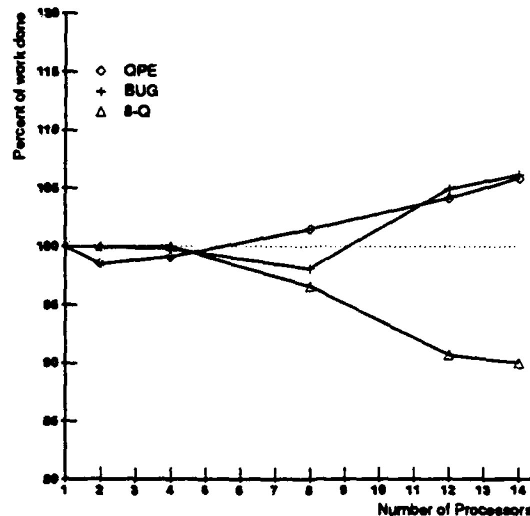


Figure 2: Amount of work done (as a percentage of that for P=1)

into a loop, pulling off nogoods from this list until the list is empty. Now, when a processor calls Adder() and finds the database to be in Mode 2, instead of simply waiting for the Mode to change, the processor also pulls nogoods off the global list and processes them.

The speedups obtained from Algorithm A3 (Figure 11) are still far from ideal. While contention for the environment database is greatly reduced, it is still substantial. We also still have a substantial speedup reduction due to processor idle time.

We have yet to examine one possible cause of reduced speedup in the parallel implementation, redundant work. In the ATMS, it is difficult to establish a measure of how much "work" is being done. There are a number of routines which are called often and take large amounts of time, yet no one routine dominates the others. One routine, the subset test routine, appears to be a reasonably accurate measure of how much work is being done. Subset testing accounts for more of the runtime of the sequential ATMS than any other routine. Also, many of the other routines which take time do large numbers of subset tests. If these routines are being called more frequently, this would be reflected in the number of subset tests done. Figure 2 gives a picture of how many subset test are done for the 3 traces. Though the subset test numbers show interesting trends as the number of processors grows larger, the differences for less than 14 processors are not significant. According to our subset measure of work, the parallel ATMS does between 90% and 106% of the work of the sequential ATMS for 14 or fewer processors.

5.1 Other Approaches

Variation in grain size is still a problem in our implementation. Furthermore, the problem would be much more severe if Node-Query commands were more frequent. One possible way to further decrease the grain size would be to split Update requests into smaller pieces. In Algorithm A3, an Update request contains a list of new environments which have been added to the label of an antecedent. In order to decrease the size of a single grain, we could split this list into many smaller lists. We could use a heuristic to determine approximating how long an Update task will take. Depending on the estimate, the list can be split so that other processors will not go idle while this task is being executed. In the extreme, Update requests can be split into single new antecedent environments.

Performing Updates with smaller lists of environments can generate a large amount of avoidable work.

All Other Locks
Node Locks
Environment Database Locks
Idle Time
Processor Utilization

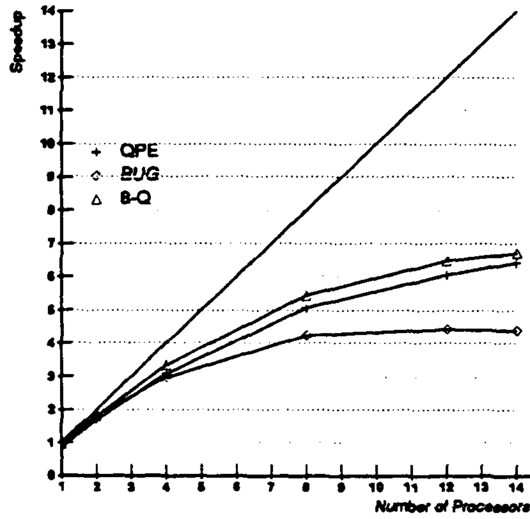


Figure 11: Speedup for Algorithm A3

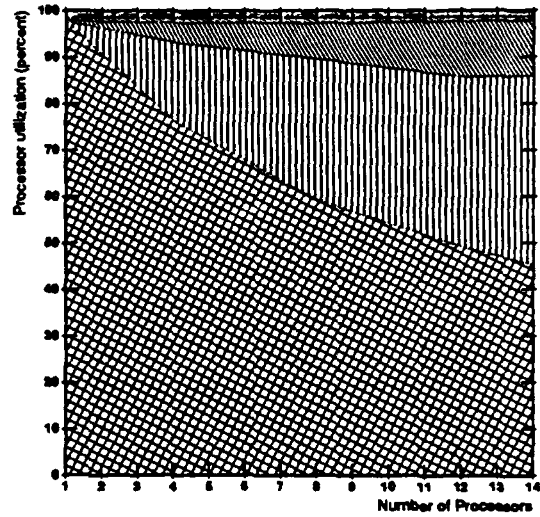


Figure 13: Processor utilization for BUG, Algorithm A3

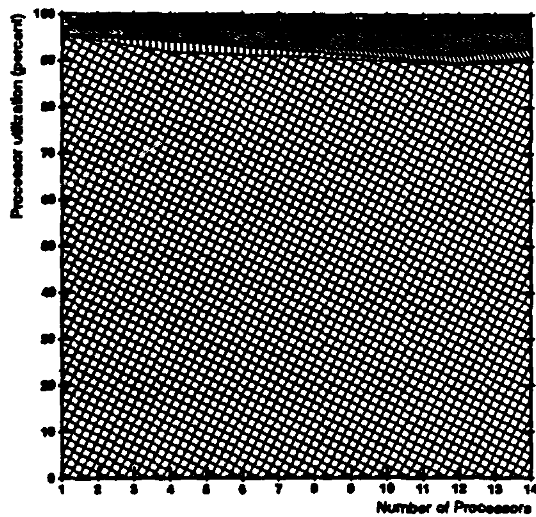


Figure 12: Processor utilization for QPE, Algorithm A3

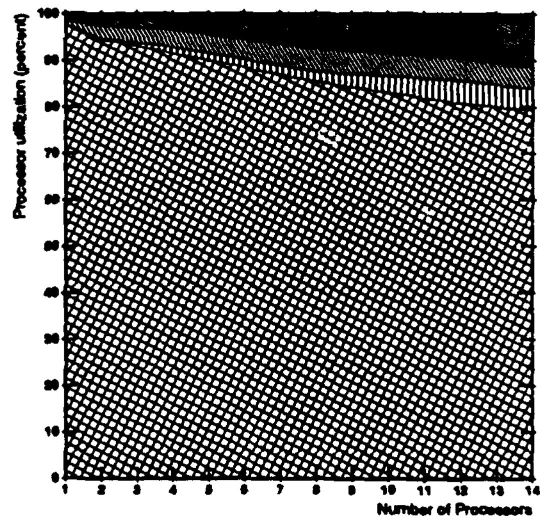


Figure 14: Processor utilization for 8-Q, Algorithm A3

however. Consider the following cross product:

$$(\{A\}, \{B\}, \{C\}, \{D\}) \times (\{D\})$$

If we simply perform the cross product, we get the list $(\{D\})$. If we split the list $(\{A\}, \{B\}, \{C\}, \{D\})$ into two parts and perform seperate cross products, however, we get $(\{A, D\}, \{B, D\})$ from the first part and $(\{D\})$ from the second. Now, instead of propagating a single list of length one to the successors of the consequent, a list of length two and a list of length one are propagated.

We can see this happening in the BUG trace file. The largest Update task in the trace arises from a justification:

$$x_1, x_2, x_3, x_4, x_5 \Rightarrow n$$

The Update request comes from x_1 with a list of 8 new environments. Nodes x_2, x_3, and x_4 all have 8 environments in their labels, and node x_5 has 1 environment in its label. The resulting cross product environment list could contain as many as 8^4 environments. It actually produces only 293 environments, because of nogood subsumption and list minimality. If the incoming new environment list of 8 environments is split into two environment lists of 4 environments each, one resulting cross products has 367 environments and the other has 55. The net effect of splitting this single Update request into two smaller requests is substantial. The sequential execution time for BUG increases from around 82.31 seconds to 119.96 seconds. an increase of 46%. While we could have all processors working on a single Update synchronize and combine their results before propagating them on, the added synchronization combined with the fact that the pieces of a split Update are not necessarily smaller than the whole Update combine to make such an action unwise.

Due to the above reasons, our initial efforts to go to a smaller grain have not resulted in much success. In order to get significantly more speedup from some ATMS instances, we need to find a natural task grain which is smaller than that of an Update request. Unfortunately, no obvious alternative presents itself.

6 Conclusions

In this paper, we have presented the details of implementing both a serial and a parallel ATMS. The results we obtained from executing the parallel implementation on an Encore MultiMax allow us to draw a number of conclusions about executing the ATMS in parallel.

- The traces we examined seemed to present abundant opportunities for parallelism. They consisted of thousands of relatively independent tasks, capable of being distributed among a number of processors. However, this apparent abundance of parallelism proved to be somewhat elusive to exploit.
- The obvious source of parallelism in the ATMS, the thousands of justifications, generated grains which varied enormously in size. In one trace, for example, a single justification accounted for 43% of the total runtime, making effective parallel distribution of grains impossible. In order to make grain sizes more uniform, we were forced to decrease grain size by treating a single justification update as a task. We also introduced the notion of modal access to the environment database in order to alleviate the sequentiality constraints imposed by the global consistentency requirements. Modal access requires that, at any one time, environments can be added in parallel or removed in parallel, but not both.
- With these modifications. we were able to obtain speedups of between 4.4 and 6.7 using 14 processors for the three trace files which we examined. Further speedups were limited by a number of factors. including still too large of a variation in task grain size, processor contention for numerous mutual-exclusion locks. and hardware contention issues.
- We further note that we examined the best case scenario. where Node-Query commands are infrequent and global synchronization is necessary only at the completion of the entire trace. While it's not clear what the average case would be, it would almost certainly present fewer opportunities for parallelism.

- By combining a highly efficient C-based implementation with a modest degree of parallelism, we have created an ATMS implementation which is significantly faster than currently available LISP-based implementations.
- We believe that in order to acheive near-linear speedups, parallelism in the ATMS must be exploited at a finer grain than that used in the three algorithms presented here.

While in this paper we have explored how ATMS parallelism can be exploited on a shared-memory multiprocessor, a related question is how it can be exploited on other types of parallel machine architectures. Michael Dixon and Johan deKleer [3] have studied the implementation of the ATMS on the Connection Machine, a massively parallel processor with between 16K and 64K processors [7]. Their implementation has shown promise in the tests which they have tried, but it remains to be seen whether it will offer a dramatic speed advantage for a wide range of problem solver domains.

In the future, we plan to investigate the tradeoffs between using a shared-memory architecture versus a message passing or Connection Machine architecture for exploiting parallelism in the ATMS. We plan to investigate how the grain size can be reduced without introducing an enormous amount of extra work. We also hope to integrate our parallel ATMS with LISP-based problem solvers, allowing the exchange of commands and data through inter-process communication.

7 Acknowledgements

We would like to thank Johan deKleer and Ken Forbus for providing us with problem solver trace files. We would like to thank Hiroshi Okuno for his assistance in the initial stages of this research. This research is supported by DARPA contract N00014-87-K-0828. Edward Rothberg is also supported by an Office of Naval Research graduate fellowship. Anoop Gupta is also supported by a faculty award from Digital Equipment Corporation.

References

[1] deKleer, J., "An Assumption-based Truth Maintenance System", *Artificial Intelligence*, **28**, 1986.

[2] deKleer, J., "Problem Solving with the ATMS", *Artificial Intelligence*, **28**, 1986.

[3] Dixon, M. and deKleer, J., "Massively Parallel Assumption-based Truth Maintenance", *Proceedings of the AAAI*, 1988.

[4] Filman, R.E. "Reasoning With Worlds and Truth Maintenance in a Knowledge Based System", *Communications of the ACM*, **31**, 1988.

[5] Forbus, K., "The Qualitative Process Engine", University of Illinois Technical Report No. UIUCDCS-R-86-1288, December, 1986.

[6] Gupta, A., Forgy, C., Kalp, D., Newell, A. and Tambe, M., "Results of Parallel Implementation of OPS5", *Proceedings of the ICCP*, 1988.

[7] Hillis, D., *The Connection Machine*, MIT Press, Cambridge, Massachusetts, 1985.

[8] Okuno, H., "An Efficient Parallel Execution of the ATMS", to appear.