AD-A234 524

2

RADC-TR-90-290
Final Technical Report
November 1990

CONTENT ADDRESSABLE MEMORY PROJECT

Rutgers University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. 6403

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC-TR-90-290 has been reviewed and is approved for publication.

APPROVED: [illegible]

DAVID M. GUBBINS
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: [illegible]

IGOR G. PLONISCH
Directorate of Plans & Programs

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

CONTENT ADDRESSABLE MEMORY PROJECT

Saul Y. Levy
J. Storrs Hall
Donald E. Smith

Contractor: Rutgers University
Contract Number: F30602-88-D-0027
Effective Date of Contract: 13 September 1988
Contract Expiration Date: 30 January 1990
Short Title of Work: Content Addressable Memory Project
Program Code Number: 8E20
Period of Work Covered: Sep 88 - Jan 90

Principal Investigators: Dr. Saul Y. Levy and Dr. J. Storrs Hall
Phone: (201) 932-3523

RADC Project Engineer: David M. Gubbins
Phone: (315) 330-2158

This research was supported by the Defense Advanced Research Projects Agnecy of the Department of Defense and was monitored by David M. Gubbins, RADC (COTD), Griffiss AFB NY 13441-5700 under Contract F30602-88-D-0027.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
November 1990

3. REPORT TYPE AND DATES COVERED
Final Sep 88 - Jan 90

4. TITLE AND SUBTITLE
CONTENT ADDRESSABLE MEMORY PROJECT

5. FUNDING NUMBERS
C - F30602-88-D-0027
PE - 62301E
PR - F403
TA - 01
WU - P2

6. AUTHOR(S)
Saul Y. Levy, J. Storrs Hall, Donald E. Smith

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rutgers University
Lab for Computer Science Research
New Brunswick NJ 08902

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
1400 Wilson Blvd
Arlington VA 22209-2308

Rome Air Development Center (COTD)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-290

11. SUPPLEMENTARY NOTES
RADC Project Engineer: David M. Gubbins/COTD/(315) 330-2158
The Prime Contractor for this effort was: Syracuse University, Skytop Ofc Bldg, Syracuse NY 13244.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Content Addressable Memory Project consists of the development of several experimental software systems on an AMT Distributed Array Processor (DAP) for evaluation and research purposes. The resulting determinations of the strengths and weaknesses of the DAP are discussed. Two new machine architectures were developed in an attempt to consolidate the strengths and overcome the weaknesses. The effort led to the analysis of certain parallel algorithms from a new point of view, and thus to a new class of parallel algorithms, the semiserial algorithms.

14. SUBJECT TERMS
Content Addressable Memory, Parallel Architecture

15. NUMBER OF PAGES
120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Content Addressable Memory Project

Annual Report for 1989

Table of Contents

Overview

We have explored massively parallel architectures based on the concepts of Content Addressable Memory. We took as a point of departure a virtual machine architecture which we will refer to as the CAM.

Our research objective was twofold:

- Develop a single-chip massively parallel processor with a suitable architecture as a vehicle for these applications. Experiment with the relationship between the algorithms and the architecture, and with various tradeoffs in the implementation.
- Implement a programming environment using algorithms from a variety of different fields. We did not require anything beyond the current state of the art in any of these fields, but it would be significant to bring the existing results together in an integrated environment under a single parallel computational paradigm.

History

Involved in the CAM project are Professor Saul Levy, P.I.; Josh Hall, research scientist, Don Smith, research assistant professor, Miles Murdocca (AT&T Bell Labs) and three graduate assistants. Previously, primarily under the auspices of AFOSR, we had developed the CAM architecture and a low-level programming language, CAML, primarily with an eye to implementation as an optical computer.

The CAM model is similar to a conventional machine. In its essence it is simply a processor and a memory. However, where in a conventional machine the processor specifies loading and storing in memory by a single address at a time, the CAM processor can specify conditions, ranges, and collective functions.

The CAM memory consists of a set of words which are addressed either conventionally or associatively. The conventional addressing is extended to allow specifying ranges. Where in a conventional machine one can specify, "*word 4256*", in the CAM we can specify "*words 4201 through 5557 inclusive*", or, addressing the words by their contents: "*all words containing a number greater than 17*".

If we can specify a selected subset of the words in memory at once, we can do something to them at once. For example, we can set them all to a new value (it must be the same value), or add something to each one (it must be the same something). We can also discover the address (instantly) of the first word which we have specified in some way, ie, "*give the address of the first word whose contents are greater than 17 and less than 39.*"

Furthermore we can divide the memory into sections and use a different value in each section: "Add 20 to all words in section 1, 52 to all words in section 2, etc." The sizes of the sections are restricted; they must correspond with the tree structure (see below). Furthermore, the value to be used with each section must come from a word in that section.

The collective functionality reflects a hardwired tree of simple ALU's terminating in the processing elements (words). It can perform certain functions (sum, maximum, etc) of the active words by direct configuration.

Up to the beginning of 1989 we had accomplished the following with the CAM architecture:

- We had developed techniques for digital design which allow us to embed arbitrary circuits into the regular interconnection patterns required by the free-space regime.
- We had developed a class of computer architectures that take advantage of the massive parallelism implicit in the optical switching technology.
- We had developed a programming model and methodology to be able to use our architectures effectively.

We had produced a preliminary design of a CAM machine for optical technology. Fragments of the design were being implemented and tested (in multiple quantum well SEED technology) at Bell Labs. (Note that the recently announced optical processor from Bell Labs was built by the very group we were collaborating with. Although much too small to use any actual CAM architecture, the processor was designed using some of the techniques referred to above.)

To prove the general usefulness of the CAM, we had investigated algorithms from many areas of computer science, including numerical analysis, combinatorial optimzation, systems programming, programming language compilers and optimizers, and artificial intelligence. Not only had we found the CAM model to be congenial to all of these areas, but the rising interest in massive parallelism has engendered such widespread results that the question of general usefulness is well on its way to having been put to rest.

We had produced good results in non-numeric programming fields such as graph theory, network optimization and computational geometry. The set operations basic to these algorithms are near-primitives on the CAM. We had algorithms for:

- Min, max, member, insert, delete, intersection, union, find.
- Spanning trees, shortest path, connectivity, depth and breadth-first search, transitive closure, etc.
- Convex hull, 3-D convex hull, Delaunay triangulation, line and curve fitting.
- Sorting and searching (although sorting is usually rendered unnecessary and searching is usually a simple primitive operation).
- Means, medians, modes, histogramming, generation of permutations, binomial coefficients, random numbers, and so forth.

A New Emphasis

At the beginning of 1989, DARPA procured for us a DAP 510 massively parallel processor to be used in architectural experimentation. The DAP proved to be something of a challenge, and we were forced to change our focus from the more theoretical approach we had been following, to a practical, experimental regime.

Our progress reports give a flavor of the sequence of events:

[1Q] "We have begun to develop the ParaSol architecture, based on the mesh-of-trees concept. We are giving a paper on the subject next week in the Massively Parallel Processing Symposium at U. of S. Carolina. We have received the DAP supplied us by RADC/Syracuse and have succeeded in bringing it up to a usable state. This required significant effort on our part since AMT software is a few releases behind the current release of the host Sun software, in use on our existing systems. We are now doing extensive benchmarking and architectural analysis."

The ParaSol (the paper is included in this report) was an extension in which the "highways" of the DAP were developed into trees so that collective functionality (which is present in the DAP only in very rudimentary form) could be exercised in orthogonal dimensions.

[1Q] "We appear to be the only site in the US trying to use a DAP without FORTRAN. This unique situation caused us some additional headaches in addition to the above. AMT was helpful, to the extent of giving us FORTRAN free; however, we believe we now know more about DAP software than AMT's service techs. We now have a working system, and can indeed program it solely in assembler. We are implementing a variety of algorithms, to determine "the edges of its envelope"; we have implemented line drawing algorithms, for example, that are significantly faster than the supplied graphics library."

Specificlly, the AMT software techs had originally shipped us the FORTRAN compiler and the assembler, but not their common preprocessor, under the impression that the preprocessor was the FORTRAN compiler. Then when they "gave" us "FORTRAN" the only new file was the preprocessor, which we simply used to run the assembler.

[2Q] "We have developed the ParaSol architecture and presented it at the Parallel Processing Symposium at Univ. of South Carolina. This is a generalization of the structure of content addressable memory which is particularly useful for numerically-based recognition algorithms such as neural networks and hidden Markov models. We are also, out of pure frustration, designing a rational version of the DAP architecture."

ParaSol is a great architecture for the specific mentioned problems, but by that time we were tangling heavily with the problems of implementing language control structure and general-purpose code on the DAP. We realized that many of the architectural blunders of the DAP we had blissfully copied right over into the ParaSol. This "rational DAP," of course, ultimately developed into the Short Stack.

[2Q] "We have designed a higher-level language for the CAM model that expresses the associative and parallel operations of the machine naturally within the rigorous framework of sparse array theory. We have implemented on the DAP a suite of primitive functions which is to form the core of an implementation for the language, and have attained a fair amount of expertise in systems-level details of its operation. We are also attempting to port an implementation of C to the DAP."

"Primitive" here means things like subroutine linkage convention– not terribly glamorous but it must be efficient if there is to be any hope of an efficient machine. This suite of functions ultimately formed the core of our C implementation.

[3Q] "A half-year's experience programming the DAP has exposed the major weakness of its architectural type for general-purpose use (the DAP is sold as a special-purpose machine). This weakness is a decoupling of the control flow from the actual parallel processing. We have been developing a small-scale architecture, the SHORT STACK, which addresses this weakness and also address software portability issues, which we have heretofore paid insufficient attention to."

A simple example: the DAP has a 1024-bit-wide bandwidth to memory, but when the registers are to be saved (as in a subroutine call) they must be stored one at a time, using three cycles per store. The reason is a bottleneck between the register file and the array plane. In the Short Stack the register file is part of the array plane, so that all the registers can be saved or restored at once. This enhances the ability to port ordinary serial software to the Short Stack; any parallelism available in the program may be taken advantage of without having to split the data involved off into a separate parallel engine.

[3Q] "All our software effort for the period has been spent on our DAP C compiler. We started with an (allegedly) retargetable C compiler for conventional machines and have been modifying it to output DAP assembly language. Our present status is a compiler that can handle some small programs but it is all too easy to uncover bugs. The machine model of the DAP differs just enough from the conventional model to cause problems without any compensating advantages. Ergo SHORT STACK above."

Silly things, like being unable to use a negative offset with an index register. More important things, like being unable to modify the size of a process once it had been created, due to an archaic address-mapping mechanism.

Results

We have learned a lot from our experience with the DAP, and it has all gone into the design of the Short Stack. The rest of this report deals in detail with the systems we directly implemented on the DAP. The development of the systems was slow and piecemeal, since we would constantly be taking them apart and analyzing all the low-level algorithms, and trying other techniques, and so forth. This was, after all, what we were here for; to analyze the machine from the point of view of the low-level software.

It turned out, as isn't uncommon in basic research, that there was a better way than the approach we had proposed at the outset. There were many mistakes to be made in architectural tradeoffs we did not see as the year began. Among these is the concept of the single-bit processing elements. We can show now that there are much more cost-effective ways of using the same hardware. This has cost us a year of work on a frustrating machine, but we consider the knowledge gained cheap at the price.

Bibliography

Advanced Micro Devices: *AM99C10 Content Addressable Memory: Device Description*, AMD, Sunnyvale CA 1988

Bacha, Hamid: *Beyond the WAM: A PAM for the CAM*, CASE Center TR-8816 Syracuse University 1989

Batcher, K. E.: *The Multidimensional Access Memory in STARAN*, IEEE Trans. Comput., c-26 (1977), pp 174-177.

Blair, Gerard M.: *A Content Addressable Memory with a Fault-Tolerance Mechanism*, IEEE JSSC, vol SC-22, no. 4, pp 614-616, Aug 1987

Blelloch, Gary: *Scans as Primitive Parallel Operations*, pp 355-362, Proceedings of the 15th International Conference on Parallel Processing, Pennsylvania State University Press, University Park, 1987

Foster, Caxton C.: **Content Addressable Parallel Processors** Van Nostrand Reinhold, New York, 1976

Fountain, Terry: **Processor Arrays: Architecture and Applications**, Academic Press, London, 1987

Murdocca, M. J., Hall, J. S., Levy, S. Y, and Smith, D: *Proposal for an Optical Content Addressable Memory*, Optical Society of America Topical Meeting on Optical Computing, Technical Digest Series 1989 (to appear)

Hall, J.S. & S. Y. Levy: *Von Neumanizing the Multi-Search Content Addressable Memory*, The Fifth Annual CSCI Symposium on Massively Parallel Processing, University of South Carolina, 1989

Hall, J. S., Levy, S. Y, and Murdocca, M. J.: *Design Techniques for an Optical Connection Machine*, Proc. AIAA Computers in Aerospace IV, Wakefield Mass., October 1987

Hall, J. Storrs: *System Design and Programming Methodology for a CAM-based General-purpose Computer*, LCSR-TR-16, Rutgers University 1982

Hall, J. S. and Dunn, M.: **CAML Programming Language User's Manual**, Department of Computer Science, Rutgers University 1987

Hall, J. Storrs: *A General-Purpose CAM-based System* in **VLSI Systems and Computations** H. T. Kung, Bob Sproull, and Guy Steele Computer Science Press, Rockville MD 1981

Hillis, W. Daniel: **The Connection Machine** MIT Press, Cambridge, 1985

Hillis, D. and G. L. Steele, *Data Parallel Algorithms*, Communications of the ACM, December 1986

Ilgen, Sener and Isaac D. Scherson: *Parallel Processing on VLSI Associative Memory*, pp 50-53, Proceedings of the 15th International Conference on Parallel Processing, Pennsylvania State University Press, University Park, 1987

Kohonen, Teuvo: **Content-Addressable Memories** Springer-Verlag, Berlin, 1980

Koo, J. T.: *Integrated Circuit CAM*, pp 208-215, IEEE J. Solid State Circuits, SC5, 1970

Kruskal, Clyde P, Larry Rudolph, and Marc Snir: *The Power of Parallel Prefix*, pp 180-184, Proceedings of the 12th International Conference on Parallel Processing, Pennsylvania State University Press, University Park, 1984

Langè, R G: *High Level Language for Associative and Parallel Computation with Staran*, in **Proceedings of the 1976 International Conference on Parallel Processing**

Levy, Saul Y: *A System for Handling Task Swapping in a Multiprogrammed, Multiprocessor Environment.* RCA Labs Internal Report, 1968

Levy, Saul Y: U.S. Patent 3470540: Multiprocessor Computer System with Special Instruction Sequencing.

Murdocca, M. J. and N. Streibl, *A Digital Design Technique for Optical Computing*, Optical Society of America Topical Meeting on Optical Computing, Technical Digest Series 1987, 11, pp 9-11.

Murdocca, M. J., A. Huang, J. Jahns, and N. Streibl: *Optical Design of Programmable Logic Arrays*, Applied Optics, 27, (May 1, 1988)

Scherson, Isaac and Smil Ruhman: *Multi-operand Arithmetic in a Partitioned Associative Architecture*, Journal of Parallel and Distributed Computing 5, (1988) pp 655-668.

Stolfo, Salvatore J. and Daniel P. Miranker: *DADO: A Tree-Structured Architecture for Artificial Intelligence Computation*, pp 1-18, Annual Review of Computer Science, Annual Reviews, Palo Alto, 1986

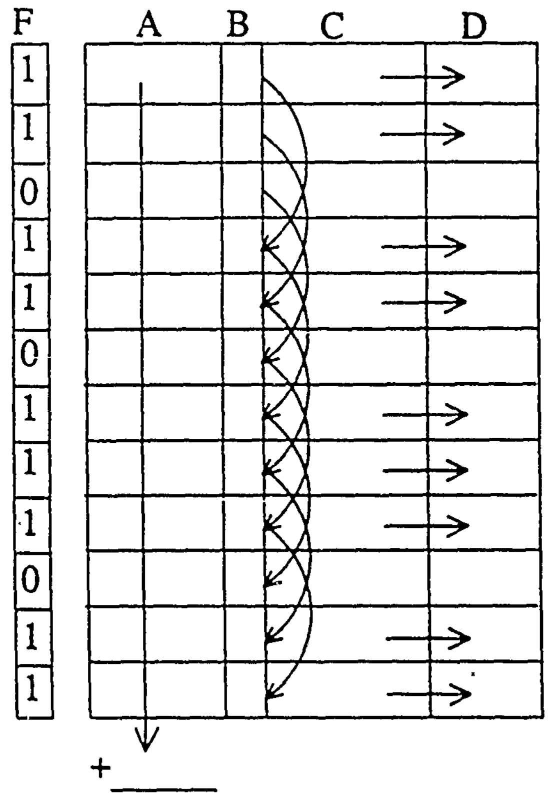
Thurber, K. J.: **Large Scale Computer Architecture** Hayden, Rochelle Pk, NJ, 1976

Thurber, Kenneth J and Peter C Patton: **Data Structures and Computer Architecture: Design Issues at the Hardware/Software Interface** Lexington Books, Lexington, Mass., 1977

Potter, J. L. ed: **The Massively Parallel Processor**, MIT Press, Cambridge, 1985

Wexelblat, R. (private communications) Wexelblat implemented a proprietary (ITT-ATC) object recognition system which had teachability and partial pattern recognition capability similar to a neural network model but was based on the collection and recombination of well-defined statistical parameters of the domain.

The CAM virtual machine model



Data structure: arrays of records. Records are separated into fields.

Parallel operations are supported between fields, i.e. add C to D in each record.

Activity control (F) allows data-dependent "skipping" of records.

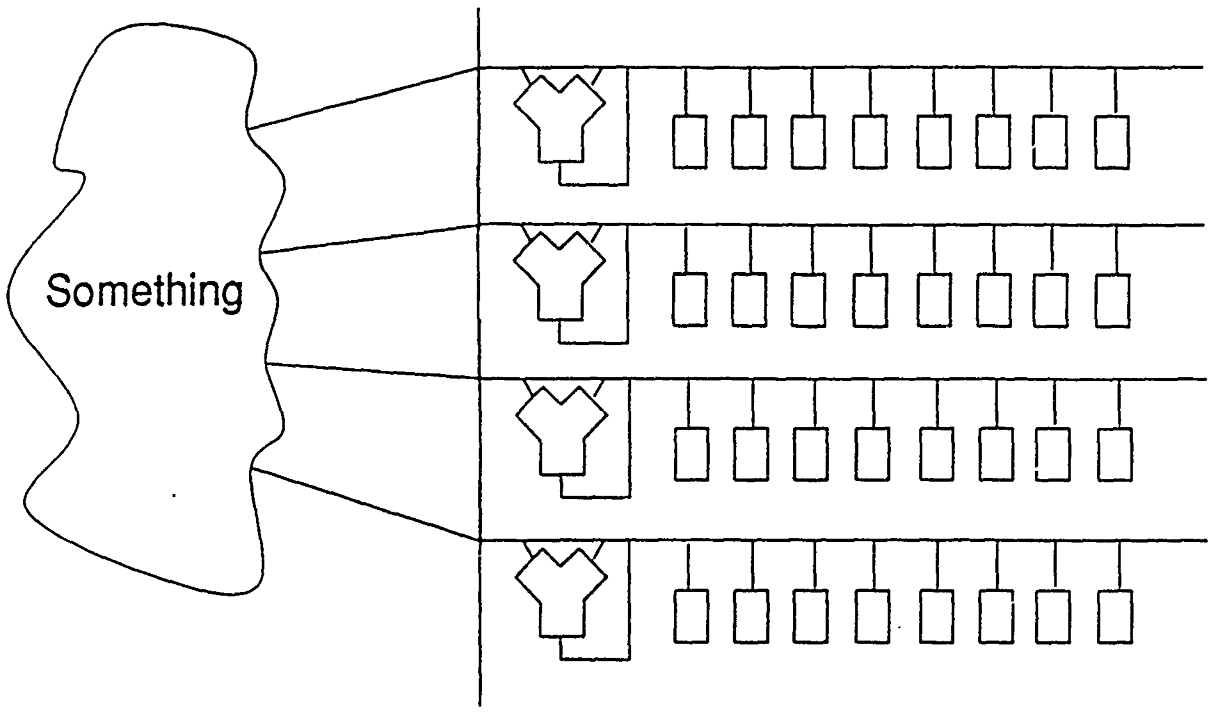
Collective functions such as sum (A), max, scan, etc, are supported for the values of a given field in all records.

Data motion (B), at least shifting.

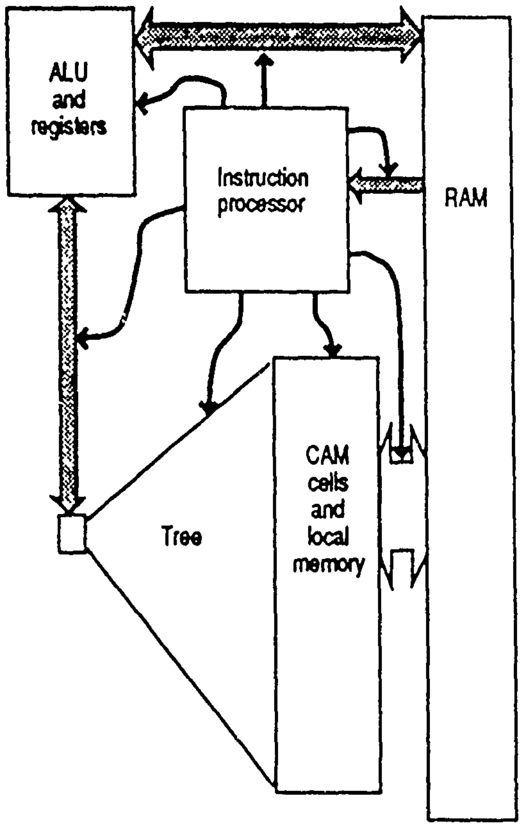
Parallel application of collective functions to subianges.

This model is sufficient to parallelize a very wide range of computing tasks--so wide that it can be considered a completely general-purpose paradigm.

Implementations of CAM



Machine	"Something"
STARAN	Flip Network
DAP, MPP	square mesh
Conn. Mach.	hypercube
U.Mass IUA	pyramid
Rutgers	
CAM	tree
ParaSol	mesh of trees



CAML

CAML is a simple, elegant language designed to express algorithms for the CAM machine model.

We implemented a CAML compiler and wrote CAM versions of algorithms from many areas of computer science.

(includes: set theory operation primitives
typical graph operations
network optimization
computational geometry
statistical and numerical codes
sorting and searching (database)
compiler algorithms
memory management
other systems software)

Linear C is an unlovely hybrid language which imports the CAM programming model into a fromework that will support unmodified existing C code and allow for moving to parallel algorithms incrementally statement by statement.

Linear C

Expresses the same parallel algorithms as CAML (Ie, reflects the CAM machine model).

A superset of C

- Same source code efficiency as C
- Programs can be ported incrementally
- Datastructures exactly the same
  - Identical declarations
  - The "substitutable imperative"
  - Arrays vs Vectors
    - Array: a structure of storage
    - Vector: a structure of operations

Plans for Future Work

Instruction Set for Short Stack

Linear C compiler (to simulate)

Register transfer level simulator
for the Short Stack

Linear C compiler (for real
instruction set).

Higher level language definition

The Short Stack

A SIMD processor architecture

As mentioned elsewhere, the Short Stack was originally conceived as as a rationalization of the sort of cellular array processor represented by the NASA MPP, ICL DAP, ITT CAP, etc. The major impetus was the observation that the array-to-memory bandwidth represented a resource that was being almost entirely wasted during many common system functions, such as subroutine linkage.

The classical method of dealing with this problem is to decouple the system functions even further, in the hopes that the overhead operations can be overlapped with array manipulations. The Connection Machine, for example, follows this strategy.

The Short Stack is the concept of a machine built to test the opposite hypothesis: that by integrating the control functions even more closely to the array, the processing power, and more importantly perhaps, the memory bandwidth of the array could be brought to bear on the system functions.

This would have the major advantage that the necessity of tearing a program apart into "parallel" and "serial" portions would be avoided, and thus porting programs onto the architecture would be easier. Another big advantage would simply be the speed gained in the system functions themselves, assuming that the bandwidth could be harnessed appropriately. That would allow efficient executions of some paradigms which are completely lost on the decoupled style of array processor. For example, Prolog execution consists almost entirely of its form of subroutine linkage.

Architecture

The DAP consists of an array, generally thought of as a square matrix of single-bit processors, and a more-or-less conventional set of registers upon which serial instructions are executed. The departure represented by the Short Stack is to identify rows of the array with registers. Thus the entire register set can be saved or restored in a single memory cycle, greatly simplifying and speeding up operations like subroutine calling.

The DAP also has a mode of operation in which the rows of the array are operated as words, although it isn't strongly supported–word-mode operations are considerably slower than the bit-mode ones. Even so, we found word-mode operation to be superior to bit-serial for several algorithms, and this prompted us to investigate abandoning bit-serial mode altogether. We now believe this to be a good idea and the Short Stack architecture has no bit serial modes at all. Among other things, this obviates the conversion between different formats for scalar and vector data.

We therefore refer to the Short Stack as a "word-parallel" machine as distinguished from a bit-serial one. This leaves a 1k-pin memory bandwith representing 32 32-bit words instead of 1k 1-bit ones, but still able to process 1k 32-bit words in 32 cycles, same as the bit-serial. The tradeoffs are complementary: the word-parallel machine handles shorter arrays of 32-bit data better, the bit serial handles 1k arrays of shorter data better. System functions consist almost entirely of the former type of operation.

Collective Functions

The feature of our original CAM virtual machine architecture that gave it its algorithmic flexibility was the set of collective functions that was to be implemented directly in the hardware, and those are similarly a feature of the Short Stack. The collective functions are such operations as "sum all the words," which produce a result that depends on the value of each word.

The Short Stack has the collective functions as register set to register set operations, implemented to be efficiently chainable for operations over longer vectors. (The parallel scalar operations are similar.) This allows both the processing of long, memory- resident arrays, and the optimization of short-array algorithms designed to use the Short Stack memory bandwidth to best advantage.

An example is simply searching for a key value in a memory-resident data structure. The collective function that might match this problem is "find first value greater than or equal to". We can compare algorithms for this task on serial, word parallel, and bit serial architectures. (The serial machine is disadvantaged from the outset by its limited memory bandwidth, and may be viewed as a control; the two parallel alternatives are intended to represent two different ways of arranging the same amount of hardware resources.)

For concreteness, let us assume we are searching a table of a billion entries for keys which are 32-bit quantities. In the serial machine, we implement a binary tree of depth 30, which we can then probe in 30 memory cycles.

In the bit serial machine, we can form a 1024-ary tree that is only 3 levels deep! However, it takes us 32 cycles to read one node, bit serially, so that even if the key-finding in the node is instantaneous, it takes 96 cycles to probe the tree.

In the word-parallel machine, node size is 32 and thus the tree is 6 levels deep. However, node reading is a single-cycle operation, yielding a 6-cycle search.

If, that is, we can find the key position in the 32-value node with a similar alacrity once we have the node in a register set. This is a collective function, and it shows the necessity of being able to do that function fast. We could search the node itself by a binary search, but that would take 5 probes, reducing the whole algorithm back to the serial machine performance!

A similar proviso applies to the case of the use of the word- parallel machine for applications like Prolog or Ops-5. The advantage gained over the serial machine is likely to be no more than 10, as far as indexing and searching are concerned, and thus it is critically important to avoid losing the cycles back to collective functions.

Instruction Set

The Short Stack takes an advantage of SIMD in having a much smaller instruction than data bandwidth. Thus instructions can be kept in the same memory as data with only a minor impact on overall speed. Even running straight-line code with no loops, the 1k-bit machine (the smallest we envision) would get one plane with 32 words of instructions at a time, and not need another for many cycles.

The instruction format has not been fixed yet. Present indications are that a long-instruction-word format, able to choreograph the several major functionalities of the machine, is desireable. At the very least, memory read/writes, parallel scalar register-to-register operations, and collective functions could be overlapped and/or pipelined.

Data Format

The DAP, we opine, went too far in the direction of trying to squeeze the last bit of efficiency out of variable precision data. The Short Stack limits this to the same 8-, 32-, and 64-bit objects as a conventional processor. In order to avoid the problems of conversion that plague the DAP, we lay the memory addressing out in a pattern that allows operations between vectors of different sized objects without any inter-processor communication. This is a simple interleaving scheme but must be hardware-supported (by local barrel shifters in the memory data path) if the speed advantage is to be retained.

Semi-Serial Algorithms

The major advantage of the Short Stack architecture for systems code and "general purpose" use is that its most efficient modes correspond to the most commonly performed functions. However, there is also a particular class of algorithms where the word-parallel architecture has a theoretical advantage over the bit-serial one. We call these the "semi-serial algorithms."

A semi-serial algorithm is one in which a vector of results can be calculated in parallel using one operation (say, multiplication), or calculated serially using a less expensive operation (say, addition). This is exactly the class of algorithms that is susceptible to the well-known optimization of "reduction in strength."

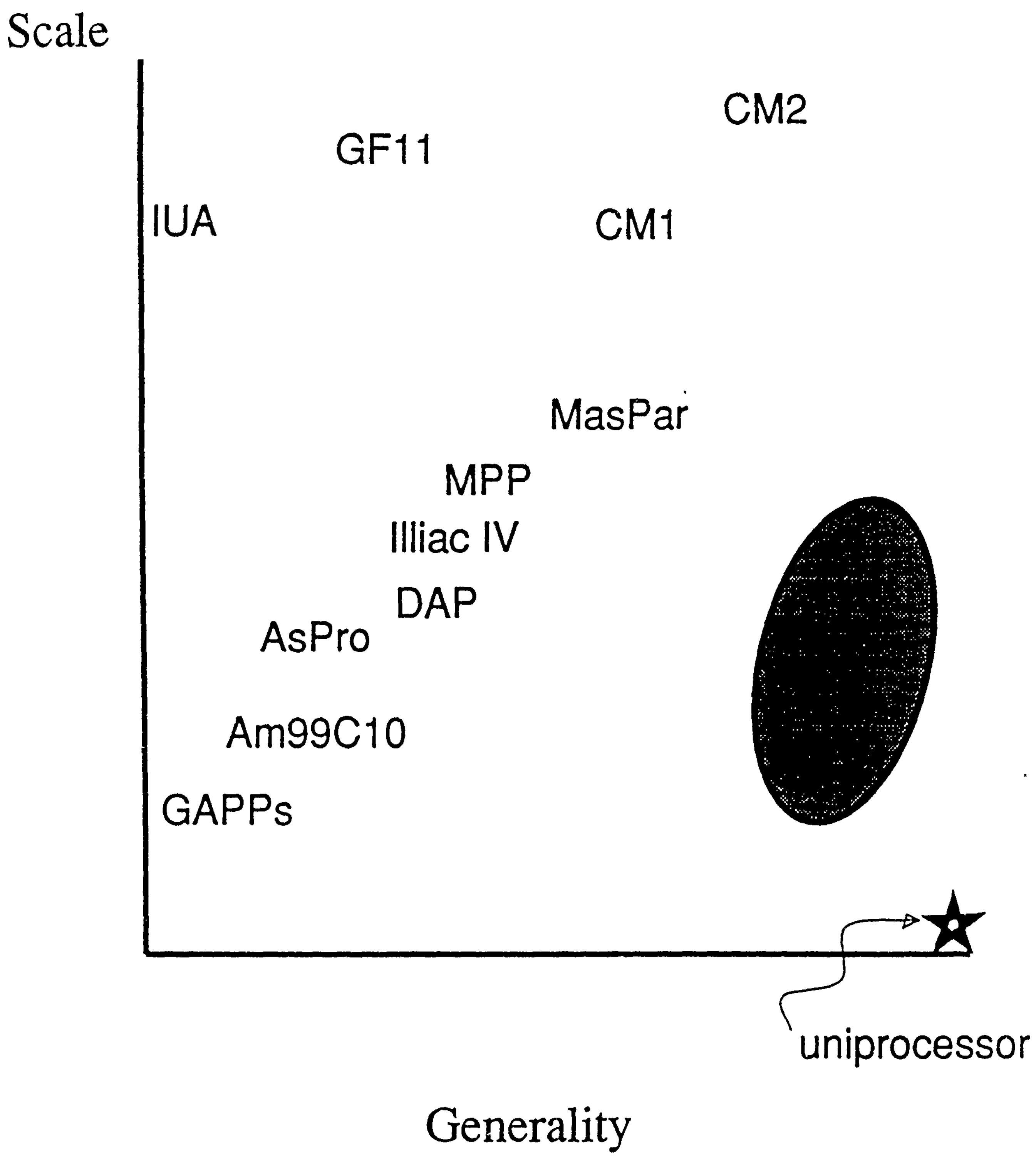
A simple case suffices for an example: consider calculating 1k values in an arithmetic progression. The serial algorithm is to add the difference value again and again, and the parallel algorithm is to multiply that value by 0, 1, 2, ..., 1023. Now consider a 1k-bit-wide machine: if it is arranged as 1k bit-serial processors operating in parallel, each will have to do a multiplication, a 1k-cycle operation if the data are of 32-bit precision!

A word-parallel arrangement of the same hardware would do instead 32 32-bit adds in one cycle, and 32 32-bit mults in 32 cycles. (Please note that the raw operations-per-cycle of the machines is the same: 1k adds in 32 cycles, 1k mults in 1k cycles.) However, the word-parallel machine can calculate 32 values in parallel, taking 32 cycles, and then follow with 31 cycles of 32 additions each, using the serial formulation, for a total of 63 cycles.

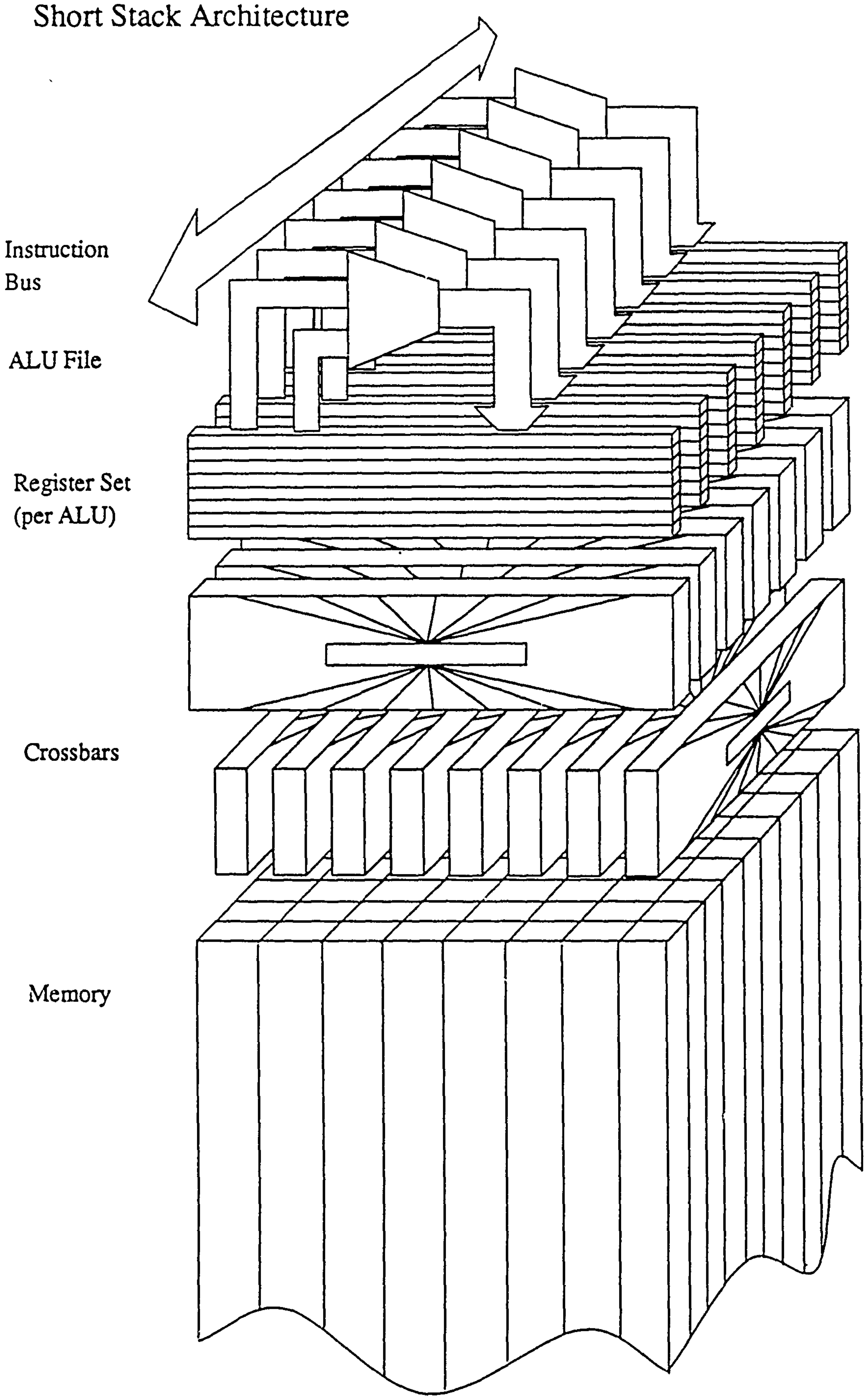
There are many other pairs of operations which fall into the same category,[1] so that similar results can be expected in set- and graph-based algorithms, and other non-numerical applications.

[1] See Paige, Robert: **Formal Differentiation: A Program Synthesis Technique**, UMI Press, 1981

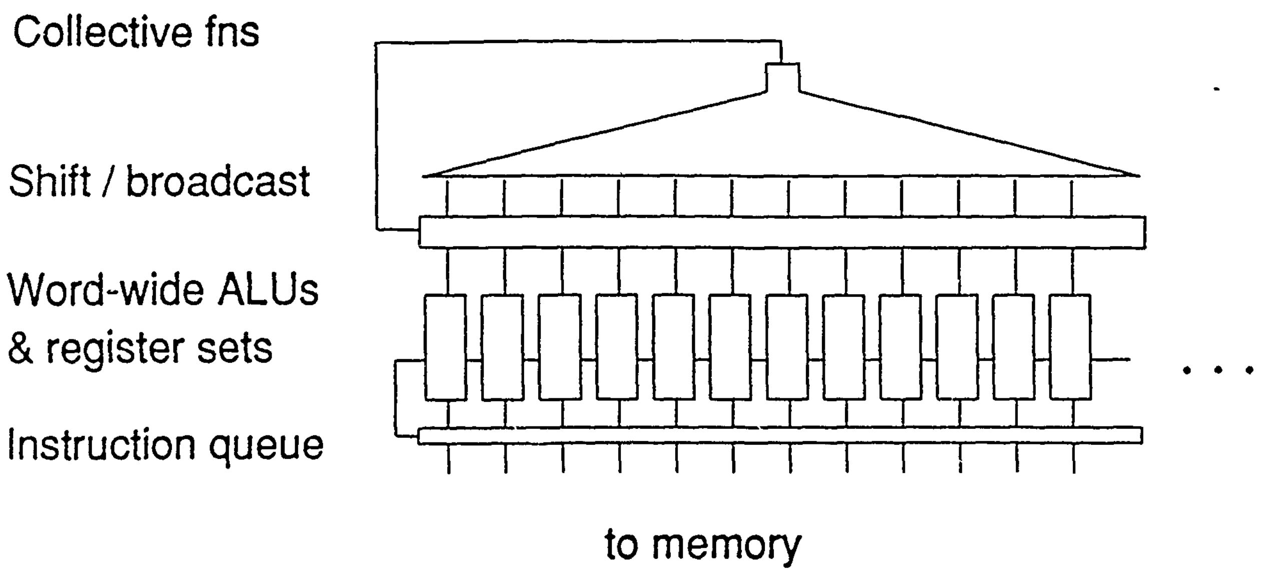
Associative Parallel Processors



Short Stack Architecture



Short Stack Architecture



Word-parallel -- no separate controller

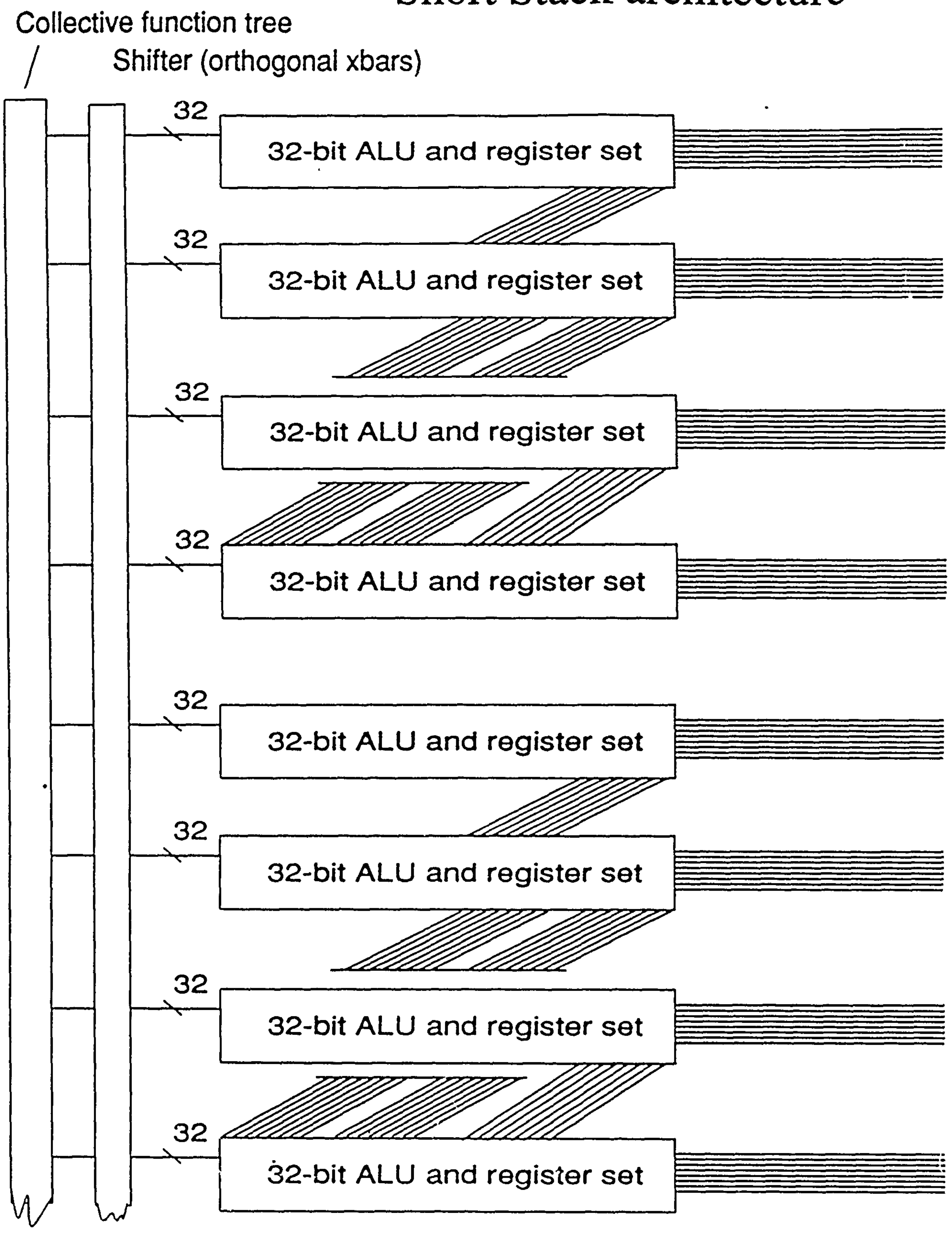
Allows parallel operations on scalar data.

Increases speed on algorithms with
low parallelization.

Allows incremental parallelization.

Enhances associative control flow operations
(Prolog, OPS5, object-oriented languages).

Short Stack architecture


Collective function tree
Shifter (orthogonal xbars)
32
32-bit ALU and register set
32
32-bit ALU and register set
32
32-bit ALU and register set
32
32-bit ALU and register set
32
32-bit ALU and register set
32
32-bit ALU and register set
32
32-bit ALU and register set
32
32-bit ALU and register set

. . . 128 in all

Semi-serial Algorithms

Three architectures, each with the equivalent circuitry of 1024 full adders:

A) 1024 bit-serial one-bit ALUs
cycles to add: 32
cycles to multiply: 1024

B) 32 word-parallel 32-bit ALUs
cycles to add: 1
cycles to multiply: 32

C) 1 full 32x32 fast multiplier
cycles to add: 1
cycles to multiply: 1

Problem: compute some parallelizeable but strength-reducible function for 1024 points.

E.g. $Y = mx + b$ for $x = 1,2,3, \ldots, 1024$.

Machine	mults	@	adds	@	cycles
A	1	1024	0	32	1024
B	1	32	31	1	63
C	1	1	1023	1	1024

Linear C: A C superset for SIMD parallel processing

J. Storrs Hall
Laboratory for Computer Science Research[1]
Rutgers University

Abstract

Linear C is a C language extension for incremental portability of C programs to a parallel machine. Other parallel extensions to C require redesign of algorithm and data structures to take advantage of parallel operations. Linear C is designed to obviate this necessity.

[1] Supported by Air Force Office of Scientific Research under AFOSR-86-0294 and DARPA under RADC t.n. B-8-3615 and B-8-3616.

The Substitutable Imperative

Linear C is a language designed to allow the incremental extension of C programs to a parallel machine. There are many parallel extensions to C, but all of the ones we are aware of require the programmer to redesign the algorithm and data structures of his program to take advantage of the parallel operations. In Linear C, on the other hand, the translation to parallel form can be done *incrementally*, that is, one statement at a time. There are several benefits to this approach:

a) There is no high initial investment in reprogramming necessary.
b) Only the most critical and/or easiest parts of the application need be converted at all.
c) Since system functions and libraries using parallelism interact seamlessly with serial code, substantial advantage can be obtained with no translation whatsoever.

(It should be mentioned that Linear C is a language designed with a specific (SIMD) parallel architecture in mind, namely one which allows such a tight coupling between serial and parallel code. This is not true of many existing parallel architectures, even the SIMD ones.)

The principle behind the design is what we call "the substitutable imperative". That is to say the declarations of a Linear C program are exactly the same as C; the data structures are identical. For any given Linear C program there is an equivalent C program, statement for statement; the state of the data structures between corresponding pairs of statements is identical. To put it another way, you could substitute at random corresponding statements of the two programs and get the same results. (This is called the "substitutable imperative" since it only makes sense for the imperative statements, the declarations already being the same.)

Vectors vs Arrays

To understand the parallel extensions that constitute Linear C, one must realize that the array as defined in C is not a "first class citizen". If `A` is the name of an array in C, it actually denotes the address of the array, i.e., a pointer. `A+1` means the address of the second element, not any operation on the array itself; and `A=1` is illegal.

In Linear C we introduce a new collective type that is the opposite of the array. The array cannot take part in arithmetic operations: the vector can. The array can be stored into and is non-volatile: the vector cannot and is ephemeral.

With the restrictions imposed by the "substitutable imperative", it was impossible to perform some of the elegant generalizations of the other C extensions. However, the syntax and semantics of C are relatively ad hoc to begin with, and the Linear C extensions are quite consistent with its spirit. Furthermore, they are few in number. For example, the following implementation of `strcpy()` (string copy) in Linear C, while cryptic–

```
        b[#@!*#a] = *#a;
```

–is not particularly more so than the "standard" code from Kernighan and Ritchie:

```
while (*b++ = *a++);
```

(Although the experienced C programmer will of course recognize the latter as a cliché of C programming.)

Vectors

A vector is a sequence of C objects that is to be operated on in parallel. The programming model we are using is essentially a machine with a vector register and perfectly ordinary memory, whose contents are described by perfectly ordinary C data structures. However, while a statement is being executed, the vector register(s) may be operating on a sequence of values in parallel, although the results must be placed back into the ordinary C arrays at the end of the statement.

vector: A sequence of *values* which takes part in arithmetic operations as argument or result

array: A sequence of *storage locations* into which values may be stored between operations

The simplest way to specify a vector is simply to list the elements so:

```
a,,b,,c,,d ...
```

(Double commas are used because single comma is already an operator in C.) The items can be scalars (ordinary C values) or vectors themselves, in which case they are concatenated together.

However, the basic vector form is

```
#i
```

In this basic form, `i` must be of an integer type, and `#i` is interpreted as a *vector* of `i` successive integer values starting at 0. Thus the value of `#6` is

```
0,,1,,2,,3,,4,,5
```

The most commonly used form of vectors, however, is a binary operator:

```
a#b
```

which simply means `a + #b` (if `b` is an int). Thus `a#b` equals

```
a,,a+1,,a+2,, ... ,,a+b-1
```

If `a` and `b` are ints, this is a vector of integer values; but if `a` is a pointer (such as the name of an array) this is a vector of *pointers* to the first `b` elements of the array, by standard C pointer arithmetic.

Mixed Operations

Whenever a standard C operation is done between a scalar and a vector, the scalar is broadcast to be operated on in parallel with each element of the vector. Thus,

```
4 + (5#6)   ==    4 + (5,,6,,7,,8,,9,,10)
            ==    9,,10,,11,,12,,13,,14
```

If a unary C operator is applied to a vector, the value is a vector of the operator applied to each element of the original vector. Thus, remembering that a#b is a vector of pointers to the b first elements of a, *(a#b) is a vector of the first b elements of a. Remembering that "*(a+b)" in C is identical to a[b],

```
*(a#3)     ==    *(a ,, a+1 ,, a+2)
           ==    *a ,, *(a+1) ,, *(a+2)
           ==    a[0],,a[1],,a[2]
```

If a standard C operator is applied to two vectors, they must be of the same length, in which case the operation will be done elementwise, or one or both of them must have indeterminate length. If one is indeterminate, the length of the other will be used; if both are, the result is also of indeterminate length.

The semantics of # used as a unary operator depend on the type of its argument. If i is an int, #i <==> 0#i. If a is a declared array with length 1, #a <==> a#1, ie, a vector of pointers to the elements of the array:

```
 ...
int a[3];
 ...
     #a      ==     a#3
             ==     a ,, a+1 ,, a+2
             ==     &a[0],,&a[1],,&a[2]
```

If a is a pointer declared with * instead of [], the length of the vector is indeterminate, and must be inherited from another part of the vector expression. (Note: *#a can be seen to denote the elements of array a as a vector:

```
 ...
int a[3];
 ...
     *#a      ==     *(a#3)
              ==     *(a ,, a+1 ,, a+2)
              ==     *a ,, *(a+1) ,, *(a+2)
              ==     a[0],,a[1],,a[2]
```

This is a very common usage.)

Assignment

In C, the left-hand side of an assignment must be a valid argument to the & operator. You cannot say 3 = x or a = x (if a is an array). Neither can you say 3#5 = x or a#5 = x. You can say *(a#5) = x, the same way you can say *(a+5) = x. In the first case, if x is a scalar, each of the first 5 elements of a is set to x. If x is a vector (a vector expression, there are no vector variables) then it had better be of length 5 (or indeterminate) and the assignment is elementwise.

```
<a is an array containing 1 2 3 5 7 11 13 17>
     *(a+2) = 55;
<a now contains 1 2 55 5 7 11 13 17>
     *(a#4) = 55;
<a now contains 55 55 55 55 7 11 13 17>
     *(a#4) = *(a+4#4);
<a now contains  7 11 13 17 7 11 13 17>
```

This last could also (and would usually) have been written

```
     a[#4] = a[4#4];
```

(It is interesting to note that + and # associate; i.e., (a+b)#c and a+(b#c) have the same value.) The behavior of vector assignment, above, is exactly what you would expect from extrapolating the normal C rules for assignment and the Linear C rules for mixed vector expressions. Such a strict extrapolation causes some operations to be trivialized when you didn't expect it. For example, when you assign a vector into a scalar, all the elements of the vector are assigned into the same location!

Other Linear C Operators

(In the following, v represents some expression evaluating to a vector, and s denotes a scalar. a denotes an array, i an int. + is used to denote any associative C operator (ie, +, *, &, |, ^, &&, ||)):

i /\ v	(vector):	the first i elements of v
i \/ v	(vector):	v with first i elements removed
v // w	(vector):	those elements of w that correspond to nonzero elements of v
??v	(scalar):	the length of v
s ?? v	(scalar):	the number of occurences of s in v
v @ s	(scalar):	the index of the first occurrence of s in v
v @@ s	(vector):	the indices of all occurences of s in v
@v	(scalar):	the index of the first nonzero element of v
@@v	(vector):	the indices of all nonzero elements of v
+/v	(scalar):	the + reduction (e.g. sum) of v
+\v	(vector):	the + scan of v, ie, the sums of the sequence of initial substrings
$v	(vector):	the elements of v in reverse order
i $ v	(vector):	v cyclically shifted i places (to the left)

Vectors distribute properly across "... ? ... : ..." – in particular, this expression allows for "associative" or "content addressable" operations. Similar parallel control structure can be obtained by using && or || on vectors. The "lazy evaluation" semantics of these operators allows an operation (say, with side effects) to be done on some elements of a vector and not on others.

Vector expressions involving modification assignment ("+=" etc.) distribute the same way that = does, but notice that if a vector is assigned into a scalar, the result is no longer a triviality:

```
z += *#a   <==>   z += +/*#a   <==>  z = z+a[0]+a[1]+...
```

Vectors do not normally distribute across the "," operator (it is difficult to see what semantics would be implied!). However, if the "," expression is part of a "? :" expression (or && or ||), the implied "activity control" is propagated into all the subexpressions.

Examples

It is now possible to explain the strcpy() implementation exhibited earlier:

```
b[#@!*#a] = *#a;
```

It is assumed (in both forms of the code) that a and b are pointers into strings of characters which are considered null-terminated. The expression *#a (it occurs twice in the statement) denotes the sequence of values starting where a points (but requiring a length specification to be supplied). The ! is simply C word-negation, @ finds the non-zero negation of the null terminator, and the unary # produces an index string of that length.

This form of the expression copies the string without the null terminator; that could be copied by

```
b[1#@!*#a] = *#a;
```

As a more thoroughgoing example, here is the Dijkstra's algorithm translated from the CAML version we have displayed in the past: (We use "_" to represent a "min" operator.)

```
main ()
{struct {int x,y,t; float a,u;} edge[NEDGES];
 int k;
 /* edge.x and edge.y are set up to represent the graph
    edge.a is the weights */
 *#edge.y==STARTNODE ? *#edge.u=*#edge.t=0 :
                       (*#edge.u = +INF, *#edge.t = 1);
 k = STARTNODE;
 while (||/*#edge.t)
    {*#edge.t && *#edge.x == k ?
        *#edge.u _= *#edge.a + _/(*#edge.y==k)//*#edge.u;
     k = edge.y[*#edge.u @ _/*#edge.t//*#edge.u];
     edge.t[*#edge.y @@ k] = 0;
    }
}
```

Bibliography

FORTRAN-PLUS Language, AMT Ltd. DAP Series Documentation number "man002.03"

Budd, T. **An APL Compiler**, Springer-Verlag, NY, 1988

Hillis, W. D., **The Connection Machine**, MIT Press, Cambridge, 1985

Iverson, K, **A Programming Language**, Wiley, NY, 1962

Kernighan, B., and D. Ritchie, **The C Programming Language**, Prentice Hall, Englewood Cliffs, NJ, 1988

Steele, Guy. L., Jr., "Languages for Massively Parallel Computers", Proc. 2nd Frontiers of Massively Parallel Computation, Oct, 1988, IEEE cat. no. 88CH2649-2

CAML Programmer's Manual

J. Storrs Hall
Laboratory for Computer Science Research
Rutgers University

Introduction

CAML is a systems-level programming language intended for use on machines with content addressable memory. It supports the special features of such machines with a simple and integral data structure model.

The capabilities that CAML assumes do not correspond to a specific architecture, but to a set of related architectures. Thus CAML has an implied specification for a "least common denominator" machine on which it will run. There is considerable latitude in the specification for variation in the implementation of particular features.

Consider "systems programming" languages such as C, Bliss, or Pascal. These languages have a close relation to the architecture and capabilities of conventional processors. Yet between the specific machines in the class thus defined, there can be orders of magnitude of difference in power and throughput. The languages abstract away the inessential detail such as register number, bus width, linkage conventions, and suchlike. Nevertheless they are close enough to the programer's idea of what is "really going on" that they are often called "portable machine languages". CAML is a portable machine language for content addressable machines.

CAM Architecture

The CAM model is similar to a conventional machine. In its essence it is simply a processor and a memory. However, where in a conventional machine the processor specifies loading and storing in memory by a single address at a time, the CAM processor can specify conditions, ranges, and collective functions.

The CAM memory consists of a set of words which are addressed either conventionally or associatively. The conventional addressing is extended to allow specifying ranges. So where in a conventional machine one can specify, "*word 4256*", in the CAM we can specify "*words 4201 through 5557 inclusive*", or, associatively, "*all words containing a number greater than 17*".

If we can specify a selected subset of the words in memory at once, we can do something to them at once. For example, we can set them all to a new value (it must be the same value), or add something to each one (it must be the same something). We can also discover the address (instantly) of the first word which we have specified in some way, ie, "*give the address of the first word whose contents* *are greater than 17 and less than 39."*

As far as actual data motion in the memory, we require only that there be the ability to shift data up and down in one dimension, all words moving the same distance. We do not specify the speed with which this must be done, although obviously the faster the better.

It might be wondered why one would want to build what is essentially a highly parallel processor with such a limited communication structure. The theory which CAML is designed to develop (and to test) is that many useful computations can be implemented on a SIMD architecture where the collective functions available in the communications scheme are simple but fast. The communications available on existing machines is flexible but slow, on the order of milliseconds. Our contention is that a much more limited functionality operating on the order of microseconds has a usefulness as yet unexplored.

This functionality reflects a hardwired tree of simple ALU's terminating in the processing elements (words). It can perform certain functions (sum, maximum, etc) of the active words by direct configuration. It is synchronously controlled, and at no time are "messages" or "packets" sent over it.

Overview of CAML

The basic data construct in CAML, besides scalars of various types, is the array of records. A *pseudovector* is the occurences of some field in a subset of the records of the array. In an array `foo` of records with two fields `foo.a` and `foo.b`, a pseudovector is something like all the `foo.b`'s where `foo.a` is greater than 17. In CAML this would be written `foo[|a>17].b`. Only one array can be the basis of the pseudovectors in any one statement, so the fieldnames are used alone after the initial mention of the array. For example, `foo[2:34 | x>22 & y=z].z := x-17` means "in records 2 through 34, inclusive, of `foo`, in which the x field is greater than 22 and the y field is equal to the z field, place in the z field the value of the x field minus 17." Indexing is zero-based.

CAML allows operations between pseudovectors only where these can be done in parallel on the hardware. Thus, like other systems level languages, program text remains a fairly good indication of efficiency. Primitive functions include those which are implemented at the microcode level, ie arithmetic between fields in parallel, count responders (`\#foo[|x>3]`), address of first responder (`@foo[|x>3]`), etc.

Control constructs in CAML also reflect the capabilities of the CAM. Iteration through a pseudovector, for example, is included since the deaddressing capability allows this to be done in time that depends on the number of items in the pseudovector, but which does not depend on the length of the base array (the original array in which the selected records lie).

CAML Syntax

CAML is defined in terms of functions, statements, and expressions. The expressions are conventional in form; the statements follow a two-dimensional syntax. This syntax is merely a formalization of the indentation commonly used in programming languages to indicate levels of nesting in recursively defined statement constructs.

Formalized Indentation

Consider the following fragments of Common Lisp code:

```
(cond ((null a2) 0) ((numberp a2) (incf n)) ((memq a2 '(o1
o2 o3 o4 o5 o6)) (or (cdr (assq a2 regs)) (cdar (setq regs
(acons a2 (incf n) regs))))) (t (setf (gethash a2 memmap)
(incf n))))
```

and

```
(cond ((null a2) 0)
      ((numberp a2) (incf n))
      ((memq a2 '(o1 o2 o3 o4 o5 o6)))
       (or (cdr (assq a2 regs))
           (cdar (setq regs (acons a2 (incf n) regs))))
      (t (setf (gethash a2 memmap) (incf n)))))
```

The point of interest is that the first fragment is right and the second is wrong–it contains a parenthesis error. Nowadays most programming language code is maintained in display editors which parse it and indent it automatically. But when the code is changed, the indentation can look correct while the actual syntactic nesting is wrong. Often the user forgets to re-indent the code after changing the parentheses.

CAML syntax is an attempt to sidestep the problem: if the indentation looks right, it is right. It is simple to parse the indentation, and since the code is maintained in a display editor anyway, it is as easy to change the indentation as it would be to add parentheses.

A sequence of lines bearing the same indentation is called a *group*. The line just above a group which is less indented is called its *headline*. The headline defines a syntactic unit and all the lines in the group are part of that unit. A line may be a headline for more than one group, if its syntax allows for double nesting. Compound assignment statements are an example of this:

```
1.| array[index].field := expression
2.|             .stream := face
3.|             .meadow := figure
4.|      [preface].field := gloop
5.|               .stream := frog
6.|      [title].meadow := 0
```

Line 1 is the headline for the all the groups. The first group consists of lines 2 and

3. We often refer to the subexpression of the headline at the same level as the other statements in the group, as part of the group. Thus we might say that the part of line 1 beginning "`.field`" is part of the group with lines 2 and 3.

Similarly there is a group at the level of "`[index] ...`", "`[preface] ...`", and "`[title] ...`". Finally, line 4 is the headline for the group consisting of the subline of line 4 starting "`.field ...`" and of line 5.

Comments

Any left parenthesis "(" which is the first non-blank character on a line introduces a comment. The comment continues to the matching right parenthesis ")". A comment may span several lines. No code other than comments may fall on the same line as a comment. The leftmost characters of continued comment lines may fall to the right, but not under or to the left, of the opening paren.

```
(This is a worthless comment which doesn't explain anything.
 It only demonstrates the syntax of comments.)
array[index].field := expression
            (This very important comment is indented.)
            .stream := face
            .meadow := figure
```

Declarations

Declaration statements may occur at the beginnings of environments (such as function bodies). Such a declaration creates an environment containing the variables declared. Declaration types are:

```
env <array>[<length>](type)
    <array>[<length>] .<field>(<type>) .<field>,.<field>(<type>)
    scalar <var>(type) <var>,<var>,<var>(type)
    scalar <array>[<length>](type)
    scalar <array>[<length>] .<field>(<type>)
```

1. An associative array, ie, one that will be stored in cam, is merely named and typed, e.g. `foo[50](int)`.
2. An associative array of records is similar, but the fields are typed individually. Field names are preceded by dots in declarations, e.g. `foo[10] .p,.f(bit) .n(int)` to make them easy to distinguish visually from variable names.

   Arrays may be multidimensional, e.g. `a[5,10]`. This extends both dimensions along the associative memory, i.e. it is as if we had declared `a[50]`. Fields may also be given dimension, e.g. `foo[1000] .hash[50](bit)`. This dimension is orthogonal to the associative memory, and is not associative. (I.e. we could do `foo[| hash[12] & hash[33]]` but something like "`foo[333].hash & foo[666].hash`" is meaningless.

3. Scalar variables rarely need to be declared since this is the default in CAML. The default type is int, but for scalars there is little difference between the integer types (including char, similar to C). Furthermore, some type extrapolation is done, so most undeclared variables will be treated properly.
4. The concept of a scalar array may seem odd, but merely means that the array is held in normal memory and cannot be used in associative operations. Otherwise the declaration is similar to the associative ones.
5. "Scalar" arrays of records are also allowed for completeness.

For example, we might declare a matrix as

```
env a[20,20](float)
```

but for a large and sparse matrix, we need only store the nonzero elements. We could store their indices explicitly, and address them associatively. Thus we might declare

```
env a[500] .i,.j(short) .x(float)
```

Then we could refer to all of row 14 by `a[|j=14].x`.

Types are int, short, char, bit, and float in the preliminary version. Variables and fields need not be typed explicitly throughout the program, but are typed by the compiler at each occurrence. The type of the value associated with a variable *name* may vary throughout the program, but the type at any one *occurence* of that name must be fixed, even though it is only implicit. The same is true for the values of expressions.

In general, the programmer need only specify as much information as necessary; the compiler will supply it if it is obvious from the program. for example, in

```
{foo[1000] .x .y .z}
 foo.x := 3.1415926
    .z := x<y*2.718281828
```

the compiler can easily tell that `foo.x` and `foo.y` are floats and `foo.z` is a bit. It *was* necessary for the programmer to declare them as fields, however, since field-names and scalars are ambiguous in the syntax. If such a name occurs free (ie, without having been declared), it is taken to be a scalar.

A declaration establishes an environment coincident with the group of statements it controls, ie, those just below it indented farther than it is. A more complete explanation of variable environments is to be found in the function and program organization section.

Temporary Declarations

There is a kind of declaration which declares temporary additions to existing arrays. The idea is for there to be a field to be used in some calculation but which need not hang around afterward. The syntax is:

```
env array + .field := optional initialization
```

Suppose we are doing depth-first search in a digraph which has been declared

```
env graph[1000] .from,.to(short)
```

(ie., a list of edges). To do the DFS algorithm we need a "mark bit" which tells if we have seen a vertex yet. We might declare

```
env graph + .visited(bit) := 0
```

and then proceed with the DFS algorithm.

where the initialization may use fieldnames of the original array.

Another kind of temporary declaration changes the dimensionality of an array. This allows a dynamic basis for local collective operations. This declaration does not allocate any space but merely changes the way an existing variable is addressed.

There are two variants:

```
env foo = [21;37;59;hike]
    bar = baz[36;24;36]
```

The first temporarily "rearranges" `foo` to be of the new dimensions (note that variables can be used). The second introduces a new name `bar` to be used with the new dimensions of `baz` so the old name (and dimensions) can still be used.

Operations

```
   factor ::= variable |    constant        | (expression)
     term ::= factor   |  factor * term     | factor / term
      sum ::= term     |  term + sum        | term - sum
     comp ::= sum      |  sum > comp        | sum < comp
                       |  sum = comp        | sum ~= comp
                       |  sum >= comp       | sum <= comp
                       |   ~ comp
condition ::= comp     | comp & condition   | comp V condition
                       | comp X condition
```

These operations, at least in terms of scalars, are quite conventional through the factors, terms, and sums. The comparison operators, however, have an extended interpretation in compound expressions. `a<b<c` means `a<b & b<c` and so forth. This is more natural than most programming language syntaxes and is straightforward–with one possible exception: `a~=b~=c` is interpreted with the same rules, and thus `a` may equal `c` and the expression will still be true.

The comparison operators (including `~`) always produce a bit value. The logical operators `&`, `V`, and `X` may not. They will return one of their operands as a "true" value–`&` returns its right operand if both are true, `V` returns its left if it is true, and if one operand of `X` is true and the other false, the true one is returned.

Assignment is specified by `:=`, and modification assignment is denoted by placing the scalar operation between the `:` and the `=`. Thus `a :+= b` is the same as a := a + b. (We could even write "bit :~=" as a complete statement, meaning complement bit in place.)

Arrays

References to arrays are by index and/or selection. We write *array*[*index*].*field* (scalar-valued), or *array*[*range*|*condition*].*field* (pseudovector-valued). Range is *index*:*index*. Condition is a bit-valued pseudovector compatible with array. If we write *array*.*field*, i.e. leaving out the range and condition altogether, we mean the whole array, i.e. that field in every record.

One base array reference is allowed in any statement. All pseudovectors in the statement must be compatible with it (with the exception of those in scalar-valued expressions). All the pseudovector references in the statement are given merely as fieldnames of fields of the base.

Assignment Statements

Pseudovector assignment statements may specify more than one field compatible with the base. These are specified by specifying a different field name under the first one given in the base, starting with the period, lining up the period. For example,

```
foo[|p].bar := 12
       .baz := 13
```

This means to set the bar field to 12 in those records of foo where p is true; and set the baz field to 13 in the same records. Note that the selection is only done once, before any assignment is done. In:

```
foo[|bar=baz].bar := bar+1
             .baz := baz-1
             .dat := 0
```

the bar, baz, and dat fields are changed in exactly the same set of records, regardless of the fact that bar and baz aren't equal in those records after the first assignment has been made, and the new bars and bazzes may now be equal in other records. Note on the other hand that the assignments are sequential and cumulative:

```
foo.bar := baz
   .baz := bar
```

does *not* exchange the bar and baz fields; the values of bar have been wiped out in the first assignment. (The exchange may be accomplished by the special purpose construct foo.bar :=: baz.)

There may be several clauses to an assignment statement, each with a different base. The array must be the same in each case, and is omitted in all the clauses but the first. These line up vertically below the subscript/selector in the primary clause. This allows for the "catchall" selector "[]" in the final clause which means "any record not selected yet":

```
foo[|x=1].bar := first
         .baz := last
   [|x=2].bar := second
         .baz := penult
   [].bar := general
     .baz := random
```

However, there is more going on than meets the eye. Consider the following assignment statement:

```
env foo[n] .bar,.baz(int) .dat(float) .p(bit)
  foo[0:k | bar>baz].p :&= dat-66 <= bar+baz
                     .dat :+= bar
     [| p].bar := dat-66
```

Foo is the base array for the selection `[0:k | bar>baz]`, and the pseudovector `foo[0:k | bar>baz]` is the base for references to p, dat, etc. But the base for the selection `[| p]` is *all the records not selected by* `[0:k| bar>baz]`. Thus even if several p's got set to 1 in the first clause, none of those records would be affected by the third one. (If we had wanted them to be, we could merely have written

```
foo[| p].bar := dat-66
```

as a completely separate statement.) This feature is known as *exclusive selection.*

We are simulating a chamber filled with moving particles. Each particle has x, y, and z positions and velocities. There is a wall in the middle which represents a potential gradient delta. We want to find particles which have hit the wall in the last iteration (let us say, the plane x =w). The particles from the low side (x <w) may bounce if they aren't energetic enough; those from the high side always cross, and are accelerated. The trick is not to mistake particles we've just processed for ones which hit the wall from the opposite direction:

```
particle[|x<w & x-v>=w].x :+= delta*(w-x)/v
                        .v :-= delta
        [|x>w & x-v<=w & v<delta].v := -v
                                 .x := 2*w-x
        [|x>w & x-v<=w & v>delta].x :+= delta*(w-x)/v
                                 .v :-= delta
```

Exclusive selection makes this statement work; otherwise particles which have had their interaction might become candidates for another. Exclusive selection also allows the compiler to collapse the first and third cases without changing the semantics.

Scalar-valued Functions of Pseudovectors

The size of a pseudovector, written `#foo[|something]`, is the number of items in it. Note that no field need be specified here; that would be superfluous. Note also that a scalar valued function of a pseudovector is no different from any other scalar expression; it can be used freely inside statements involving pseudovectors of an entirely different base.

An important scalar function is address-of-first. Written `@foo[|expression]`, this returns the index in `foo` of the first record in foo where the expression is true.

Whereas # and @ are functions of the structure of a pseudovector, other functions require some field with values, and are essentially vector-to-scalar functions. We write, for example, `<foo[|bar].xyz` for the minimum value of the xyz field among records in foo in which the bar field is true.

Often we want the value of some field from the record with the min (or max) of another field; say we wanted the value of the abc field from the record with the minimum xyz field where bar was true – we could write

```
foo[@(foo[|bar].xyz= <foo[|bar].xyz)].abc
```

but since this is a fairly common case we introduce a contraction:

```
foo[|bar|<xyz].abc
```

The sum of a field is written `+/foo[|bar].fld`. This diction is borrowed from APL, and is used for all the functions where a similar "summation" is done:

Function	diction
sum	+/foo[|bar].fld
and	&/foo[|bar].fld
or	V/foo[|bar].fld
parity	X/foo[|bar].fld

Non-Scalar Pseudovector Functions

Another borrowed APL diction is `+\\foo.fld`. This means the pseudovector of which each element is the partial sum of the elements up to (and including) that element of the argument.

Control Structures

A control statement is typically a keyword followed by a statement or expression. It is the headline of a group of statements which it controls. For example:

```
for i := @foo[|bar]
    <statement in the for loop>
    <another stmt in the loop>
<statement not in the loop>
```

The semantics of this is that the pseudovector `foo[|bar]` is iterated through. The condition `bar` is only evaluated once. `i` takes on the value of the index of the successive elements in the original pseudovector. Compare this form of the `for` statement with the `while` statement. In the `for` statement, the expression "`i := @foo`..." is not a real assignment but part of the syntax of the `for`. Whereas we can write `while i := @foo[|bar]` which doesn't modify the semantics of the assignment at all, but merely loops until there aren't any bar's left in foo (see below). `For` can also take a conventional-style "i := 3 to 17" construct. ":" is equivalent to "to" in this context.

The `if`, `unless`, `while`, and `until` constructs take conditions which can be either bit scalar values or any pseudovector. A pseudovector used as a condition is "true" if there are any elements and "false" if there are none.

The if statement is as follows:

```
if <condition>, <statement>
                <statement group>
elf <condition>, <statement>
                 <statement group>
else, <statement>
      <statement group>
<statement not controlled>
```

The `elf` section may be repeated or omitted; the `else` may be omitted. `elf` merely means `else if`. The `unless` statement is merely the opposite of the `if` without the `elf`s (elves?).

```
unless <condition>, <statement>
                    <statement group>
but, <statement>
     <statement group>
<statement not controlled>
```

The `while` and `until` forms are likewise a conjugate pair, and simpler, since there are no subsidiary clauses:

```
while <condition>, <statement>
                   <statement group>
<statement not in the loop>
until <condition>, <statement>
                   <statement group>
<statement not in the loop>
```

It is important to remember the difference between these and the associative `for` construct. In the `for`, the condition is evaluated once, and the resulting pseudovector iterated through. In the `while`, the condition is evaluated at every iteration, and looping continues until the condition is false throughout (or true somewhere in the case of `until`).

The syntax of any of the above has a variant:

```
keyword <condition>,
        <statement group>
```

for cases where the line becomes too crowded.

Any of the integer types (int, short, char, and bit) may be used wherever a condition is called for, with the expected semantics that 0 means false, anything else means true.

There are two auxiliary statements that can be used in any of the iterative constructs, `exit <tokens>` and `next <tokens>`. In either case, `<tokens>` is matched with the opening of the enclosing forms and then that form is exited, or skipped to the next iteration. Exit can be used with non-iterative constructs as well, such as `if` and functions. The initial keyword in `<tokens>` may be omitted if this is unambiguous.

Functions

A function call consists of the function name, followed by a fixed number of arguments separated by commas, and optionally *preceded* by another argument. A function definition headline has the same structure, preceded by the keyword `function`. Function definitions may not be nested. A function headline may be the first line in the function:

```
function foo a,b,c
```

or it may be included in a declaration environment if one wishes to declare the parameters there:

```
env scalar parm(bit) a1(float)
    c[66] .a, .b(int)
  function parm gronkulate a1,c
    <function text...>
```

(Note that in these examples `parm` and `a1` are scalars but `c` is an associative array.)

Caml functions are called, and return their values, by *value*; that is to say that functions cannot affect the values of their arguments in the calling environment. Variable scoping is lexical, but closures are not possible because functions cannot be nested.

Example CAML Algorithms

Many areas of programming can benefit from the abilities of a CAM, taking advantage of the parallel processing to simplify algorithms or to gain speed. First an example of simplification:

A Compacting Garbage Collector

This is a garbage collector for a conventional heap, with blocks of pointers. There are N words. There are two sets of comments; those on the same lines as the code tell what the purpose of the code is in terms of the algorithm, and those under the statements tell what they do in terms of the machine and data.

```
env mem[n] .f,.g(bit) .ptr(int)
    (this is the heap itself)
    scalar root
    (this is the root pointer from which pointers are traced)
  function collect
    env scalar bot
    (moves through memory in the compacting phase)
    (memory--is full of blocks with pointers.
     The first word in a block has the length of the
     block (not counting itself) instead of a pointer)

    mem.f&.g := 0
    (clear mark bits)
    mem[root].f := 1

    (mark phase: mark from the root)
    while i := @mem[|f]
      (find half-marked cell, ie not marked from)
      mem[i].g := 1
      (indicate fully marked)
      for k := 1 to mem[i].ptr
      (half-mark everyone it points to)
        mem[mem[k+i].ptr].f := 1
      mem[|g].f := 0
      (remove halfmarks from fullmarked cells)

    (sweep phase:)
    bot := 0
    mem.f := g
    (f now means relocated.  setting it to g
     guards the lengths, which shouldn't be)
```

```
for i := @mem[|g]
(find the first marked place)
  mem[i].g := 0
  (unmark it)
  for k := 0 to mem[i].ptr
  (move down)
 mem[bot+k] := mem[i+k]
  mem[|~f & ptr=i].f := 1
     .ptr := bot
  (relocate all pointers to it)
  bot :+= mem[bot].ptr
  (move up for next)
```

The conventional form of the same garbage collector is much more complex, requiring forwarding addresses, an extra relocation pass, and an extra pointer per record. However, there is no asymtotic speedup.

Graph Algorithms

The garbage collector is much simpler on the CAM, and runs faster by some constant factor, but is still a reflection of a data structure designed for a conventional machine. In some cases, we can speed up an algorithm by a linear factor by appropriate rearrangement of the data structure. Here, for example, is Dijkstra's algorithm for shortest path in a digraph (basic labelling algorithm).

[1. initialize] There are n vertices. we are looking for the shortest path from vertex 1 to all the rest. Array u(n): u(1) = 0, u(i¿1) ≐ +infinity. Array a(n,n): a(i,j) is the arc length from vertex i to vertex j. The set T contains all vertices except (vertex number) 1. Set k to 1.

[2. update u] For all i in T u(i) := min(u(i), u(k)+a(k,i))

[3. new k] K := i such that u(i) is a minimum for i in T. Remove k from T. Repeat steps 2 and 3 until T is empty.

In a straightforward implementation steps 2 and 3 are each linear time in the size of T, so the whole thing is n^2 in number of vertices. We can get around this, however, by rearranging the datastructures. The trick is to distribute u and t around to the edges so we can do the update u operation in parallel. This leaves us with a copy of u(i) for each edge leading into vertex i, with the "true" value of u(i) being the minimum of that set. Likewise t is represented by a t for each edge coming into a vertex; these all change in parallel, so each is the "true" value.

```
env edge(n^2) .x,.y(int) .a,.u(float) .t(bit)
(0. [initialize])
(assume fields x, y, and a have been initialized)
 (assume there is at least one record with y=1;)
  (if not, a dummy may be inserted.)
  edge[|y=1].u := 0
    .t := 0
      [].u := 999999
.t := 1
  k := 1

  while edge[|t]
(2. [update u])
    (essentially the same as above, but we only update
      the copy of u(i) associated with (k,i).)
    edge[|t & x=k].u :min= min(edge[|y=k].u)+a

(3. [new k])
    (taking a ''grand total'' minimum of all the u's
       instead of doing each u(i) incrementally)
    k := edge[|t|min(u)].y
    edge[|y=k].t := 0
```

This new form of the algorithm runs in time proportional to n, the number of vertices. It remains to prove that it works:

Every vertex with at least one incoming edge has at least one u value since the u's are stored corresponding to the y's.

The successive values of u(i) in the conventional version correspond to the values assigned at the various incoming edges in the CAM version.

[a] The current value of a u(i) at any point in the conventional form is a running minimum of the values presented. Thus the min(u) is the same as a "grand total" min over all the u's in the parallel form.

[b] The fact that the u's for all the y=i have not been updated does not matter, since the u used in the addition is the result of the grand total min as per [a], and value of u for this edge can only be higher or equal the "correct" u(i) for the vertex.

The t values are manipulated only by —y=k and are thus set and tested only in blocks corresponding to the individual bits in the serial version.

Minimal Spanning Tree

Like the garbage collector, the minimal spanning tree algorithm gains a marvelous simplicity (compared with an efficient conventional version), but it gains a major speedup as well. The algorithm is one well suited to a CAM: Pick edges of minimum cost that don't form a cycle until all vertices are connected.

A sophisticated conventional implementation is *nlogn* in the number of edges in the graph; this CAM algorithm is linear in the number of edges in the eventual spanning tree.

```
env tree[] .x,.y(int) .cost(float)
  function span tree
    env tree + .bx := x
               .by := y
               .mstp(bit) := 0
      (find edge of min cost which doesn't form a cycle)
      while i := @tree[|bx~=by|<cost]
            tree[i].mstp := 1
            (put this edge in tree, and change the partial tree
               number on one of the subtrees to the other one)
            new := tree[i].bx
            old := tree[i].by
            tree[|bx=old].bx := new
            tree[|by=old].by := new
```

von Neumannizing the Multi-Search Content Addressable Memory

J. Storrs Hall and Saul Y. Levy
Laboratory for Computer Science Research[1]
Rutgers University

Abstract

Bit-parallel VLSI associative memories can compare each of a set of k keys with each of a set of n values in time independent of n and k but proportional to number of bits in a value. However, the results of a search cannot be used as keys in subsequent searches in parallel. This severely limits the usefulness of associative memory for SIMD parallel algorithms.

We analyze the architecture of the associative memory and illuminate the relation of its interconnection scheme to the requirements of multiple associative search. Then we re-interpret the associative memory as a SIMD parallel processor and show that the new formulation is more robust as a vehicle for extended algorithms. The critical element of the approach is to be able to use the results of one stage of computation as control information for the next (hence the title).

[1] Supported by Air Force Office of Scientific Research under AFOSR-86-0294 and DARPA under RADC t.n. B-8-3615 and B-8-3616.

Introduction

Content addressable memories are by far the form of parallel processing with the smallest cost per processor. The typical processing element consists of one bit of memory and an XOR gate[2]. However, CAMs have found themselves in a no-man's-land of commercial development; too expensive when viewed as memory, not powerful enough when viewed as parallel processors.

Part of the reason CAMs have remained a backwater is that algorithm designers are used to thinking in terms of numbers, comparisons, and arithmetic operations like addition and multiplication; whereas to utilize the parallelism of the CAM one must break these down into the component bit operations. Consider, for example, multiple comparison in the partitioned associative architecture of Scherson and Ruhman[4] (see figure 1):

```
[0] key memory contains k keys, data memory contains n items, each
    with a k-bit flag field.
    Set the entire flag field in all words to 1's.
    i := 0.
[1] form a bit vector f consisting of the ith bit of each key.
    AND f into the flag field of each word in which bit i of the
    data item = 1.
[2] f := NOT f.
    AND f into the flag field of each word in which bit i of the
    data item = 0.
[3] increment i and loop to [1] until all bits are done.
```

Multi-compare algorithm for CAM.

Another drawback to CAM for the conventional algorithmicist is that the CAM has annoyingly obscure limitations on its parallelism when operations are described in higher-level terms. For example, we could formulate the following algorithm for simulating an NDA on a string of input: (The NDA is represented by a set of triples of the form (*from*, *label*, *to*) which mean it accepts symbol *label* and goes from state *from* to state *to*.) (see figure 2.)

```
[0] Let S be a bit vector with a bit for every state, all 0 except
    for the initial state.
[1] Select every triple whose FROM field is some state in S.
    (this is a multi-search followed by a mismatch probe
    for all 0's in the flag field.)
    Read an input symbol and further select those triples whose
    LABEL field matches the symbol.
[2] Set S to the union of the states represented by the TO fields
    of the selected triples.
[3] If there are any more symbols to read, goto [1].  Otherwise,
    if S contains a terminal state, the string is accepted;
    if it does not, the string is not accepted.
```

Naive NDA algorithm using Multi-Compare.

The first problem with this algorithm is it requires moving a set of values from primary to key memory. This can be done in a time depending only on the number of values moved, but that still degrades the complexity of the algorithm to sequential running time.

This problem can be overcome with the structure in Figure 2. We let the *from* and *to* fields be of length the number of states, and denote which by a single 1 bit in the field. Now we cannot use multi-compare, (since being bit-serial it would increase running time to worse than sequential) but with the representation thus "blown out" a standard associative probe suffices. Then S can be formed by the standard ORring read of selected *to* fields.

However, we have created a worse problem: the representation is so verbose no reasonably-sized problem would fit in a reasonably-sized CAM! Since the sequential speed penalty can never exceed the number of states, the only NDAs worth running in parallel are relatively large ones. But the size of the CAM required by the new representation is $> 2EV$, ie, quadratic in the size of the graph.

Consider what happens when we extend the NDA to handle a more realistic interpretation problem. Let each input be a vector of probabilities corresponding to all the symbols in the recognized set. Further let each edge in the state graph have an attached probability as well. Then at a given step in the interpretation process, state i has probability S_i, there is input probability I_c for each symbol c, and each edge from state i to j bearing label c has probability $E^c_{i,j}$, then for the next step the state probabilities are

$$S_j = \sum_c I_c(\sum_i S_i E^c_{i,j})$$

. The CAM, even with a count-responders circuit[2] allowing a single overall summation, would not be able to perform the "multi-summation" implied.

The Structure of Communication in Pure CAM

In the standard CAM model, each bit of storage and its associated comparator undergoes the following interactions (see Fig. 3):

1. It receives data from the CPU on the major data bus, which data is also sent to all the other bits in its column;
2. it sends data to the CPU along the major data bus, which may be ORred with data from other bits in its column;
3. it receives data from its response flag (the word-enable line), which is also sent to all the other bits in its row; and
4. it sends data to its response flag, which is ORred with similar data from all the bits of its row.

It is plain that at an abstract enough level, the rows and columns of the CAM are interchangeable, as long as the functions of the CPU bus word and/or the response bit vector are properly accounted for.

A second feature to notice is that in the multi-compare operation, the **flags** field is being used in parallel at every stage of the operation, whereas the **data** field is being iterated through a bit at a time, just as are the **keys**. It should again be apparent that in an abstract enough view, the data and keys are interchangeable, and that the flags field forms an n-by-k array of one-bit elements doing nk comparisons one bit at a time.

Scherson[4] also shows how to do "multi-operand arithmetic" with the CAM, but this means adding (say) each **data** item to *one* of the **key** items (in parallel), rather than doing nk arithmetic operations at once. However, there are many problems in which it would be advantageous to be able to do the complete "cross product" set of operations; and since we have nk processing elements, however minimal, it would seem reasonable to try to do this.

The ParaSol Architecture

Thus we want to look at an architecture that consists of a rectangular grid of processing elements, where we are able to broadcast to the rows and the columns, and take the OR of the rows and the columns. These functions can both be represented as a single wire (as in the standard CAM model) or as a tree; as the size of the array grows, the tree with its logarithmic time operation becomes more attractive compared to the wire with its linear propagation time. (See Fig. 4)

At this point we can depart from a strict interpretation of the original CAM functionality and ask what the architecture is capable of in terms of a greater range of operations, such as the "cross product" arithmetic operations mentioned above. If the non-terminal nodes of the trees are considered to be processing elements, the trees are themselves capable of non-trivial computation in support of the base array. In particular, if a one-bit ALU is provided at each node, then the elements of each row (or column) of the base array can be summed by a pipelined carry-save bit-serial operation.

This gives us an architecture capable of supporting a generalized "dot-product" operation directly. One vector can be sent down the column trees, and another down the row trees. In the processing elements of the array, each element of one is combined with each element of the other with some operation (equality test, for multi-compare). The results are then accumulated by one set of the trees under some other operation (OR, for multi-compare), giving the results.

The only further thing that need be done is to make the operations cascadable, "von Neumannizing" the architecture. This is simply done by bringing the roots of the column trees and the row trees together in a vector of processing elements. (Thus row 1 and column 1 have a common root node; row 2 and column 2 have a common root node, etc.) We will also add a single tree whose leaves will be this diagonal vector of roots of the row/column trees, for reasons that will become apparent later.

Technology Constraints in Associative Arrays

A major problem with associative memory is that while it allows the highest parallelism for a given amount of hardware, it must be loaded serially. For many problems, this destroys the parallelism. Associative processor arrays have tended to try to get around this by simulating the associative word by a bit-serial processing element, and having at least one pin for each PE on the chip. Examples are the MPP[5], the DAP[6], and the Connection Machine[7]. However, the bit-serial nature of the PEs means that the multi-compare operation no longer works[4].[1]

Can we use the same trick and retain the parallel multi-compare operation? The answer is yes! The orthogonal broadcast allows the same data to reach the PEs as reach the bits in the `flags` field of the original algorithm.

However, we have the same problem as any associative array processor, that of pinout.[3] For the ParaSol structure, a chip with n-by-n PEs needs n^2 memory pins of this kind. This acts to reduce the hardware efficiency of the architecture considerably over that of pure CAM; one typically ends up with (say) 64 PEs per chip instead of 12000.[8]

There are two functions immediately clamoring for space on the chip, given that PEs are to be so few and far between. The first is making the PEs themselves multibit ALUs, and the other is local memory. At first it seems silly to make a multibit ALU that is fed by a single bit-serial pin; however, examining the algo-

[1] In all fairness, it must be pointed out that the DAP etc. are capable of the multi-compare operation, in much the same way it works on the ParaSol. Indeed, a Connection Machine with its routers could simulate a ParaSol fairly well, except that its general communication scheme reduces bandwidth along the critical paths to a fairly substantial degree.

·:hms we describe below reveals that the operation done most often at the PEs is ·:.ultiplication. Multiplication on one-bit ALUs is a disaster.[2] Thus a 32-bit ALU and a few words of local memory makes an enormous difference in the time each PE takes to do such operations. Our preliminary design for such a PE with 16 words of local memory comes to about 5000 transistors, for a total for 320,000 for the 64 PEs.

The processing element consists of a 32-bit "accumulator" and 16 words (see Fig. 5). Two of the words are shift registers, tied to the bit-serial lines to the tree and memory. These allow data to be clocked in from either or both of these sources independently while other operations are being done. In the basic multiply operation, the multiplicand is in one of the words and the multiplier is coming in from one of the trees LSB first. As each bit arrives, it is anded with the multiplicand and added to the product, which is shifted to the right and the LSB sent up the orthogonal tree. (If it is desired to save the lower bits of the product they can be reflected through the "tree in" line and shifted into the "tree in" register.) Simultaneously the "memory i/o" register may be reading the multiplicand for the next operation.

The seven-node trees connecting each 8-element row and column of such a chip are easily included. It is advantageous to be able to run the trees both directions simultaneously, so there are two pins per tree for 8 row trees and 8 column trees. This comes to 96 pins per base array chip, leaving room for a good double handful of control lines in even an average pin grid array. The upper levels of a row or column tree could easily be placed on a single chip, complete with appropriate registers to convert from bit-serial to parallel and vice versa. With a 32-bit bus for its root connection, and two pins for each of 32 subtrees, this chip has the same number of data lines as the base array chip.

Using the tree chip pinout as a limiting factor, a full blown ParaSol would have 64K base array PEs. It would require 20 boards, 16 containing 8 by 8 base array PE chips (and memory) and 4 containing 8 by 8 diagonal PEs. (See Fig. 7). It is possible to arrange the boards in three-space so that each diagonal PE board is adjacent (at least diagonally) to all the base array boards it communicates with; and the interconnection technology ("button boards", see []) exists to build the machine in that configuration. This configuration would be capable of well over ten billion multiplications per second at peak rates.

Using the ParaSol

Let us now repair to the example problem of parsing an input string with a nondeterministic finite state automaton (NDA). The multiple search operation

[2] Multiplication of 32-bit quantities on a 1024-element DAP achieves a peak rate barely higher than that of its serial controller; Thinking Machines rather than do number crunching on the one-bit PEs added Weitek floating point chips to the CM.

combined with a simple data structure implemented a linear-time algorithm for this problem, but the lack of self-referentiality in proposed CAMs precluded it. A modification of the pure CAM, along with a more complex coding scheme, sufficed but remained brittle, being insufficient for, e.g., an NDA with weighted edges encoding a hidden Markov model[1] with transition function

$$S_j = \sum_c I_c(\sum_i S_i E^c_{i,j}).$$

(See the Introduction.) This boils down to be the multiplication of the matrix E by the vector S for each symbol c (with a different E for each c), taking the vectors thus formed together to form a new matrix, and multiplying that one by the vector I to form the result.

Multiplication of a matrix by a vector is very nearly the primitive operation for the ParaSol. The matrix is stored in base array memory; the vector in the diagonal processors. The vector is then broadcast down the columns (for $M \times V$; down the rows for $V \times M$)[1] and the broadcast value multiplied by the stored value at each PE. Then the values are summed up the row trees (column trees for the commuted case) to the diagonal PEs.

Another increasingly common application is the simulation of neural network models. In the standard formulation of these models, a matrix of weights is multiplied by a vector of input values (and then the result vector is subject to some non-linearizing function, such as a threshold for perceptrons or Hopfield nets, or a sigmoid function for the popular backprop model.) Oddly enough, when this matrix multiplication is put onto the ParaSol architecture in the standard way, each row tree ends up doing exactly the summation of one "neuron" unit in the neural net model.

What happens if we have a bigger neural net (or indeed any matrix multiplication problem) that is bigger than the physical ParaSol? Luckily, the structure of this operation is remarkably congenial to division into smaller matrix multiplication problems. And herein lies the real power of the ParaSol. Any matrix M may be viewed as the tesselation of smaller matrices $m_{i,j}$ and any vector V as the concatenation of smaller vectors v_i. (Assume all the small matrices are k-by-k and all the small vectors length k; and that our machine is k-by-k.) Now every element in the overall product, $(M \times V)_i$, is seen to be the sum of corresponding elements of products of the submatrices and subvectors,

$$(M \times V)_i = \sum_j M_{i,j} V_j = \sum_b \sum_d (m_{a,b})_{c,d} (v_b)_d = (\sum_b m_{a,b} \times v_b)_c$$

where $a = \lfloor i/k \rfloor$ and $c = i(mod\ k)$.

[1] Such models occur commonly in interpretation algorithms in fields such as speech recognition; see, e.g., [8] p. 137.

[1] For clarity, we will use $\times$ to indicate matrix multiplication.

References

[1] Batcher, K. E., "The Multidimensional Access Memory in STARAN", IEEE Trans. Comput., c-26 (1977), pp 174-177.

[2] Foster, Caxton C.: **Content Addressable Parallel Processors**, Van Nostrand Reinhold, New York, 1976

[3] Fountain, Terry: **Processor Arrays: Architecture and Applications**, Academic Press, London, 1987

[4] Scherson, Isaac and Smil Ruhman: "Multi-operand Arithmetic in a Partitioned Associative Architecture", Journal of Parallel and Distributed Computing 5, (1988) pp 655-668.

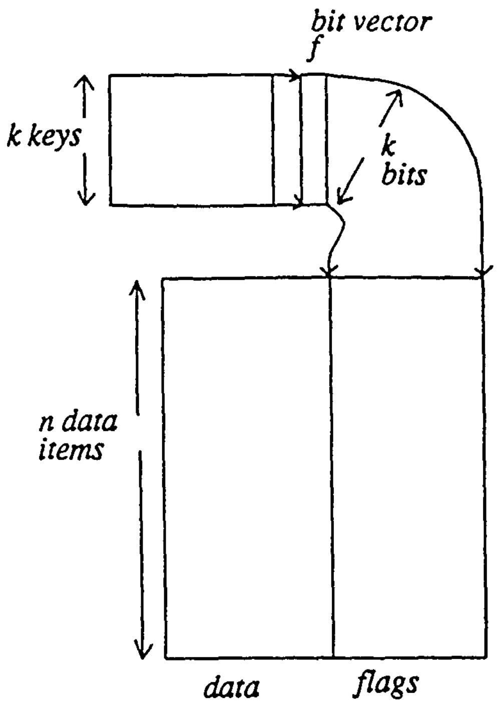


Fig 1. Scherson's partitioned associative memory and the multi-compare operation.

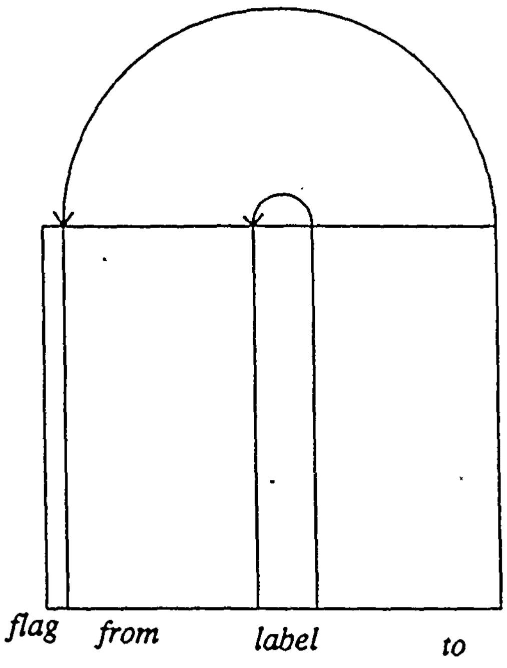


Fig 2. "von Neumannized" associative memory and the algorithm for NDA interpretation.

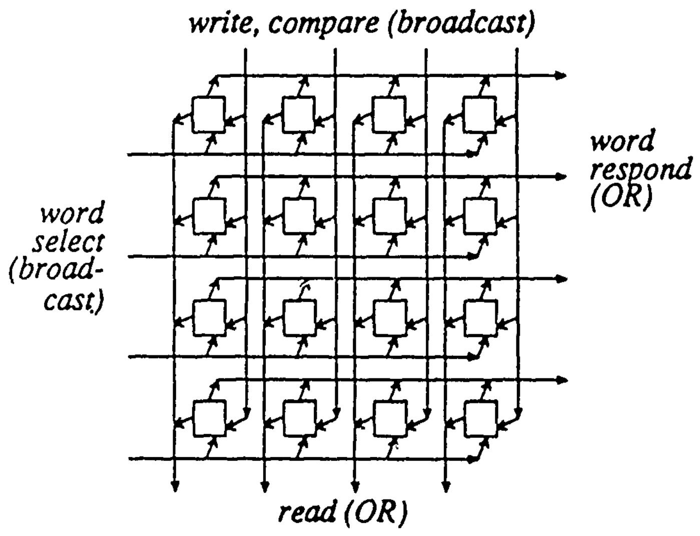


Fig 3. structure of communications in hardware associative memory.

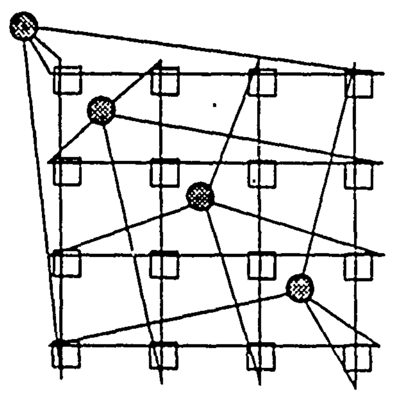
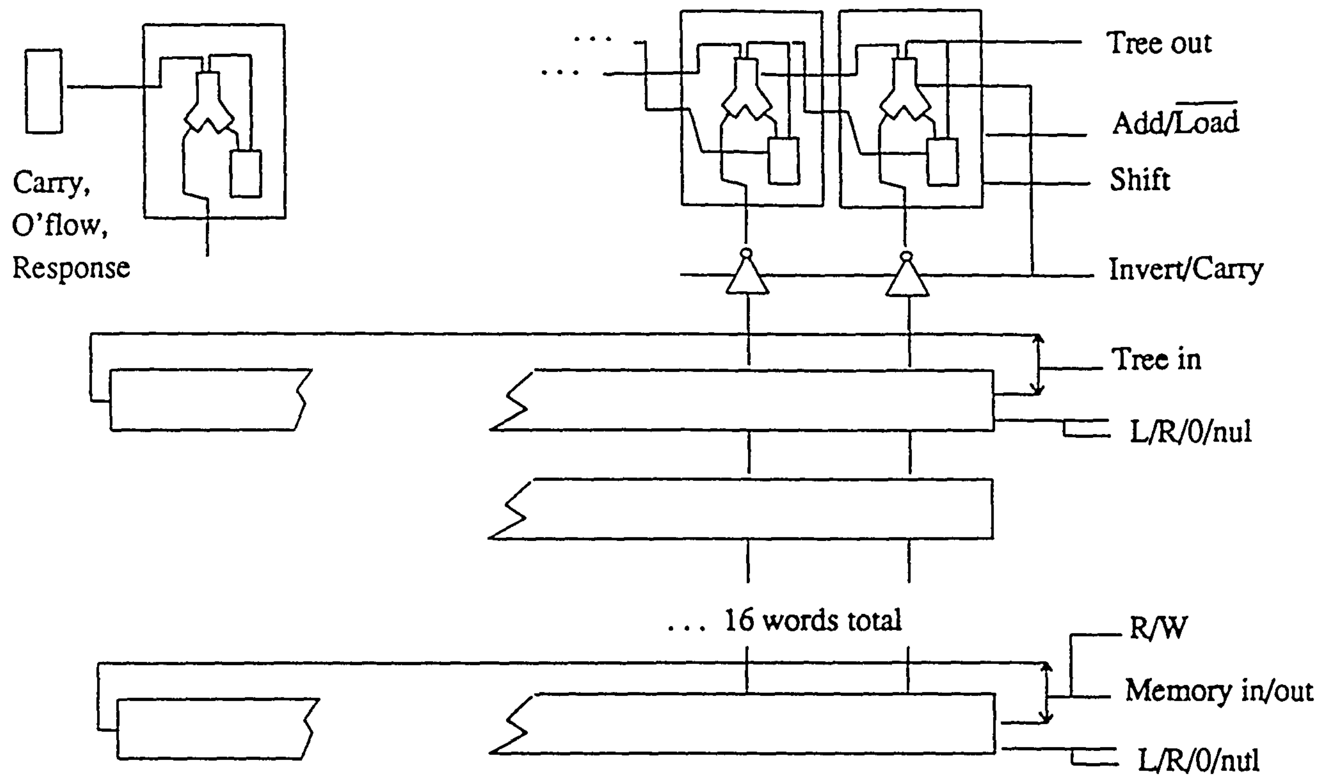


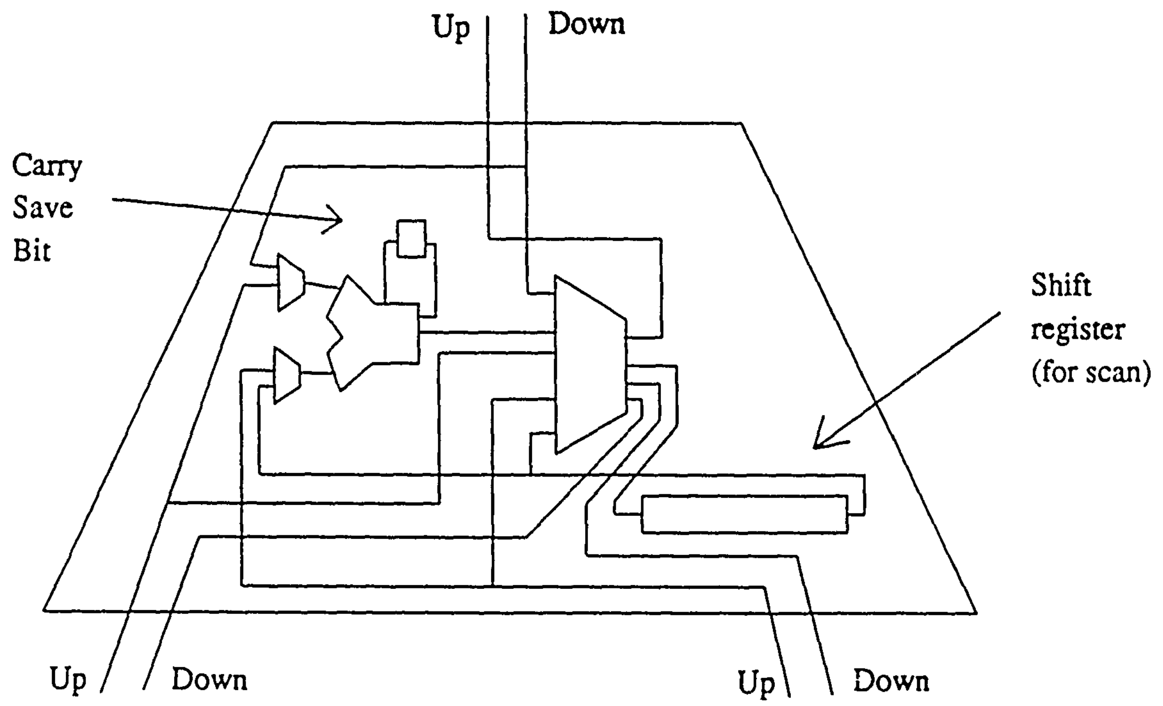
Fig 4. The ParaSol architecture. Each row tree [i] meets column tree [i] at the ith diagonal PE. (the upper-level tree connecting the diagonal PE's is not shown for clarity.

figures for

von Neumannizing the Multi-Search Content Addressable Memory



(Fig. 5) One ParaSol Processing Element



A Tree Node

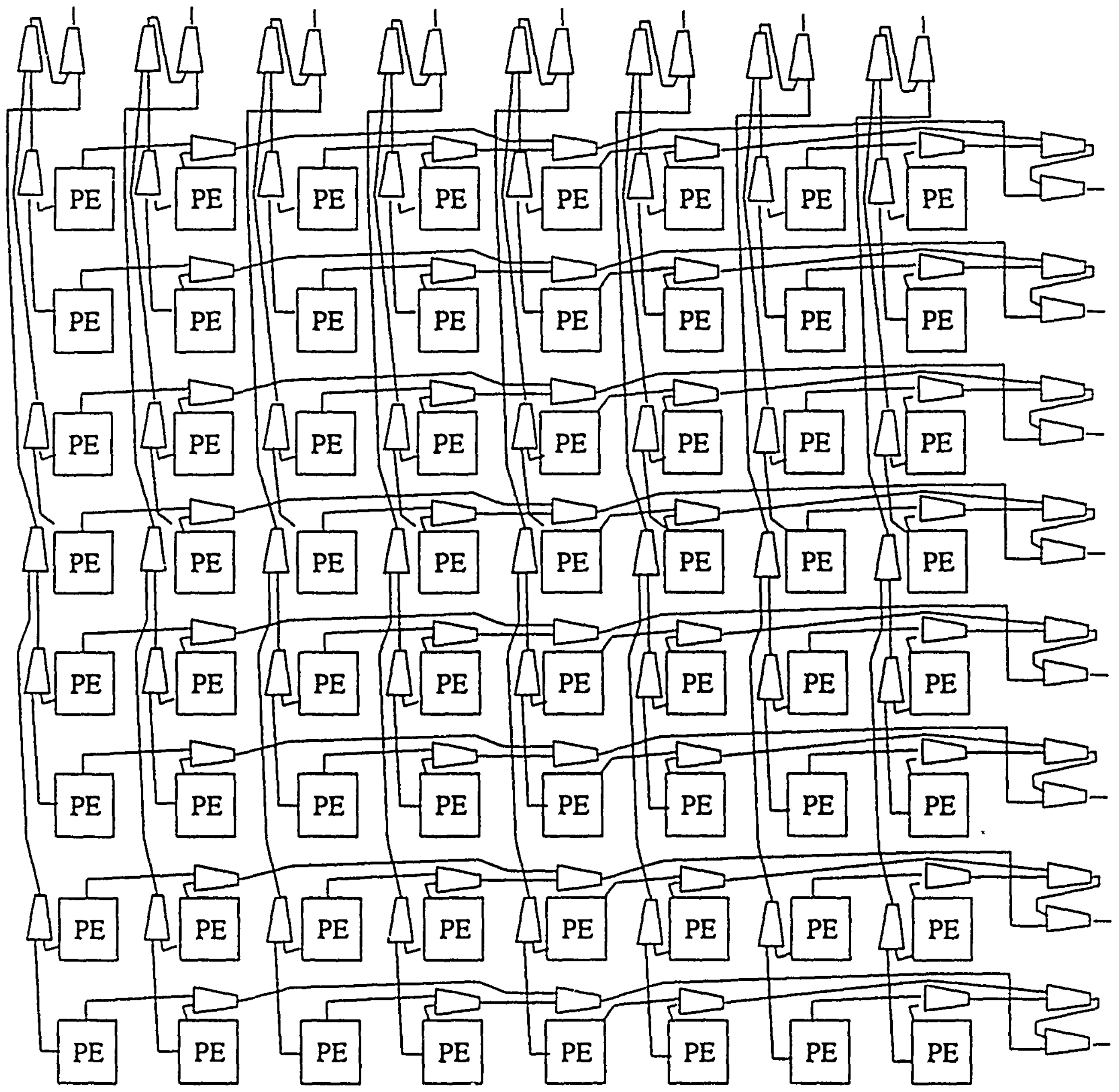


Fig 6: One ParaSol base array chip

0	1	2	3
4	5	6	7
8	9	A	B
C	D	E	F

Logical arrangement of boards. "p0, p5, pA, and pF" represent boards containing diagonal PEs.

Other boards contain up to 8x8 base array chips and memory.

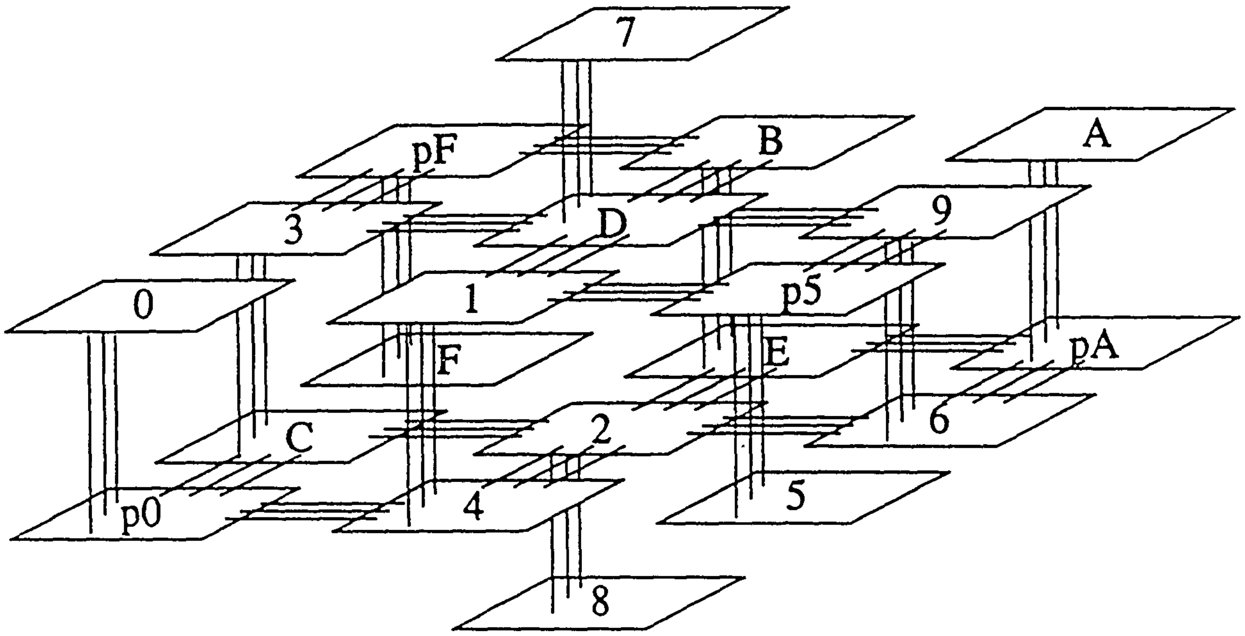


Fig 7: 3-D arrangement of ParaSol boards

"Button board" technology allows arranging boards in space as shown. In practice, the two center planes would each be one larger board. Note that each diagonal processor board is at worst adjacent by a 3-d diagonal from every array board on its row or column.

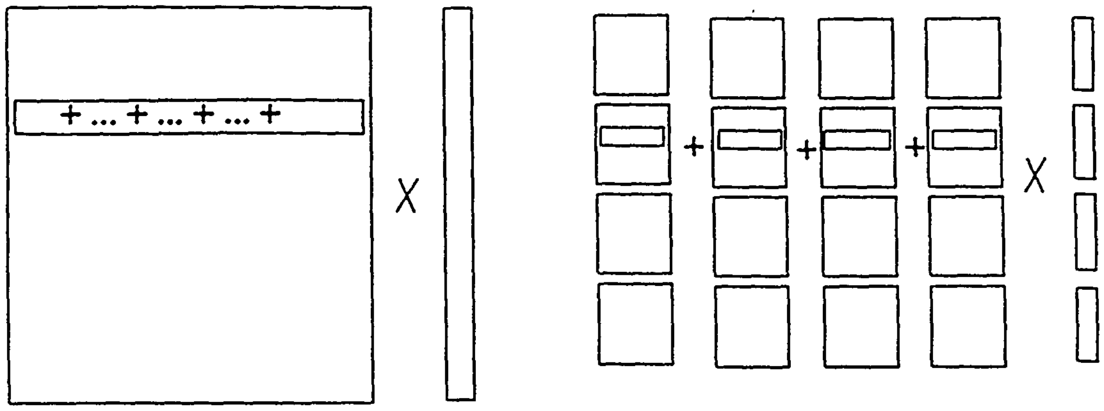


Fig 8: The recursive breakdown of Matrix multiplication.

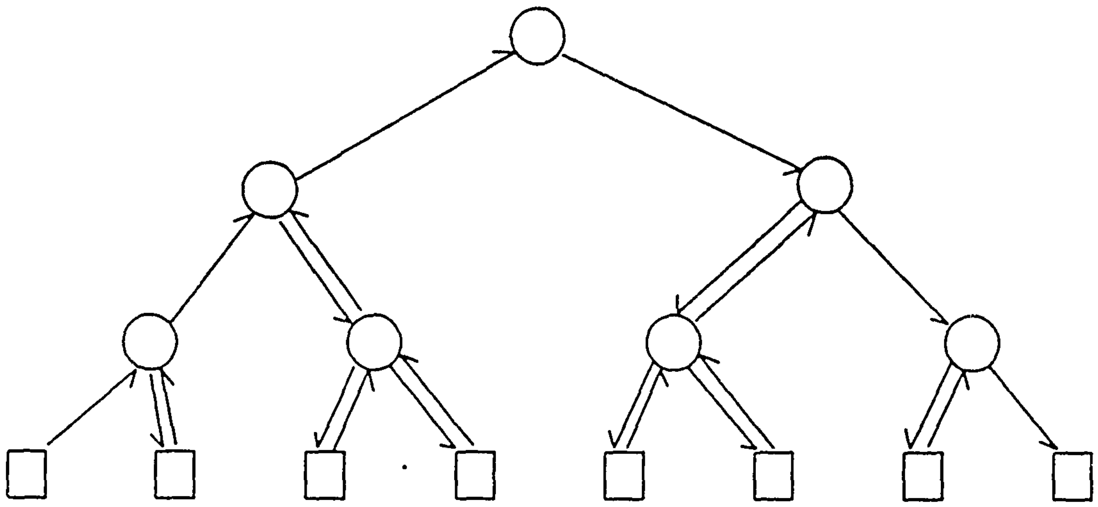


Fig. 9: Shifting using a row or column tree.

DAP codes

1 Interface for using C on the DAP

The DAP software/hardware interface is lacking many of the features required of a general purpose computer, most notably the ability to read from a keyboard or write to a terminal. An interface that provides many of these features has been designed and implemented.

The interface is designed to function with the version of C we have developed for the DAP; however, the functions composing the design may be called from Fortran or APAL.

1.1 Communication Interface

The design has two communication interface modules, one running on the DAP and the other on the host machine, in our case a SUN. Actions that cannot be performed exclusively on the DAP are "simulated" by requesting the host SUN perform the desired action and report the results back to the DAP. This interaction between the DAP and SUN is controlled by a protocol employed between the communication interface modules. These requests are transmitted between the machines using the DAP supplied routines "amt5stop", "amt5start", "dapsen", and "daprec".

The process is initiated by the DAP writing the required data into a communication buffer and transferring control to the SUN using the "amt5stop" command. When control has been returned to the SUN it issues a "daprec" command to read the communication buffer. Once read the data is interpreted and the required actions performed by the SUN. Any data to be returned to the DAP is stored and upon completion of the operation transmitted to the DAP using the "dapsen" command. The final action of the SUN is to issue the "amt5start" command which returns control to the DAP at the point where it had given control to the SUN. The DAP then reads the communication buffer and performs the actions required to complete the operation.

The C code in Figure 1 is an example of when such interactions are required. It reads two integers from the keyboard and print them on the screen. Since the DAP has no support for reading the keyboard or writing to the screen, the communication interface must provide these facilities.

```
scanf("%o %x",&i1,&i2);
printf("%d %d \n",i1,i2);
```

Figure 1: C code requiring DAP communication to host SUN

The DAP half of the interface processes the three arguments of the scanf and passes them along with a request for a scanf operation to the host SUN. The SUN half of the interface decodes the request, performs the required scanf using C's standard IO package, and returns any resulting data to the DAP. Finally, the DAP stores the returned values in the specified locations. The communication interface handles the printf in a similar fashion.

1.2 Running C on the DAP

Once the DAP C code has been compiled, assembled and linked to form the DAP executable image it can be run using a standard interface. The interface accepts the name of the file containing the DAP executable image (include the extension), loads the file onto the DAP, sets up the necessary interface between the host SUN and the DAP, and transfers control to the DAP. The compiler has included in the executable image the necessary prologue, epilogue, and subroutine packages to execute the compiled C code as though it were on a general purpose machine.

Interface

Interface (spelled with a capital I) is the SUN code that is used to initiate a DAP C program. It accepts the file name containing the executable image from the command line or, if necessary, prompts the user for a file name. It then establishes contact with the DAP using the DAP command *dapcon* and finally transfers control to the DAP at the label DAPmain.

DAPmain

DAPmain is the program which provides the epilogue and prologue for all DAP C code. DAPmain first gets the ids of the the host SUN's three standard IO files (i.e. stdin, stdout, stderr), it then initializes DAP memory for use by malloc and other related memory handling functions. Once the prologue is completed it calls main (the main procedure provided in the C program). When the C program has completed, control is returned to DAPmain which cleans up the dynamic memory used by the program and returns control to the host SUN.

1.3 Interface routines callable from C

For each operation that requires the communication interface (e.g. scanf, printf, ReadMouse, etc.) two functions must be written, one for the DAP half of the interface and one for the SUN half. Currently there are eleven functions supported by the interface. Additional functions can be added by writing the necessary pair of functions, including them in the interface package, and updating the SUN's main interface control. The functions supported are fopen, fclose, feof, GetSTDs, DAP_Exit, printf, fprintf, scanf, fscanf, OpenMouse, and ReadMouse.

fopen, fclose, feof

These three functions operate as defined in C's standard IO package.

GetSTDs and DAP_Exit

These two functions provide the necessary "hidden" interfaces to the host SUN. GetSTDs initializes the DAP so it can use the default files stdin, stdout, and stderr. DAP_Exit terminates the DAP program returning control to the host SUN.

printf and fprintf

These two functions operate as defined in C's standard IO package. A call of printf is simulated using fprintf with stdout as the file id. The DAP half of the interface parses the format statement and sets up a data structure containing, the format statement, the data to be written, and a structure to contain any returned results(recall that some printf conversion specifications return values). It passes this data structure along with a request for an fprintf to the SUN.

The SUN half of the interface also parses the format statement. It performs the specified translations using C's standard IO interface and stores the results in the structure it received from the DAP. Upon completion it returns this information to the DAP. The data returned is copied into the proper DAP locations using the data structure set up during the initial parse of the format string.

scanf and fscanf

These two functions operate as defined in C's standard IO package. A call of scanf is simulated using fscanf with stdin as the file id. The DAP half of the interface passes the format statement along with a request for an fscanf to the SUN.

The Sun half of the interface parses the format statement. It performs the specified translations using C's standard IO interface and stores the results in a structure to be returned to the DAP. Upon completion it returns this information to the DAP. Upon return, the DAP code parses the format statement and stores the results in the proper DAP locations.

OpenMouse, ReadMouse

These two functions are used to control the mouse. OpenMouse opens the host SUN's mouse device and flushes all pending IO events from that device. It then establishes a correct sync for the mouse and returns control to the DAP.

ReadMouse reads the next "significant event" from the mouse. The DAP half of the interface is a simple request for input, the SUN half processes the actual mouse input and constructs "significant events" for transmission to the DAP. These events are seen from the DAP as a four tuple: the first item is a two's complement integer given the change in the x coordinate since the last call of ReadMouse, the second is is the change in the y coordinate, the third is an encoded button status, and the fourth is an encoded indication of which button has changed since the last call of ReadMouse.

The encodings of the button information is done using the low order three bits. The 1 (low order) bit represents the right button, the 2 bit represents the middle button, and the

4 bit (high order) represents the left button. The button status is formed by setting a bit to 1 of the button is down and to 0 if the button is up. The change in bu..on status is set to 1 if a button has changed and set to 0 if it has not changed. For example, a button status of 5 and a changed button status of 3 indicates that the left and right buttons are currently depressed and the left and middle buttons have changed since the last call to ReadMouse.

The need of transmitting "significant events" arises due to the high latency in the communication interface between the host SUN and DAP. The latency problem is inherent in AMT's interface routines "amt5start", "amt5stop", "dapsen", and "daprec" which require a minimum of several milliseconds to complete the simplest operation. Due to this latency we designed an interface that would reduce the number of transmissions. This was done by forcing the SUN to transmit only *significant* events.

A event is *significant* in one of two ways: it is either an event that changes the button status or it is the last event in the SUN's mouse queue. This definition allows the input queue for the mouse to be compressed by combining all pending events that are exclusively mouse movement events into one *significant* event. This substantially reduces the number of transmissions performed and improves the performance of the mouse interface.

2 Memory Allocation for DAP C

Most C programs require the ability to allocate and free memory dynamically. In order to support these operations a package of five functions was built. Three of these function are callable from DAP C and are named and operate just as their equivalent C versions. These functions are malloc, free, and realloc. The other two functions, Initialize_Allocate and Terminate_Allocate, are used to initially configure the memory and to record use statistics before termination. The user is unaware of these function since they are called by DAPmain.

The functions are implemented by reserving a fixed amount of memory when a program is initiated and allocating and freeing from this memory. This approach is limited by the inability to dynamically obtain memory from AMT's operating system for the DAP; however, other than this limitation, the memory management routines work exactly as their C counterparts. In addition, these procedures keep statistics on the use of dynamic memory by recording data describing each call to malloc. For each call the system records the amount of memory requested, the size of the free block from which it was obtained, the length of the free list at the time of the call, and the number of free list elements scanned. This data is buffered before being sent to the host and can be read using the program AllocStats.

2.1 APAL support of the Memory Allocation Package

The operation of the memory manager is controlled by three assembly variables: PlaneCnt which specifies the number of planes to reserve when a program begins, StatsBL which specifies the buffer size used for recording statistics, and Epsilon which specifies, in planes, the smallest number of planes that can be allocated. This memory is maintained as a set of contiguous planes (i.e. 32 words) using a linked list of available sections, a map of the allocated planes, and a link from the last available plane of section to the first available plane of that section. The sections are kept maximal is size by compacting planes as they are freed by the user.

The linked list of free blocks is maintained in the same memory that is allocated to the user. This *overlap* of memory is feasible since unallocated planes of memory are, by design, not accessible by the user. Using this scheme the user and memory manager have mutually exclusive use of planes of the memory; however, it is possible for a user to *accidentally* write over the package's free list. The map of each block's status is maintained in an area of the DAP memory accessible only to the memory manager. Each plane of available memory requires a halfword of memory in this table. The link from the last plane of an available section to the first is *overlapped* with the users memory as is the linked list.

Data Representation

Each available plane has four field known to the memory manage. These fields are only significant when the plane has not been allocated. Once allocated the fields are not used by the manager. The fields are shown in Figure 2.

Forward: a forward link to the next available free section
Back: a backward link to the previous available free section
Size: the number of planes in this section
Up: a link to the first plane of this section

Figure 2: Fields used by Memory Manager

The first three fields are used to form the linked list of available sections. The last field (i.e. Up) is used to link the last plane of a section to the first plane of the section. Initially, the linked list contains one section of memory that holds all planes available.

The map of allocated blocks is kept as a vector of half words that are initially zero. When a section of memory is allocated the half word corresponding to the first plane is set to the number of planes allocated. The half word corresponding to the last plane of memory is set to -1.

malloc

This function accepts one argument, the number of words requested, and returns the address of the first words allocated or a zero to indicate that the allocation was not possible. This functions requires a time proportional to the number of sections in the linked list. In addition to allocating memory malloc keeps a record of each allocation request.

In order to find an available section, malloc performs a sequential search of the linked list using a first fit algorithm. As soon as a section that contains at least the requested number planes is found the algorithm terminates its scan and allocates the required planes.

If the sections is an *exact* fit to the request, the entire section is allocated. If the section is larger than the request, the section is partitioned into two sections with one being the size of the request. The number of planes allocated is entered in the halfword of the map corresponding to the first plane in the allocated section and a -1 is entered in the halfword corresponding to the last word of the section. The linked list is updated to indicate the change in either the size of an available section or the number of available sections.

free

The functions accepts the address of a section of memory that has been allocated to the user and returns that section to the free list. It provides no return value. Free checks that the address has been allocated by testing to see if the plane map has a value other than 0 or -1. If the section had been allocated, free proceeds to free the section, compacting it with other sections if possible. These checks, as well as the ensuing compactions, can be done in constant time using the data structures described.

realloc

This functions accepts two arguments. The first is the address of memory that is to be reallocated, and the second is the number of words required in the reallocation. It returns the address of the reallocation if it was successful, otherwise it returns a 0. Conceptually, this function is a hybrid of malloc and free. Realloc attempts to satisfy the request using a sequence of options. First it decides if the request is for expanding or contracting the section. If the request is for contraction, the algorithm frees the planes at the end of the current section.

If the request is for expansion realloc attempts to *extend* the current section by adding planes at the end. If this fails, it looks for a block large enough to satisfy the request. If found, the contents of the current block is copied to this newly allocated block, the current block is freed, and the address of the newly allocated block is returned. If there is no block large enough to satisfy the request, realloc attempts to *extend* the current section by adding planes to the beginning of the section. If extension is possible in this directions the data is copied as required and the memory restructured.

If all these attempts fail, realloc returns a zero indicating that the requested action was not possible. In fact there is one condition that could enable an extension that is not considered by our code. This case is when all checks mentioned have failed but there are two sections, one preceding and one following the current section. If the sum of the lengths of these two section plus the length of the current section is sufficient to meet the request, the data could be moved and the request satisfied. It was decide that this case was so rare that it would not be implemented.

of contiguous planes (i.e. 32 words) using a linked list of available sections, a map of the allocated planes, and a link from the last available plane of section to the first available plane of that section. The sections are kept maximal is size by compacting planes as they are freed by the user.

The linked list of free blocks is maintained in the same memory that is allocated to the user. This *overlap* of memory is feasible since unallocated planes of memory are, by design, not accessible by the user. Using this scheme the user and memory manager have mutually exclusive use of planes of the memory; however, it is possible for a user to *accidentally* write over the package's free list. The map of each block's status is maintained in an area of the DAP memory accessible only to the memory manager. Each plane of available memory requires a halfword of memory in this table. The link from the last plane of an available section to the first is *overlapped* with the users memory as is the linked list.

Data Representation

Each available plane has four field known to the memory manage. These fields are only significant when the plane has not been allocated. Once allocated the fields are not used by the manager. The fields are shown in Figure 2.

Forward: a forward link to the next available free section
Back: a backward link to the previous available free section
Size: the number of planes in this section
Up: a link to the first plane of this section

Figure 2: Fields used by Memory Manager

The first three fields are used to form the linked list of available sections. The last field (i.e. Up) is used to link the last plane of a section to the first plane of the section. Initially, the linked list contains one section of memory that holds all planes available.

The map of allocated blocks is kept as a vector of half words that are initially zero. When a section of memory is allocated the half word corresponding to the first plane is set to the number of planes allocated. The half word corresponding to the last plane of memory is set to -1.

malloc

This function accepts one argument, the number of words requested, and returns the address of the first words allocated or a zero to indicate that the allocation was not possible. This functions requires a time p oportional to the number of sections in the linked list. In addition to allocating memory malloc keeps a record of each allocation request.

In order to find an available section, malloc performs a sequential search of the linked list using a first fit algorithm. As soon as a section that contains at least the requested number

2.2 Obtaining statistics about Memory Allocation performance

The statistics recorded by malloc can be reviewed by calling the program AllocStats. This program prints out a list, with headers, of the requests made of malloc. The data is kept in the file *.allocate.states* and is cumulative. If the data in the file is no longer needed the file can be deleted and the statistics will begin to accrue from the time of this deletion.

3 Low level support for DAP

Low level support macros and functions have been developed to assist in the development of DAP C. They provide a simple method to perform the repetitive operations required within a program (e.g. linking conventions and accessing the system stack).

The macros developed are divided into three major class. Each of these classes is available to any APAL program; however, the design of each macro is done with the intent to support either the running or development of the DAP C environment.

The functions developed are used to support integer division within C and to provide access directly from DAP C to AMT's Low Level Graphics Library.

3.1 Macro support of Calling Conventions for DAP C

The set of macros described in this section are used primarily for controlling the linking between subroutines during run time. The macros provide a *uniform* method for accessing parameters and returning values.

Linkage conventions for DAP C

The linkage conventions used to implement DAP C were design to facilitate the calling of C programs from C programs while at the same time allowing C to *reasonably* interface with APAL and AMT's Fortran Libraries. In all cases the conventions were kept consistent with AMT's *basic* conventions so psam (i.e. AMT's run time debugger) could be fully utilized.

The conventions used are shown in figure 3.

Entering

This macro is used to establish the necessary linkage between the calling and called program. It should be called immediately upon entry to an APAL procedure. It accepts one argument which specifies how many arguments are passed by a fortran call. If the call is from a DAP C or an APAL program the number of arguments is determined at run time.

This macro determines the calling convention used, establishes the necessary linkages, and sets up the current activation record as defined in Figure 3. There are 7 different calling

on entry:
- m0: callers return address
- m1: in C - first argument
- m2: in C - second argument
- m3: in C - third argument
- m4: in C - fourth argument
- m5: address of the activation record of the called procedure
- m6: low order 5 bits - identification of the calling convention
  high order 27 bits - address of the activation record of the calling procedure

during execution:
- m6: address of current activation record
- m7: address of next available word in the stack

on return:
- m6: address of callers activation record

Activation Record:
- offset 0: address of previous activation record
- offset 1: high order 5 bits - calling convention
  low order 27 bits - return address
- offset 7: in C - address of next available word in stack
- offset 8: in Fortran -address of return value
- offset 9: first argument
- offset 10: second argument
- offset n+8: nth argument

Figure 3: Calling Conventions

conventions supported: AMT's fortran or system convention, Rutgers APAL convention, and 5 variants for DAP C. The C variants differ only in the number of arguments passed in registers (i.e. 0 through 4). This macro determines the calling convention at run time so that one procedure that can be called from a variety codes using different conventions (e.g. Fortran or DAP C).

Exiting

This macro is the complement of Entering. It restores the callers activation record, sets up the return value as required by the calling convention in use, and returns control to the caller.

IsItC

This macro determines if the calling convention used is one of the DAP C variants. It accepts one parameter, a register number, ' and set that register to 1 if the calling convention is a DAP C variant or to 0 otherwise.

GetArgValue

This macro retrieves the value of an argument passed by a calling program. It distinguishes between a DAP C call and a fortran or system call and uses the necessary convention to obtain the value. It accepts two argument, the number of the register in which to place the value and an integer that specifies which argument to retrieve.

GetArgAddress

This macro retrieves the address of an argument passed by a calling program. It does *not* distinguish between a DAP C call and a fortran or system call since the call by reference format is the same for both. It accepts two argument, the number of the register the address will be placed in and an integer that specifies which argument to retrieve.

StoreReturnValue

This macro records a value for return to the calling function. It distinguishes between a DAP C call and a fortran or system call and uses the necessary convention for returning the value. The macro accepts one argument, the number of the register containing the value to be returned. The register must contains this value when StoreReturnValue is invoked; the macro sets up the return value but the value is not returned until the exit macro is invoked. If this macro is called more than once during the execution of an APAL routine the value returned will be that specified in the last use.

Calling

This macro is used when an APAL routine needs to call another procedure. It accepts five argument: the name of the procedure to call, a list of local registers to protect across the call, a list of argument address to be passed to the called routine, the address at which the return value is to be placed, and the type of calling convention used.

The calling convention is one of the 7 conventions supported by this package and is used to determine how the argument and return values are processed. Fortran and system calls are assumed to be call by reference while DAP C calls are call by value.

CallingC

This macro is used to call a procedure that has been compiled using DAP C. It accepts 4 arguments: the name of the procedure to call, the type of C call to use (i.e. the number of arguments that are passed in registers, a list of local registers to protect across the call, and a list of arguments to pass to the called procedure.

CallingFort

This macro is used to call a procedure that has been compiled using AMT's Fortran. It accepts 4 arguments: the name of the procedure to call, a list of argument address to pass to the called procedure, the address of the location in which the return value is to be stored, and a list of local registers to protect across the call.

CtoFinterface

This macro provides an interface between DAP C and AMT's fortran libraries. Each library module that is to used from DAP C must have an entry in the DAP C library. This entry is used to translate between the Fortran and C calling conventions. The macro accepts two arguments: the name of the Fortran function to be called, and a list of the argument types passed from DAP C.

Each element of the list of argument types specifies the type of the corresponding argument. The letter V indicates that the call is by value and the letter R indicates that the call is by reference. The necessary translations is performed and the Fortran function is then called.

The convention used in naming such DAP C function is to prepend "C_" to the Fortran function. For example if a DAP C program needed to call the Fortran graphics function Start_Graphics it would instead call the function C_Start_Graphics which would perform the necessary conversions and call the desired Fortran function.

EnteringC and ExitingC

These macros are used by the DAP C compiler to provide the necessary linkage between DAP C functions. The calling convention used is controlled by this macros so that efficient code can be generated.

PushAddr

This macros pushes the address specified into the system stack. It accepts two arguments; the address to be pushed and a working register in which to compute the address. The address includes both an offset field and an index register.

PushContents

This macros pushes the word specified into the system stack. It accepts two arguments; the word to be pushed and a working register used to store the value to be pushed. The specified word may be a register or a memory address containing an offset field and index register. If it is a register, that register's contents is pushed onto the system stack, otherwise the contents of the memory word at the specified address is pushed onto the stack.

GetWordArg

This macro is used to retrieve a word passed using the fortran convention. It accepts two arguments; the number of the parameter to be retrieved, and the register in which to store the value.

3.2 Macro support of debugging in DAP C

During the development of this package it was necessary be able to pause execution so that the system state could be read. In order to allow for control of these pauses without the need to edit or recompile the system we developed a uniform interface for controlling such pauses. The method employed uses AMT's pause mechanism under the control of a local decision based on global flags. The user can change the global flags during run time and thereby control which pauses will be enabled and which will be disabled.

DiagnosticPause

This macro causes an APAL program to pause if the specified global switch is set and to continue normally otherwise. It takes two arguments; an identifying number to display if a pause is required and the variable to determine if the pause should be executed.

3.3 Macro support for APAL programs

PushReg and PopReg

These two macros push a register onto the system stack and pop the top of stack into a register. They each accept one argument, the register number to be pushed or popped.

SaveRegs and RestoreRegs

These macros push or pop a list of registers onto or off of the system stack. They each accept one argument, a list of registers to push or pop.

GetFirstStackValue and PutFirstStackValue

These two macros function much like PushReg and PopReg; however, they do not change the number of items in the system stack. Each takes one argument, the register number involved in the operation. GetFirstStackValue copies the value in the top of stack to the specified register; the stack is not changed. PutFirstStackValue writes the specified register at the top of stack location. The number of items in the stack is unchanged; however, the value that was at the top of stack location is overwritten.

GetSecondStackValue and PutSecondStack

These macros functions similarly to GetFirstStackValue and PutFirstStackValue. They each accept one argument, the register number involved in the operation. These functions copy or replace the second value in the stack.

StoreArgs and LoadArgs

These macros write or read a list of arguments to or from a contiguous areas of memory. Their intended use is to setup or read arguments used when calling procedures. Each takes two arguments; the first is a list of memory addresses, the second is the address of the contiguous memory area. Both addresses allow offset and index register specification. Data is transferred between the addresses specified in the first list and the contiguous memory area.

GetAsciiValue

This macro converts a character to its ascii value. It takes one argument, the character to be converted and sets the assembly variable AsciiValue to the integer value that corresponds to the specified character.

MakeCstring

This macro converts an ascii string into a packed version of that string. It accepts two arguments, The first is the address of the location to hold the packed string and the second is the ascii string to be packed. It assumes the string is a *simple* format string used by a C printf treating the \ as an escape character.

3.4 Support of Integer Division for DAP C

Support of Integer Division for DAP C is accomplishes using two subroutines. One performs signed division and the other unsigned division.

divsi3

This routine performs signed division using the same rules a C. All values involved in the computation are represented by 32 bit fields passed in registers. The dividend is passed in register m1 and the divisor in m2. The quotient is returned in m1 and the remainder in m2.

udivsi3

This routine performs unsigned division using the same rules a C. All values involved in the computation are represented by 32 bit fields passed in registers. The dividend is passed in register m1 and the divisor in m2. The quotient is returned in m1 and the remainder in m2.

3.5 Interface to Low Level Graphics Routines

The following routines provide access from DAP C to the Low level Fortran graphics functions provided by AMT. These routines assume that the caller will use normal C calling conventions passing fixed size arguments by value and arrays, strings, and place holders for return values by reference. The functions currently supported are C_Start_Graphics, C_Stop_Graphics, C_Set_Lut, C_Put_Lut, C_Get_Lut, C_Put_Frame, C_Start_Sequence, C_Stop_Sequence, and C_Clear_Screen.

4 Filling enclosed regions on the DAP

Fill is a procedure that works in conjunctions with other graphic primitives to provide a *basic* graphics interface. Fill *colors* closed regions of a plane based on the specifications provided by the user. The algorithm is adapted specifically for the DAP's fast IO interface to a color monitor. Fill performs the specified functions using a combination of serial and parallel techniques adapted for the DAP 510's configuration.

The screen is a two dimensional grid of 1 Meg pixels (1024 by 1024). The data for each pixel is stored in eight bits, providing 256 possible values for each pixel. The DAP represents this data by grouping the screen into 1024 (32 by 32) *dapels* (our term) each of which contains 1024 (32 by 32) pixels. The 8 bits for each pixel are stored vertically in the memory stack resulting in one pixel's worth of information being spread over 8 planes of memory, with each plane contains one bit of information for each pixel in that dapel. On a DAP 510, a machine with a 32 by 32 processor array, one plane of one dapel is easily manipulated within the the processor array.

The algorithm is initiated by *seeding* one or more pixels in one or more closed regions. These points form the start of the fill pattern. The filling proceeds *out* from these points until all pixels contained in the same closed regions have been filled. The filling is performed one dapel at a time. If more than one dapel requires filling, the dapel with the largest number of seeds is processed first. This scheduling is performed using a priority queue.

Each dapel is filled using a parallel algorithm. This algorithm begins at all filled pixels within the dapel and, in parallel, fills all orthogonal neighbors of each of these pixels. If a neighbor is a part of a boundary it is not filled. This process continues until convergence (i.e. until the filled pattern does not change from step to step). Once the dapel has been filled, the neighboring dapels are checked to see if the *edges* of this dapel will effect the *edges* of its neighbors. If any pixel on a neighboring dapel is affected, that pixel is added to the seed information. Once all neighboring dapels have been updated, the next available dapel is filled.

4.1 Interface to user

All access to fill and related operations are obtained through function calls. These calls all return 0 if no error is noted and non-zero otherwise. This non-zero return contains information as to the exact cause of the failure.

InitFill

Called once when the graphics packages is initiated. It initializes the priority queue and sets the fill state to idle. The error return indicates the state that fill was in when init was called.

SetFillMode and SetUnfillMode

These functions set the filling mode and should be called before any pixels are seeded. SetFillMode sets the mode to fill using the *fill color* while SetUnfillMode sets the color to the background color providing an approximation to erasing a filled region. Note that in both cases the region is actually colored by a predefined color, it is never actually erased. The error code indicates the state that fill was in when the function was called.

SeedPixel

This function places a pixel into the the queue. The pixel value is passed as two arguments, the x coordinate as the first and the y coordinate as the second. The coordinate system is screen oriented with the upper left pixel being 0,0 and the lower right pixel being 1024,1024. An error code of -1 indicates that a boundary pixel was specified for seeding, otherwise the state that fill was in is returned.

PerformFill

This function processes all the seeded information and fills (or unfills) the regions as specified. An error code indicates the state that fill was in when the function was called.

AbortFill

This function empties the priority queue and resets fill's state. An error code indicates the state that fill was in when the function was called.

4.2 Priority queue macro and function interface

The priority queue used by the fill routine is accessed via macro that implement a call to the necessary functions. Currently, the macros perform only a subroutine call and assume that all linkages and returns are handled directly by the code. The calling conventions for the queue are non-standard and designed to reduce the number of cycles required to access the queue. The queue is limited to 1024 elements each with an id between 0 and 1023.

initPQ and initPQop

This macro/function pair is called once to initialize the priority queue operations. The queue is set empty by this call.

incpriPQ and incpriPQop

This macro/function pair is called to enter an item into the queue as well as increase the priority of an item already in the queue. The function accepts 2 arguments, the element name in register m1 and the amount to increase the priority by in register m2. It returns a zero in register m1 if the operation is successful, otherwise it returns a non-zero value. If the element being incremented is not currently in the queue, it is inserted with the priority specified in the call.

delmaxPQ and delmaxPQop

This macro/function pair is called to remove the highest priority element from the queue. The element id is returned in register m1.

emptyPQ and emptyPQop

This macro/function pair is called to test if the queue is empty. It returns, in register m1, a 0 if the queue is not empty, and -1 (i.e. all bits set to 1) if the queue is empty.

5 Reading and writing the DAP's memory from the host

The DAP 510's memory is a 3 dimensional array of bits. The array is 32 by 32 by depth, where depth is determined by the amount of memory installed in the system. The functions described in this section are used to interface to such a memory and provide read and write capability between C and the DAP.

All these functions are available in C on the host SUN'S system. The DAP memory is modeled in the C program using the three dimensions labeled rows, columns, and tunnels. The number of different tunnels is the number of 32 by 32 bit planes available on the DAP. The DAP's memory may be transferred to and from the C image as required.

ClearDapMatrix

This function clears the DAP memory and sets bounds for the number of errors or warnings for which the user will be notified. It accepts three arguments; the first is the number of planes of memory to be used, the second is the bound on the number of errors, and the third is the bound on the number of warnings.

GetDapMatrix Length

This function returns the number of significant words currently stored in the SUN'S image of DapMatrix. This function is used to determine the amount of data that must be transmitted to the DAP.

SendDatatoDap

This function sends the data stored in DAPmatrix to the DAP. It accepts one argument, the name of the area in DAP memory to which DAPmatrix is to be copied.

GetDataFromDap

This function reads the data stored in the DAP into DAPmatrix. It accepts one argument, the name of the area in DAP memory from which DAPmatrix is to be copied.

PutValueIntoTunnel

This function writes a value into a tunnel of DapMatrix. It accepts 5 arguments; the first three are the indices of the row, column, and tunnel at which the first bit of the value is to be stored. The fourth is the value to be stored and the fifth is the number of bits to be stored. These bits are selected from the low order end of the value.

GetValueFromTunnel

This function reads a value from a tunnel of DapMatrix. It accepts 4 arguments; the first three are the indices of the row, column, and tunnel at which the first bit of the value is to be stored. The fourth is the number of bits to be stored. These bits are written into the low order portion of a word which is then returned to the caller.

PutValueIntoRow and GetValueFromRow

These functions work as do their counterparts for operating on tunnels of DapMatrix. Their arguments are exactly the same with the only difference being that the value is oriented along a row rather than along a tunnel.

PutValueIntoColumn and GetValueFromColumn

These functions work as do their counterparts for operating on tunnels of DapMatrix. Their arguments are exactly the same with the only difference being that the value is oriented along a column rather than along a tunnel.

VectorToDapMatrixRM

This function writes a list of values into the DapMatrix oriented along successive tunnels. It accepts 5 arguments; the first is the vector of values to be written, the second is the number of elements to be written, the third is the number of bits to be written per value, and the last three are the indices of the row, column, and tunnel that will contain the first value.

VectorFromDapMatrixRM

This function reads a list of values from the DapMatrix oriented along successive tunnels. It accepts 5 arguments; the first is the vector to contain the values read, the second is the number of elements to be read, the third is the number of bits to be written per value, and the last three are the indices of the row, column, and tunnel that will contain the first value.

6 Drawing Lines on the DAP

The line drawing capabilities are supported by a two level drawing algorithm. The lowest level draws straight line segments and can be called directly or through the curve drawing routine. The higher level is an algorithm that draws Bezier curves and uses the straight line drawer to put a piecewise linear approximation to the specified curve on the screen.

6.1 Drawing Straight Lines on the DAP

The Draw-line function is a primitive of the graphics system. Given a plane number and the coordinates of two points, the function will draw a straight line between the points on the specified plane. The input points can be selected by the mouse, loaded from a file, or passed by other functions (e.g. Draw-curve). The coordinate system used is the same as the conventional system, i.e., the lower left corner of the screen is 0,0 and the upper right corner is 1023,1023. The algorithm is designed to make use of the parallel properties of DAP 510's configuration.

Initially, the algorithm employs a division table to compute the slope of the line. The value of the slope is then used to determine if the line should be treated as a horizontal or vertical segment. If the slope is less than or equal to 1, the segment is treated as horizontal (i.e. a column scan of dapels is employed); otherwise the segment is treated as vertical (i.e. a row scan of dapels is used). A line segment can be contained entirely within one dapel or spread across two adjacent dapels.

The pixels on each line segment are drawn in parallel but successive segments of a line are drawn serially. This approach yields a parallel/serial hybrid algorithm that is extremely efficient on a grid based SIMD processors such as the DAP. The maximum time to draw an arbitrary line is only 4000 machine cycles.

Access to Draw-line is obtained through function call Drawline_d. Drawline_d has five arguments (x1,y1,x2,y2,plane_no), where <x1,y1> and <x2,y2> are two input points and plane_no indicates the plane on which the line will be drawn. The function does not generate any error message.

6.2 Drawing Bezier Curves on the DAP

The Draw-curve function draws a cubic bezier curve based on four control points. The control points can be the pixels selected by the mouse or data loaded from file.

Drawing a curve consists of two steps. In the first step, the function performs repeated linear interpolation until the control polygon approximates the Bezier curve. This results in the generation of 97 control points for each curve. The algorithm for this step is a parallel/serial hybrid that is very efficient on grid based SIMD processors.

In step two, Draw-line is called to draw the 96 line segments which approximate Bezier curve.

The access to Draw-curve is obtained through function call Drawcurve_d. Drawcurve_d has eight arguments (x1,y1,x2,y2,x3,y3,x4,y4), where <x1,y1>, <x2,y2>, <x3,y3> and <x4,y4> are four control points. The function does not generate any error message.

7 Graphics Subsystem

The graphics subsystem runs on the DAP and provides a mouse driven interface that allows users to construct line drawings composed of straight or curved lines, fill (or unfill) the regions defined by these lines, and to place text on the screen.

7.1 Initializing the graphics subsystem

Before using the graphics system the screen must be initialized, fonts must be loaded, and the real time mouse interface started. The following routines are used to accomplish this initial setup.

Main

This function is the main entry when running the graphics system. It is written in DAP C. The function calls Dap_init, InitFill, Send_chartab_to_dap, and several Fortran subroutines to initialize the system. It sets the color table for the screen and transfers the necessary character fonts. It calls Set_cursor to initialize the cursor (at the center of screen) and then calls OpenMouse and Mouse_ctrl to initialize the mouse interface.

Dap_init

This function is called by Main to initialize the screen memory. It returns the address of the screen memory.

Mouse_Ctrl

This function is the core of control of the graphics system. It is written in DAP C. The function controls the mouse and provides the interface to the menu allowing selection and execution operation.

7.2 Menu

The menu interface is controlled by seven functions. These functions work together with the real time mouse interface to bring a menu to the screen, move about on the menu, and select an action from the menu. All actions required of the menu interface are controlled from these functions. The font used for the menu is Roman.

Display_menu

This function is called by Mouse_ctrl to display the menu on screen at the position given by arguments. It displays the menu frame, menu contents, and cursor. The function returns the address of the current cursor position; It calls Set_cursor, Display_string, and Fill_menu.

Erase_menu

This function erases the menu or warning displayed on screen and recovers the users region that had been overwritten by the menu. The length and width of the menu or warning are passed by arguments. The function is called by function Sel_Funcs and function Mouse_ctrl.

Update_menu

This function is called by function Mouse_ctrl to update the menu when the mouse is move to a new selection. It calls Unfill_menu to unfill the region in the menu corresponding to the previous operation and Fill_menu to fill the region corresponding to the new operation. It also changes the function flag.

Fill_menu

This function is called by Display_menu and Update_menu to fill the region in menu corresponding to the selected operation.

Unfill_menu

This function is called by Update_menu to unfill the region in menu corresponding to the previous selected operation.

Sel_Funcs

This function is called by Mouse_ctrl and determines the desired operation based on the current cursor position. It returns the value of function flag. The function will call Erase_menu.

Rel_Funcs

This function is called by Mouse_ctrl to release the current operation and return to idle state.

7.3 Cursor Control

The mouse interface displays the current cursor position on the screen in real time. The following set of routines are responsible for maintaining the cursors screen position in conjunction with data received from the mouse.

Set_cursor

This function is called to set the cursor at a particular position of screen. The initial coordinate of the cursor is 0,0. To move the cursor, the distance between the new cursor position and the current cursor position, i.e., delta x and delta y, should be given using arguments. Usually, the delta x and delta y are obtained from the mouse. The function returns the address of the current cursor position.

Erase_cursor

This function is called to erase the cursor from the screen.

7.4 Text and Fonts

Send_chartab_to_dap

This function is called by Main to send the characters fonts to the character table.

There are seven different fonts that may be used to display characters in the system. They are Default (12*9 pixels), Roman (18*11), Roman bold (19*11), Gallant (15*12), Courier (24*13), Courier bold (24*14), and Roman32 (32*30). Each font is stored in an individual file in which each character consists of 32 32-bit hexadecimal numbers. Fonts are read from these files and then sent to character tables. Although each character is stored in a 32*32 dapel, most characters occupy only a small portion (i.e. about 1/4) of the upper left corner of the dapel. The size of each font is hard coded in the function Text_Ctrl. Based on the value of the argument (i.e. font_no) the system can determine the horizontal distance between two characters. The vertical distance between two rows of characters is determined by the user usually through the mouse interface.

Display_string

This function displays a character string on the screen, starting at the current cursor position. The length, address and font of the string are given as arguments. The function returns the address of the current cursor position. This function calls Text_Ctrl.

Text_Ctrl

This function is called by Display_string to display characters one by one. The ASCII code and font of the character to be displayed are passed as arguments. The function returns the address of the current cursor position. The function calls Display_char and Set_cursor.

Display_Char

This function is called to display a character or cursor on the specified screen plane. Using masks, the function can display the pixels of the cursor or character in parallel. Among the five arguments, the first one is shared by cursor and characters operations. The second and third are used only for displaying the cursor and the fourth and fifth are used only for for displaying characters.

Load

This function is written in DAP C and called by Mouse_ctrl. This function provides a means for a user to construct a control file that can be executed within the graphics interface. The interface reads and executes the commands until a 'q' is encountered or the file ends. Each operation occupies a line: the first character identifies the graphic function to be executed (i.e. l-line, c-curve, f-fill, u-unfill, and s-string) and the remainder of the line contains the necessary arguments. For example, the four numbers following an 'l' would be the two points of a line.

Access to Load is obtained through the function call load_in. The user will be asked to input the name of the command file and an error message will be returned if the file cannot be found or an invalid operation is found in the file.

7.5 Low level graphics support

The following three functions are low level support services provided to all components of the graphics subsystem.

Display_warning

This function is called by Mouse_ctrl to display a warning on screen when a Clear-screen or Exit operation is selected. The function returns the address of the current cursor position. The function will call other functions, Set_cursor and Display_string.

Clear_screen

This function is called by Mouse_ctrl to clear planes 0-2 of screen memory. This results in a display containing only the cursor.

Clear_planes

This function is called by Mouse_ctrl to clear a plane of screen memory. The plane number is passed as the argument.

8 The GDD C compiler

8.1 Introduction

GDD is a C compiler for the AMT DAP510 massively parallel mesh-connected computer. It is a derivative of the GNU C (*gcc*) compiler from the Free Software Foundation, and thus public domain [1].

The motivation for the compiler was the lack of a good systems programming language for the DAP. The only languages available have been Fortran, which has never been used as a systems programming language, and an assembly language, which has been found to be unusually restrictive and time consuming. A parallel extension of a language like C would be the most important tool that a group like ours could get for developing software for the machine. This compiler is meant to be a step in that direction.

The version of GDD described here is a port of a serial C compiler to the scalar processor of the DAP. While using a serial language on a parallel machine does not appear to make much sense, experience shows that large portions of code that take much effort to write in assembly are control-oriented and sequential in nature. Since we have gone to great pains to allow for relatively simple mixing of C and APAL code, the combination can be as efficient as the user desires.

[1]The legal term here is 'copylefted'

8.2 Using GDD

The structure of the compiler

Like gcc, GDD is a multi-pass compiler system that consists of several programs, of which the actual compiler is only one. Here is a list of the programs in the sequence they are run:

Control program `DAPBIN/gdd` A program whose sole purpose is to run the appropriate following programs, and manage the temporary files created. Only slightly changed to account for new filename suffixes, and hardcoded paths.

Preprocessor – `gdd/lib/cpp` Completely unchanged. Produces a `.h` file.

Compiler – `gdd/lib/cc1` Multi-pass compiler that produces assembly code, with the .da extension, and separate file with data declaration (because of the single pass assembler).

Assembler prepass – `gdd/lib/fixdatafile` An Awk script that reorganizes the data declarations into appropriate sections.

Assembler – `DAPBIN/dapa` The DAP APAL assembler.

Linker – `DAPBIN/dapa` Same programs doubles up as a linker.

Executable switch setter – `DAPBIN/dapopt` Sets various switches, of which the control program only sets whether the program is to be run on the hardware or simulated by a simulator program.

Running the compiler

The GDD compiler operates in general identical to the GNU C compiler. The following machine dependent options for cc1 have been implemented:

`-mstrict-tests` Perform tests for overflow after subtraction needed for the comparison statement. Default setting.

`-mlazy-test` Do not generate those overflow tests.

-mdap510 Generate code for DAP510. Default setting.

-mdap610 Generate code for DAP610. Currently does nothing.

Switches forwarded to assembler and linker

The gcc compiler system forwards several switches on to the Unix assembler `as`, and the Unix linker `ld`, which clearly are of no use with the DAP system software.

Instead, the following switches are passed to the assembler:

`-e` Generate external references and section listing in a `.lst` file.

Generate external references and attribute listing.

`-Ln` Generate source listing of the given level n.

`-tn` Generate assembler source trace statements of level *n*.

The following switches are passed to the linker:

`-mn` (specified as `-Xn`) Generate consolidator map.

`-sn, -s+n` Set/increase the DOF stack record.

`-A` Set simulator flag.

File system

The filesystem for the GDD project all resides in the subtree `/u7/planchet/halldors/gdd/`, which we refer to here as simply gdd/. Following is a description of the main directories.

`gdd/src` Source files *.c*, and the *md* machine description file.

`gdd/include` Include files.

`gdd/bin` Executables - gdd

`gdd/lib` The subprogram executables (cc1, cpp, scripts)

`gdd/testbin` Development executables

`gdd/doc` Documentation

`gdd/test` Test harnesses, results, and testing programs.

`gdd/test/files` Test files.

`gdd/othermach` Descriptions of other machines.

`gdd/sys` Glue files for connecting with DAP.

`gdd/gnubin` Executables used in the generation and compilation of the compiler.

Source changes

The files src/md and include/tm.h contain almost all the description of the machine and machine dependent macros that are used to generate a correct code generator. Their configuration is described in the next section. Unfortunately, due to the unusual architecture, instruction set, and restrictive system software, some changes to the "meant-to-be-machine-independent" source had to be performed. We did the utmost to keep those patches as limited and localized as possible; see the file gdd/doc/source.diff for the textual differences.

The following is a complete list of those alterations.

expr.c, stmt.c The macro GET_MODE_STORAGE_SIZE(Mode) was defined to translate from bytes to storage units, and used everywhere in the file in place of GET_MODE_SIZE.

expr.c A function call to setup_new_frame(bytes pushed) (see the new file dap-extra.c) added before a new activation record is written to. Necessary because of the unusual activation record structure of the DAP.

stor-layout.c All references to BITS_PER_UNIT replaced by BITS_PER_BYTE (found in tm.h).

stor-layout.c The definition of the macro GET_MODE_ALIGNMENT(Mode) changed, due to the different alignment requirements of the DAP.

toplev.c, varasm.c All data declaration written to a separate file, with the extension .data.

toplev.c To compute the stack size properly, a global counter total_locals_in_planes is defined, and referenced in tm.h.

toplev.c Instead of the ".s" extension, assembly file have the extension ".da".

8.3 Work done on GDD

```
  Changes to GNU CC in porting to the AMT DAP 510
  -----------------------------------------------

--- Factors that made the porting job hard ---

I. --- DAP Architecture ---
  * Unusually limited addressing modes available.

II. --- DAP system software ---
  * Unorthodox, non-standard, e.g. assembler single pass.

II. --- Problems with GCC compiler ---
  * Lack of 'lint'-ability
```

* Its treatment of nested activation records

* Variables had to be declared before use
  : Use separate file for data declarations

* Unorthodox scoping rules for variables, making sharing globals variables difficult
  : Postpass to place each global variable in a separate section

* Assumption of compiler that the size of a type is the same as the size of the storage it uses
  : Change definitions to reflect that 'char','short',and 'long all use 32 bits of storage, while their computational size remains 8,16 and 32 bits.
  (Still not completely fixed, e.g. short)

* Addressing globals

* Register usage, way different from compiler's assumptions
  : Stack thought of very differently; usually, most things can be referred to using the stack point, but on the DAP, no negative addressing can be made.
  : Wound up having to tie up three register:
m6 : Activation record and function argument address pointer
m7 : 'Frame pointer' = Pointer to local variables
m13 : stack pointer
  plus m5 was used on function calls, and m12 when referring to globals.

* Function calls require highly unusual setup
  : Special treatment for nested function calls
  : New routine: call to setup-new_frame added in expr.c
  : Difficulty remaining compatible with several different function call formats. E.g. both AMT formats did not allow for the possibility of variable number of arguments.

--- Fairly minor changes to source ---

* Output of string constants
  : Special routine, string_constant_output() added.
  : Special care needed to be taken care of since an assembler source line may not, under any circumstances, contain more than 79 characters.

* Sectional system incompatible with compiler's assumptions of

```
assemblers
  : Some taken care of in supplied macros
  : Certain 'end's added to source

* DAP specific filenames, and non-standard locations of executables


--- Problems fixed in mach.description file ---

* Switch statement -> Changing from address table to jump table
```

8.4 Current status of compiler

Original restrictions

Floating point operations were intentionally left out. Given that the processor has no floating point instructions, we hold the belief that most users are better off with a fixed-precision library. Floating or fixed-point was considered a future extension.

Current bugs

Short integers Any use of shorts causes the compiler to crash. This is caused by the expectation that storage unit size is equivalent to data type unit size. Requires changes in some source files to fix, but since the datatype is really not of much use it has not been a high priority item.

Bitfields Bitfield extraction produces an incorrect code.

Optimization Some non-redundant statements are optimized away in code with switch statements. Also when making comparisons, a register sometimes gets clobbered when optimized.

Features to be tested or known to be missing

Function pointers Not likely to work without some ingenuity.

Unions Not fully checked.

Recursive functions While recursive calls works per se, since the APAL compiler requires a fixed stack size declared, the level of recursion must be anticipated.

Wish list

- More automated testing system.
- More complete test harnesses.
- String library.
- Fixed-point or floating-point library, possibly in-lined.
- Interface with profiler.
- Debugger possibilities.
- Increased use of parallel operations
- New assembler
- Easier ways of interacting to and from host.

A Project log

A.1 Work Finished

```
Worklist for gdd compiler
-------------------------

III. Jobs that are finished
    ----------------------

1. Check if function calls work. (-> YF) Aug 10
[Aug 14]  Having tried the following successfully:
1) function "test" calls "sub";
2) "test" calls "sub1" which in turns calls "sub2"
  But failed the following:
1) when there are more than 4 arguments, since
   negative offset is generated for any argument
   after the 4th one.
[See #10]

2. Check if calling & linking library routines work. (-> YF) Aug 10
[Aug 14] Having passed a test with "libcall1.c" program which
 issues a call to "Get_PlaneCnt" which is an entry
 point defined in "~dap/local/lib/DAP.dl".
```

Other routines defined in DAP.dl need to be tested
later on. (as described in number 4 below)

3. See if using asm() works, esp. for pause and trace. (-> MMH) Aug 10
[Aug 17] Works fine. See tst18.c

13. Set up gdd fully : connect with Sun4's dapa. (-> MMH)
[Aug 11] dapa now run on planchet. Library directory not connected.
[Aug 12] Symbolic link from /usr/lib/dap to current Sun4 directory.
[Aug 13-17] dapa called from gdd, both as an assembler and as a linker.
All useful options passed through to it. (Need to document)
dapopt called when option -A is given
[Aug 17] Quite compilation flag now also sent to dapa

14. Check structs, unions, constant strings (-> YF)
[Aug 14] Having passed test for "struct1.c"; "struct2.c"
uses pointers, so needs to wait for the fix-up
of compiler's handling pointers.
[Aug 16] Passed "struct2.c".
Passed "union1.c" and "union2.c".
[const. strings do have problems. More on that later]

27. String constants in Apal code (MMH)
[Aug 23] Fixed. Code in varasm.c altered. (MMH)
[Aug 24] '%' in comments is now escaped (MMH)

12. Have the compiler use other calling conventions
Fortran, Rutgers DAP conventions - Add keywords to the C.
[Cancelled. The Calling Macros have been made intelligent instead]

8. How to interface Sun & Dap strings
[Not our business - Handled by Don & Sizheng]

16. Behavior of automatic variables, including scalars and arrays. (YF)
[Aug 21] passed "autovar1.c" which tests the allocation of
space for the variables defined in blocks. Either
having "auto" specified or not does generate the same
code. (having "auto" or not does not matter when vars
declared in blocks.)
"autovar2.c" tries to use a variable in a block while
that var is defined in another block; error message
issued by the compiler.
passed "autovar3.c" which allocates arrays in blocks.
[Aug 31] Handling of automatic variables changed radically.

17. Check the allocation of space when scalars and arrays are mixed
    in the declaration. (YF)
[Aug 21] 5 files, "declare*.c", are used to observe how the
 (locally) declared variables are assigned space.
 It is surprising to find that scalars are always
 aligned w.r.t planes, while arrays are allocated
 4 words beyond its previous variable (scalar or
 array).

 In addition, there is still a bug in the allocation of
 space if the first declared variable is an array. See
 "declare4.da". Combining with our previous problem with
 the initialization for arrays, the real problem is that
 space assigned to arrays is not aligned correctly.
[Aug 31] Handling of automatic variables changed radically.
- Stack alignment always by words now
- Space for arrays allocated correctly (should check further)

29. Add the STACK command to Apal output (MMH)
[Sep 23: Done --> See 35]

31. Change CT : CT [dof-file] < testfile
[Sep 23 : Done]

34. Make sure commented data string doesn't make line too long
[Sep 23: Done]

30. Version control, update notes etc. (MMH)
[Sep 24: Set up file ~dap/local/doc/gdd-rel-notes]

* 26. Write a _main() routine (that calls Init&Terminate allocate) (MMH)
[Sep 27: Done]

24. Code generation for mod or remainder (%) operation. (MMH)
Always call "_divsi3" or "_udivsi3" to implement the division
and % operations. Quotient returned in m1, while remainder or
mod returned in m2.
[Sep 27: Mod&div handled specially in produce_function_call()]

19. Bitfield. (feature test)
[Aug 22] I'm pretty sure that the code generated for "bitfield*.c"
 is not correct. See bugs.log. (YF)
[A current bug (Nov 17)]

21. Switch statement. (feature test)

```
[Aug 22] The code generated for switch has two bugs, see bugs.log.
[Oct 3] These two bugs fixed (see bug #14). Test remains insufficient.

25. Split data section into: globals, statics & constants  (MMH)
    -> Also: Place each global into a separate data section,
(with elements of each constructor together)
- Means writing awk programs
- Then try sharing, e.g. the screen, as an extern structure
    [Nov 18,19: Done. Each global variable in a separate section.
Remaining variables in a single 'statics' section.
- gdd altered to automatically do the conversion.
- Currently awk script + shell script driver
- Could extend so as to merge data decl. file with .da file.]

22. Break and continue statements. (feature test)
[Aug 23] passed test for "continue*.c", which have continue
 statements in for/while loops.
[Dec 5] break used heavily in graphics code and work fine.

35. Make MODULE_STACK_SIZE depend on the number of functions in module
[Nov ~30] Done.

Opt iii) Redundant statement when producing a table jump
[Dec 5] Done. See bug#29.
\newpage
II. Work Log : a) Everything about function calls
--------------------------------------------------

7. Work on calling with more than 4 arguments. Use 'type=C4' w/ Calling. Aug 10


Aug 29: Work on function calls.
tm.h:
  - Macro STACK_BOUNDARY changed from 1024 (32 words) to 32 (1 word)
That probably causes the 32 in "add m5, 32"
and the 32 words diff. between local variables
  - Changed STACK_POINTER_REGNUM from 5 to 7
  - FIRST_PARM_CALLER_OFFSET commented out
  - Add update stack ptr statement to fn.prologue, for local vars
Aug 30:
  - STACK_POINTER_OFFSET commented out (for efficiency)
  - STARTING_FRAME_OFFSET commented out : since we set up SP in Entering
  - Define PUSH_ROUNDING to convert from bytes to storage-units (i.e.words)
This does not handle the block mode - See about that later
        [Converted back: PUSH_ROUNDING only good for machines with push
```

```
instruction]
  - Check what happens if RETURN_POPS_ARGS is set to nil.
[It works: SP gets decremented same amt as it was incremented]
  - expr.c changed: All refs to GET_MODE_SIZE get an added byte->storage
unit conversion
Aug 31:
  - int_size_in_bytes, and size_in_bytes already produce storage units
  - the macro SETUP_NEW_FRAME written, compiled, and tested
  - stmt.c: GET_MODE_SIZE altered to GET_MODE_STORAGE_SIZE
Sep 1:
  - STARTING_FRAME_OFFSET set to 8 + {some constant, preferably #args}
  - SP incremented by SIZE - STARTING_FRAME_OFFSET in FUNCTION_PROLOGUE

What is correct now:

  - All setup a function call done by the compiler, correctly.
  - Stack pointer updated to account for locals, at the beg. of the function.
  - Correct stack pointer used (not m5 - was never really tested).
  - All alignment, of arguments on stack and of locals, now correct.
  - Locals now at (almost) correct place, right after arguments.

What needs work:

  - Call SETUP_NEW_FRAME for div,mod and related builtin library functions.
  - Adding #args to STARTING_FRAME_OFFSET.
  - General problem of handling variable # of arguments.
 (- integrate.c may need to be altered in the same way (e.g. GET_MODE_SIZE) )

*** New version: ***
Sep 1:
(tm.h)
  - m13 made to be the stack pointer (STACK_POINTER_REGNUM, FIXED_REGISTERS)
  - m7 made to be base for locals (ARG_POINTER_REGNUM)
  - FUNCTION_PROLOGUE: Copy m7 into m13, and subtract #locals from m7
(expr.c)
  - Save Arg-pointer into 0..6 (m6)
(aux-output.c)
  - Rename the registers used in the calling prologue
  - Add a "rr m7, m13"
Changed again:
  - m7 is the *frame pointer* = base for referencing locals
  - m6 is the argument pointer
  - expr.c and aux-output.c are now just about independent of that.
  - FIXED_REGS = m6,m13 - frame pointer need not be
  - STARTING_FRAME_OFFSET = 0
```

```
Corrected:
  - Variable number of args now handled.
Problems:
  - fn3.c crashes when trying to optimize (signal 6)
  - tst9.c crashes in compilation (not fn1.c, tst4.c) (signal 6)
  - m7 is not being used: locals accessed by horrendous complications
  - Arrays handled strangely
  - Need to pinpoint the actual bugs
  - Think more about where to store m7

Sep 11:
  - Optimized function calls miraculously working...

Sep 20,21:
  - bug2,bug3,bug4 now all working
  - see explanation in bugs.log
  - the offset of args to arg ptr, increased to 9 (leaving the 8th for ret val)

[More info here needed about fixes dealing with nested function calls.]


 ====== "Somebody else's department" ======

4. Link with malloc routine. (-> YF)
[Aug 16] passed test using malloc and free,
 and test using malloc, realloc and free.

 Sizheng's C program for testing allocation library
 needs to use printf, so it will be on hold until
 printf is available.  We tried to think of an alternative
 to do the testing, but it is too complicated.
28. Test Sizheng's malloc test routine ~wei/dap/allocatetest.c (MMH)
[Aug 24] Compiles correctly.
[Sep 22] Optimizing crashes ---> see bug6.c

[Oct??] Malloc tested and running.
[Nov17] Optimizing no longer crashes

11. I/O routines - Interfacing w/ printf, scanf
[Oct??] printf,scanf connected
```

A.2 Bugs Fixed

```
Fixed bugs
----------

6. Wrong memory addresses: plane offset [Memory offset] Aug 10
Use full spec. of offset in address:
  i.e. <plane>..<word>
Only problem for certain opcodes. Rethink when doing for proc.array
    - Fixed Aug 14: Changed the PRINT_OPERAND_ADDRESS macro in tm.h  (MMH)
    - Another fix Aug 17: Symbols should not be preceded with a plane (MMH)

9,11. Array initialization error [Any arrays,const] Aug 17
tst1.c (Hello,world) does not assemble.
Attempts to get its address by rasc-ing a label:
  - label goes to the source file not data file as it should
  - dapa appears recognize the string in the data section, not certain
[Fixed Aug 22 : Sent to datafile now. varasm.c changed MMH]

10. Passing character strings from Sun C to DAP C [strings,interface]
[Irrelevant here] Aug 17

3,12. Allocation of automatic arrays [Arrays,auto] Aug 10,21
whenever an
array is declared (locally) as the first variable, we are in
trouble, for it will overlap the formals.

7. Arguments on stack not done correctly  [Fn.arguments] Aug 10
Need to set up new activation record
Negative offset generated. (Aug 14, YF)
[Sep 1 : Function call and stack handling completely overhauled]

17. Constant shifts expand to variable shift [shift, constant] Sep 1
- Occurs also for div by const and mult by const
- See bug5.c (also tst9.c)
[Sep 3: Changed order of alternative constraints for shifts in md]



"The Crashes"

8. Increment operator on a global variable [globals] Aug 17,24
- Has to be preceded by an assignment to the variable
- Can be avoided by using +=
- Occurs when not-optimizing: doing stupid reg-allocation
```

- See bug2.c
20. Function calls crash Sep 1
- Any call crashes. Happened after overhaul
- see bug3.c
19. Division crashes [Division] Sep 1
- see bug4.c

[Sep 21:
Apparently all these bugs were caused because of the incorrect recognition of the operands of an addition statement (one that was generated internally, by the compiler, for these various reasons). Because the description of 'movsi' preceded that of 'addsi' in the machine description file (md), the operands were recognized incorrectly as a reg and a sum. This caused an error in the final output phase, where the constraints on the operands get tested.
For some strange reason, an addition stmt continues to produce the same error in bug4.c, while the operands are correct now, but the first and the second are not the identical register they should be. We can sidestep this problem by allowing them to be different, and simply producing an extra move instruction in those cases.]

5. Array declaration incorrect [Arrays, global] Aug 10
- Not correct count when declared as global: always 1.
- Zero count in bug1.c (see 16)
[Sep 23: ASM_OUTPUT_{COMMON,LOCAL} changed back. Rounded is not in bits]

22. Division not set up like other function calls ~Sep 1
[Sep 27 : Div&mod functions get a special treatment (aux-output)]

\begin{SWITCH bugs}

14. switch statement. [switch] Aug 22
a) There are two bugs: (1) in general, the switch does not jump to the right case;
[Oct 3: Fixed. The branch for non-inverted lt&ltu was inverted]
b)
(2) when there are six or more cases (including the default one), error message "gdd: Program cc1 got fatal signal 11" generated by the compiler. (YF)
[Oct 3: Fixed. The macro ASM_ADDR_VEC expected rtx but got int]
- Still problems. See bug#25.

25. Actual addresses expected in address vectors Oct 3
- Try to do define_expand that eliminates the access.

```
[Oct 4: Prev insn searched for register used as address.
  Not the ultimate in safety.]
[See bug #29. Eliminating the access is essential]

26. Test before switch insufficient Oct 3
- Tests only upper bound, not if negative.
[See bug#16,25 (switch), #4 (unsigned)]
[Dec 4: Directly linked with unsigned comparison]

29. Redundant load of address into jump table causes segm fault Dec 4
- (I.e. switch stmt). Try removing using def_peephole
- See bug #25
[Dec 5: 'casesi' pattern rewritten from 68000 specs. Now all correct]

\end{SWITCH bugs}

24. [gdd] When only the linker is run, basename disappears Oct 3
- e.g. gdd d.dc  -> dapa -o .dof d.dc <+otherfiles>
[Oct 4: Convention changed: d.out produced if no -o switch.
 -A switch works only so-and-so. Impossible to fix.]

28. Nested function calls mess up registers Oct 10
[Oct 24: Setup & function-call epilogue, altered significantly.
  Nested activation records maintained as a linked list.]
- see bug8.c

27. Spurious errors when using globals in a long file Oct (7)
- See bug7.c
[Nov 17: Switched from rtl-expansion, to using extra registers
and doing a reference of globals in the asm output generation.
- Cause not uncovered. ]

16. Referring to elements of global arrays Aug 24
- cc1 attempts to load a literal plus the constant in same stmt
- See bug1.c
[Sep 1x: Globals worked out: Pointer loaded and then referenced]
[Sep 25: Problem: plane offset used, plus negative offsets filter in]
- I.e. status: Offsets are wrong (plane offset), & neg offsets fail
- Using "a" in operand constraints seemed to give some hope
[Nov 17: Fixed. The switch to no expansion allowed moving the
  offset from the "rar" command, to the "rw" command.]
-See also bug9.c

4. Unsigned: comparisons & arithmetic [Unsigned,comp,arith]  Aug 10
- Was: the code generated is the same.
- Negative values for switch stmt will loop forever because of this
[Dec 4: Entries in md for unsigned comparison done using Carry flag]
[Dec 5: No overflow checks needed.]
[NOTE: Note thoroughly tested yet]
[Unsigned + and - work same as signed. Unsigned * and / are there.]


Appear fixed
------------
23. Using function's return value, messes up stack Sep 27
- m6 and m7 get stored into. (see c0.c,c1.c,c2.c)
[Sep 28: m7 now set to be fixed, and not stored into, but m6 still]
[Sep 28: Usage of mod regs should now be better]
- Happens only when other modregs are needed, eg fn call w/ 5 args
- Very likely to be related to 21) above.
[Oct 1: Reduced #parms in regs to 2 (from 4). Might alleviate the
  problem for good]
- The argument base reg appears to be the first candidate for using.
- Fixed regs apparently aren't completely fixed, but used only
  when nothing else is available. Unfortunately, the compiler doesn't
  know about our other usages of m6, and thus thinks it need not
  be spilled.

21. Optimizing allocatetest.c blows up (Segmentation fault) Sep 21
- Strange combination of pointers, function calls and liveness
- See bug6.c
- The reduction of parms in regs did not help.
[Nov 17: The changed handling of globals appears to have helped]
```

A.3 Release Notes

```
GDD Release Notes
-----------------

/* This file is to provide an up-to-date review of the state of the
   gdd compiler: known bugs and limitations, and recent improvements.
   In this file, or associated log file, each mini-version will be
   described in terms of changes, and bugs fixed/discovered.
 */

Sep 22: Update # 001
```

```
  - STACK statement added at the end of each module.
Currently it is given a constant argument (i.e. 20), but ultimately
it should depend on the number of functions in the module.
  - Comments following a string decl. no longer cause a too long source line.
  - Global variables now declared correctly.
  - Global arrays should be fully functional now.

Known bugs and limitations:
  - The 'short' datatype causes the compiler to crash
  - The 'unsigned' datatype is not computed/tested correctly
  - The bitfield datatype seems to be handled incorrectly
  - The switch statement does not jump to the right case.
  - A register can get clobbered when performing a test
(Happens almost exclusively when optimizing)
  - Division is not expected to function correctly (nor mod)
  - Optimization can crash the compiler
  - Hardly anything has been tested fully for a long time.

Sep 25: Update # 002
  - Switched to using the EnteringC and ExitingC macros

New bugs:
  - Reference to global arrays done incorrectly: plane offsets & neg offsets

Sep 27: Update # 003
  - The calling sequence updated & tested to be working.
  - The startfile ~dap/local/lib4/DAPmain.dc written and included automatically
  - The standard library, DAP.dl also searched automatically.

Sep 27: Update # 004
Fixed bugs:
  - Division and modulus operations working
  - (Minor ones:) Comments longer than 80 chars
New bugs:
  - When a function return value is used, it can mess up m7&m6 and thus crash.

Sep: Update # 005
  - #parms changed to 2 (from 4), to try to avoid the above bug.

Oct 24: Update #006
  - Calling mechanism altered: Setup & function-call epilog. AR stack kept.
  - Fully correct by Oct 27.

Oct 31: Update #007
  - Handling of global variables changed radically:
```

```
- Move-expansion thrown out.
- References to global symbols are treated directly in the output.
- The output makes use of register 12.

Nov 17:
  Fixed: 1) Nested function calls.  A linked list of activation records is
kept, separate from gdd's pointers.
2) Globals work correctly (while not optimal efficiency).
  Move-expansion was thrown out once again. m5 and m12 used (m5 saved).
3) References to elements of global arrays now use word offset.
4) Optimizing (see bug6.c) does not cause the same crashes as before.

Nov 18,19:
  Fixed: Sharing globals between several source files.
- Each global placed in a section of its own. Statics and constants
put in a single non-shared section.
```

A.4 Bug Log

```
Current Bugs in gdd
-------------------

Current ones
------------
1. Short crashes [Short int] Aug 10
- Compiler makes it BLKmode internally => Problems w/ GET_MODE_SIZE etc
   (Nov 28)
- One way is to throw out all 'short' refs in preprocessor

13. bitfield extraction. [bitfield,structure] Aug 22
the code generated for "bitfield*.c" is not correct.
there are two clues (I did not run any test but observed
the apal code): (1) wrong bit fields are extracted, e.g.
"bitfield1.c"; (2) reg m4 is used before defined, e.g.
"bitfield2.c".  (YF)


Optimization problems
---------------------
30. Optimizing switch code wipes out addres loading, regs, labels
- See switch2.c (-O).  See also bug# 29

2. Comparison register clobbered and not saved Aug 10
[Adding a (clobber) in the md pattern did not help (Sep 23)]
```

```
[Nor did doing an expand to an explicit subtraction]
- Probably appears only when trying to optimize
- See tst11.da when optimized

In salt
-------

Likely bugs:
 - Function pointers
 - Unions?
```

A.5 Work List

```
Worklist for gdd compiler
-------------------------

I. *** Jobs in the queue ***


a) ===== System level stuff =====

6. Set up test data for all the current testfiles.

15. Integrate all test files into a single suite.
   - Including Yong-Fong's

100. Document compiler: DAP changes; operation, functionality, bugs;
files, tools, scripts.

101. Wrap up the files, and the project

b) ==== Bug fixing

(See bugs.log for current bugs)


c) ==== Feature testing affirmation =====

18. Recursive calls; try factorials. (feature test)
[Aug 22] passed "factorial.c" which contains a recursive function.(YF)
```

```
20. Enumeration. (feature test)
[Aug 22] Without really running the code, I observed that
 enumeration is OK by inspecting the generated apal code. (YF)

36. Union constructs.

\newpage
II.  *** Cancelled jobs ***

5. Finish version 3 of CT (pass block of tuples to dap) (-> MMH)
[Merely for speed. Not too important at the moment.]

9. Document compiler options, idiosyncrasies, tools & scripts.

10. Think about profiling
- The macro FUNCTION_PROFILER could do the job

23. Implement library routines for string operations. (YF)

33. Clean the output when the -g switch is given

d) ****** Optimization/Efficiency Jobs =======

i) Div & mod should not be treated as full function calls.

ii) Accessing locals is too expensive:
   make rr m1, m7; addh m1, 11; rw m3, 0..0(m7)  into  rw m3, 0..11(m7)
```

Scan on DAP is $O(\sqrt{N})$

J. Storrs Hall and Magnús Halldórsson
Laboratory for Computer Science Research[1]
Rutgers University

Abstract

Scan, also called *parallel prefix*, is a vector operator taking a scalar function and applying it to successive initial substrings of the vector. Scan can be implemented in $O(log\ n)$ on a parallel processor with a tree connectivity. We show that scan can be implemented on a DAP architecture, which has 2-dimensional mesh connectivity with row and column busses, in $O(\sqrt{N})$ time.

[1] Supported by Air Force Office of Scientific Research under AFOSR-86-0294 and DARPA under RADC t.n. B-8-3615 and B-8-3616.

Introduction

The $\oplus$-*scan* of a vector $A = a_1, a_2, ...a_n$ is a vector S where $s_i = a_1 \oplus a_2 \oplus ... \oplus a_i$. For example, +-scan of $4,1,5,2,6,3$ is $4,5,10,12,18,21$. The *max*-scan is $4,4,5,5,6,6$.

Scan is a very useful parallel processing primitive, and indeed has been used as the basis of a model of parallel computation (Blelloch 87). This reference explains how scan can be implemented in logarithmic time on a tree-connected parallel processor. Algorithms for a mesh-connected computer[1] are given in (Fiduccia 88) which are $\Omega(\sqrt{N})$. This is unsurprising since the diameter of a mesh is $2\sqrt{N}$.

The DAP (Godfrey 86) is a mesh-connected processor augmented by row and column busses, so that its diameter is 2. However, the bandwidth of the busses is $\sqrt{N}$ (since only one set may be used at a time), presenting an interesting dual to the basic mesh connections.

In the following section, we examine scan algorithms for a single row of the DAP, considering only left and right neighbor connections and the row bus. We are also ignoring the fact that the DAP is a bit-serial machine. Thus, as an independent parameter, the timings for the bit serial implementation of the scalar function ("$\oplus$") must be multiplied by the number of times it is done in the scan algorithm. In all the algorithms below, however, this is logarithmic, which is optimal for scan.

A remark: $O(\sqrt[4]{N})$ algorithms are not often seen, and as a result, it may not be realized what an efficient timing this is. $O(\sqrt[4]{N})$ is less than $log_2\ n$ for $n < 65536$. (Since log operations is optimal for scan, though, we would expect to, and do, see a constant factor which defeats the (admittedly small) difference between the two functions in this case.)

Algorithms

Our algorithm is based on two basic algorithms for scan, based on shifting and broadcast, respectively. These algorithm statements assume a vector $A = a_1, a_2, ..., a_n$, where n is a power of 2.

Algorithm A: Shifting Scan

```
[1] Repeat step 2 with $k = 2^j$, for $j \leftarrow 0, 1, ..., log\ n - 1$:
[2] $a_i \leftarrow a_i \oplus a_{i-k}$ for all $i > k$.
```

The problem with this algorithm is that the shifting operation is linear in the distance shifted, so that the total time taken is $\sum_{i=0}^{log\ n-1} 2^i = n - 1$, the sequential time!

[1] In describing a square mesh processor array, we follow the usual convention that N is the total number of processors, n is the length of a side, where $n = \sqrt{N}$.

Algorithm B: Broadcasting Scan

[1] Recursively (in parallel) apply (any) scan algorithm to the vectors $a_1, ..., a_{n/2}$ and $a_{n/2+1}, ..., a_n$.
[2] $a_i \leftarrow a_{n/2} \oplus a_i$ for all $i > n/2$.

Similarly, the problem here is that the broadcasts in the recursive steps cannot be done in parallel, since they must use the same bus. The total number of broadcasts is again $\sum_{i=0}^{\log n-1} 2^i = n - 1$.

The timings for the steps of algorithm A are $1, 2, 4, 8, ..., \frac{n}{2}$, and those for algorithm B are $\frac{n}{2}, ..., 8, 4, 2, 1$. This suggests a combination where we do the first $2^{\lceil \frac{\log n}{2} \rceil})$ steps in algorithm A and the final $2^{\lfloor \frac{\log n}{2} \rfloor}$ steps in algorithm B.[1]

Then the time for the total algorithm becomes

$$2 \sum_{i=0}^{\log n/2-1} 2^i = 2(2^{\frac{\log n}{2}} - 1) = 2\sqrt{n} - 2,$$

if *log n* is even, and

$$\Big(\sum_{i=0}^{\lceil \log n/2 \rceil - 1} 2^i\Big) + \Big(\sum_{i=0}^{\lfloor \log n/2 \rfloor - 1} 2^i\Big) = (\sqrt{2n} - 1) + \Big(\sqrt{\frac{n}{2}} - 1\Big) = \frac{3}{\sqrt{2}}\sqrt{n} - 2,$$

if *log n* is odd. In both cases this is $O(\sqrt{n})$.

The problem, of course, is that the intermediate values of a_i are not the same for the two algorithms. To account for this, it is necessary to partition the list into $2^{\lfloor \frac{\log n}{2} \rfloor}$ sublists (each of size $2^{\lceil \frac{\log n}{2} \rceil}$). Executing Algorithm A in parallel over all the sublists is identical to executing it over the whole list, except that shifted data do not cross the sublist boundaries.

To implement a scan on all processors using row scan as a primitive, first do a row scan of all rows (simultaneously). Then perform a scan of the final *column*. Shift this result down one row, broadcast across every row but the first, and $\oplus$ the broadcast value to the result of the original row scan in each processor.

This takes two row-scan times and a few constant-time operations, so it is the same order of complexity as the row-scan, i.e. $\sqrt{n}$, which is of course $\sqrt[4]{N}$.

Implementation

We display the algorithms here in a c-like pseudocode. Arrays are expected to be the size of the mesh; `arr a[b;n;n]` means that `a` is an array of `b` bit numbers and the mesh is `n` by `n`.

[1] Assuming, if *log n* is odd, that a shift step is faster than a broadcast step–there are the same number of steps in the "contested" iteration.

Algorithm A:

```
arr a[b;n;n]
  for i=0:(log n)-1
    a[;;J>2^i] += 2^i rsh a
```

(Note that n rsh a means shift a to the right n places, taking time n (times the length to the words to be moved).)

Algorithm B:

```
arr a[b;n;n]
  for i=0:(log n)-1
    for j=2^i:n-1:2^(i+1)
      a[;;j<J<j+2^i] += a[;;j]
```

Both of these algorithms, of course, only scan along the rows; the full scan would be accomplished by a double application like:

```
<scan algorithm as above>
<scan a[;n-1;...]>
a[;I>0;] += 1 dsh a[;;n-1]
```

Conclusions

It may seem odd to talk about the asymptotic complexity of an algorithm in terms of the number of processors of a machine which has a fixed number of processors. This analysis, of course, is intended to apply to any machine with this architecture, namely a square mesh with next neighbor connections and horizontal and vertical busses, whose edge length is a power of two. $O(\sqrt[4]{N})$ is the equal of $O(\log n)$ for machines of up to 64k processors, which is the size of a full fledged Connection Machine. That the time for scan on a bussed mesh of that size is no more than the CM with all its communications hardware is, we think, significant.

We are forced to conclude, however, by noting that by the time the bit serial nature of the DAP is taken into account, and the fact that our DAP only has 1k processors, that it can do a scan no faster than its host Sun 4.

References

Blelloch, Guy: *Scans as Primitive Parallel Operations*, pp 355-362, Proceedings of the 15th International Conference on Parallel Processing, Pennsylvania State University Press, University Park, 1987

Fiduccia, C.M., R.M. Mattheyses, and R.E. Sterns: *Efficient Scan Operators for Bit Serial Processor Arrays*, Frontiers of Massively Parallel Computing, George Mason University, 1988

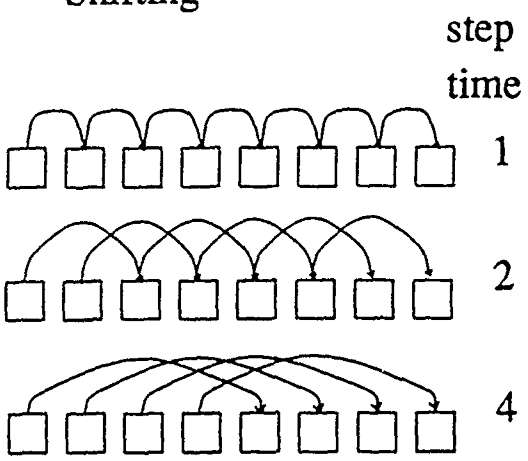
Foster, Caxton C.: **Content Addressable Parallel Processors**, Van Nostrand Reinhold, New York, 1976

Fountain, Terry: **Processor Arrays: Architecture and Applications**, Academic Press, London, 1987

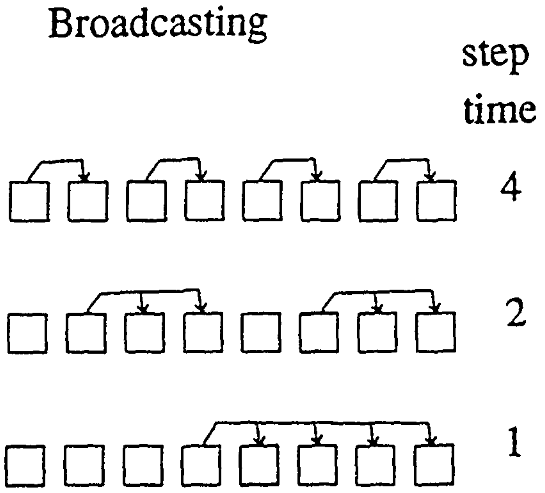
Godfrey, M.D: *Innovation in Computational Architecture and Design*, ICL Technical Journal, May 1986, pp 18-31.

Ladner, R.E. and M.J. Fisher: *Parallel Prefix Computation*, JACM 27(4), 1980, pp. 831-838.

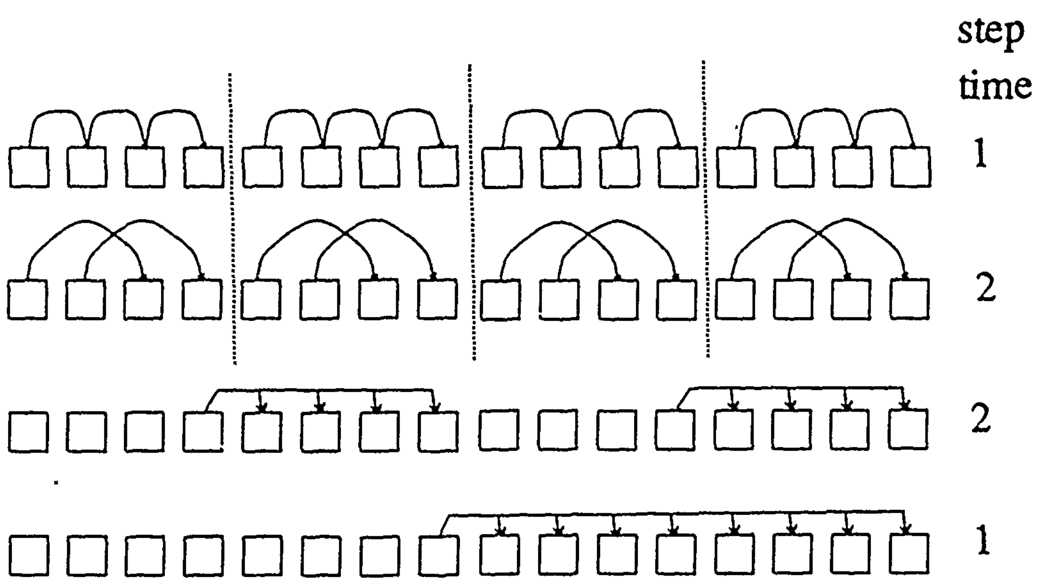
Algorithm A
Shifting



Algorithm B
Broadcasting



New algorithm: partitioned shift and broadcast


step
time
1
2
2
1