PUP: An Architecture to Exploit Parallel Unification in Prolog

Chien Chen
Ashok Singhal
Yale N. Patt

Computer Science Division
University of California, Berkeley

ABSTRACT

The architecture described in this paper achieves high performance execution of Prolog programs by exploiting fine grain parallelism. Fine grain parallelism includes unification parallelism as well as parallelism among the arithmetic and bookkeeping operations. Our implementation of the architecture has multiple functional units, each tailored to a specific task, that operate in parallel. Unification and arithmetic operations are allowed to execute out of order and are dynamically scheduled on several unification units. Simulation results for the implementation are presented and analyzed.

1. Introduction

The increasing popularity of logic programming languages, most notably Prolog [7], has resulted in a large number of research projects, including the Aquarius project [9] at Berkeley which supported this research, that are attempting to build high performance Prolog systems by exploiting parallelism at various granularities. Most of the current work attempts to exploit parallelism at the level of parallel processes executing on multiple processors. However, since each processor has only a short pipeline, the amount of overlapped execution of instructions within a process is quite limited. The architecture described in this paper, the PUP (Parallel Unification Processor), overcomes this limitation of short pipelines and overlaps the execution of several instructions of a Prolog program. We describe the architecture and an implementation of the PUP and present simulation measurements and analysis to demonstrate its effectiveness.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAR 1988		**00-00-1988 to 00-00-1988**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
PUP: An Architecture to Exploit Parallel Unification in Prolog	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

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

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The architecture described in this paper achieves high performance execution of Prolog programs by exploiting fine grain parallelism. Fine grain parallelism includes unification parallelism as well as parallelism among the arithmetic and bookkeeping operations. Our implementation of the architecture has multiple functional units, each tailored to a specific task, that operate in parallel. Unification and arithmetic operations are allowed to execute out of order and are dynamically scheduled on several unification units. Simulation results for the implementation are presented and analyzed.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**52**	

1.1. Parallelism in Prolog

In this section we briefly describe the Prolog language and the main forms of parallelism in Prolog programs (see [20] for more details). Prolog is based on the resolution of Horn clauses using a top-down, left-to-right ordering. Programs consist of a *query* (procedure call) and a collection of *terms*. Terms can be either *simple* (a variable or a constant), or *complex* (a structure). Variables in Prolog begin with a capital letter. Constants are numbers and atoms. Atoms begin with a lower case letter. A structure has a functor (name of the structure and its arity) and arguments. The arguments are themselves terms. Terms represent programs and data. A procedure contains one or more *clauses*. When a procedure is called, its clauses are tried in sequence until one of them succeeds. A procedure succeeds when one of its clauses succeeds. If none of the clauses of a procedure succeed, the procedure fails. A clause is a term that has a *head* and could also have a *body*. The head of a clause is term. The functor of the term (name and arity) uniquely identifies the procedure of which the clause is a part. The arguments, if any, define the formal parameters of the procedure. The body of a clause contains *goals* (procedure calls). A clause succeeds when the arguments of the clause head *unify* with the respective input arguments and all the goals in the body execute successfully. Unification is the process by which a set of substitutions or *bindings* of the variables in the two expressions being unified result in identical expressions. If no such set of bindings is found, the unification fails. *Backtracking* is the process by which the system restores the state in order to try an alternative clause when the previous clause fails. Thus, backtracking implements the depth first search of the tree of possible solutions to a query.

Parallelism can be exploited in Prolog at different levels. Several clauses of a procedure can be tried in parallel, resulting in OR parallelism. When several goals of a clause are executed in parallel, AND parallelism is exploited. Most research projects that attempt to exploit parallelism in Prolog, for example [4,8,12-14,17], target AND and OR parallelism at the level of processes executing on multiple processors. In unification parallelism, the various components of the two expressions being unified (the arguments of the clause head and the input arguments) are unified in parallel. Clearly, unification operations have a finer granularity than AND and OR processes and must be exploited at a lower level than a process. Other forms of parallelism include stream parallelism (where subgoals are executed in a pipelined fashion) and search parallelism (where the heads of several clauses are unified in parallel with input arguments). However, these forms of parallelism can be considered special cases of AND, OR and unification parallelism. The PUP is designed to exploit unification parallelism (a multiprocessor based on the PUP could exploit AND and OR parallelism as well).

Although most of the "useful work" of a Prolog program is done by unification operations, several bookkeeping operations must also be performed to control execution. The type of bookkeeping operations required depends on the abstract model for Prolog execution. In this project we use a register based, environment stacking model similar to the Warren Abstract Machine [23] since it has been shown to be superior to other known models (see [21]). The storage model consists of a *heap* on which the various data structures required by the program are created, a stack that contains *environments* and *choicepoints*, a *trail stack*, and a *push down list* that is used in the unification of nested lists and structures. The processor contains stack pointers and argument registers. Choicepoints are used to save the state of the machine before trying a clause of a procedure so that the state can be restored if the clause fails and another clause must be tried. The trail stores addresses of variables that are bound during the execution of the clause so that these bindings can be undone (by a de-trail operation) if the clause fails. Environments are used to save some registers before calling a goal since these registers may be overwritten by the goal but are required for subsequent goals in the body of the clause.

Bookkeeping operations in this model include creation of choicepoints, trail operations, restoration of state from a choicepoint, de-trail operations, and saving and restoring environments. Table 1 shows the percentage of the number of cycles spent on categories of instructions for some benchmarks running on the VLSI-PLM simulator [10]. The instruction categories will be explained later in the paper. The benchmarks are part of a collection maintained at Berkeley. The "boyer" and "browse" benchmarks are part of the Gabriel Lisp benchmarks coded in Prolog. The other benchmarks are smaller and have been used in several earlier studies [6,10,12]. The unification category (includes "get", "put" and "unify" instruction types) takes from 36% to 83% of the total execution time. Mulder and Tick [18] assign weights to instructions based on the number of memory references made by the instruction on another set of benchmarks. They claim that unification instructions (get, put and unify) have a total weight of only 20% to 25%. Note that their weights do not necessarily correspond to execution time. The conclusion in either case is that unification parallelism alone cannot consistently result in sufficiently large speedups. If parallel unification is to be effective in speeding up execution, the time spent on the other operations must also be reduced. In the PUP we do this by overlapping bookkeeping operations and buffering the stacks. Therefore, unification parallelism is exploited along with parallelism among the bookkeeping and arithmetic operations. We refer to this as *fine grain* parallelism.

1.2. Previous Work

As mentioned earlier, only a few research projects try to exploit fine grain parallelism beyond the limited overlap allowed by short processor pipelines, and we review these in this section. Papers dealing with other forms of parallelism in Prolog are not included.

Theoretical results about unification are discouraging at first glance. The results of Dwork et al [11] and Yasuura [25] indicate that it is very unlikely that "good" (polynomial logarithmic time using polynomial number of processors) parallel algorithms can be found for unification. However, although unification of all term pairs cannot be performed in parallel, there are also many term pairs that can exploit parallel hardware. Citrin [6] has studied techniques for determining schedules for parallel unification. He uses static data dependency analysis [5] to identify independent unification operations and schedules them at compile time. However, his work has limited practical applicability for several reasons. First, he only considers parallel unification. Without speeding up other operations as well , very limited speedup can be achieved. Second, in his scheme, parallel unifications execute in groups of independent operations. Unifications belonging to different groups may not overlap. Also unification operations belonging to the next goal may not overlap with those of the current goal. Third, static analysis results in worst case schedules that may be too pessimistic especially when clauses are "asserted" (added to the program) or "retracted" (deleted from the program) during program execution.

Several data flow models for execution of logic programs have been proposed. Ito et al [16] describe one such architecture for Parallel Prolog and Concurrent Prolog. The machine is designed to exploit AND, OR and unification parallelism. Several processing elements access *structure memories* over an interconnection network. Data flow nodes are provided for primitive operators (for example, the unification operator, check-consistency operator and substitute operator). The unify operator has two input ports for the terms being unified, and two output ports, the I port and the E port. The I port supplies an instance of the two unified terms and the E port supplies the set of bindings for variables in the two input terms. The check-consistency operator checks for consistency between two binding environments that may share variables. The check-consistency operator tries to unify the two sets of bindings to produce a single binding environment. The substitute operator performs substitutions from a given environment to a given term. Arcs of the data flow graph also connect the clauses of a procedure and goals of a clause so that AND and OR parallelism can also be exploited. Although such an execution mechanism could potentially exploit parallelism extensively, no performance results were presented and no details of practical considerations regarding the construction of such a machine were described. Practical data flow machines of this type have yet to be demonstrated.

The POPE [2] is a novel pipelined architecture for Prolog. The pipeline consists of a series of WAM based processors arranged in a ring. Adjacent processors communicate through a pipeline buffer. All the processors share a common data memory but have local copies of code. The pipelining in the POPE can be viewed as pipelining at the procedure level. In standard sequential execution, the arguments of a procedure (goal) are loaded before the procedure is called. In the POPE, the address of the procedure is first loaded into the next pipeline buffer (so that the next processor can fetch and execute instructions) and then the arguments are loaded into the buffer. The next processor can unify arguments as soon as they are loaded into the buffer. When all the code for the procedure has been executed, the processor picks up the address for another procedure call from its input pipeline buffer. This allows the overlap of both unification operations as well as bookkeeping operations. The pipeline buffers also serve as a single choicepoint buffer. Stack pointers are used as tokens for synchronization (for example, a processor cannot allocate space on the heap unless it has the heap pointer and the previous processor will only relinquish the heap pointer when it is sure that it does not need it any more for the current procedure). The POPE does achieve overlapped execution of unifications and bookkeeping operations but only operations belonging to different goals may be overlapped.

1.3. Organization of the Paper

In Section 2 we describe our design philosophy and justify our design choices. In section 3, we specify the PUP architecture and its implementation. Our simulation measurements are presented and analyzed in Section 4. In Section 5, we offer some concluding remarks.

2. Design Philosophy

In the previous section we introduced fine grain parallelism (which includes unification parallelism) that the PUP exploits. In this section we describe and justify the principles used in the design of the PUP. The main principles are the use of multiple functional units, each tailored to a specific task and operating in parallel, out of order execution of instructions, and dynamic scheduling of operations to the functional units.

2.1. Multiple Specialized Functional Units

Prolog, like most other languages, has certain distinctive characteristics. Special hardware support that caters to these characteristics can result in a high performance Prolog system. In addition, tasks that are independent of each other can be executed on parallel hardware units, each tailored to the specific tasks that it performs. For example,

independent unification operations can execute in parallel on unification units while another type of unit copies a buffered choicepoint into the choicepoint stack in memory. Our implementation of the PUP uses this approach to execute instructions in parallel.

The instructions of the PUP must be of sufficiently coarse grain for multiple functional units to be effective. Otherwise, the scheduling overhead will eliminate all the benefits of parallel execution. We chose an instruction set based on the WAM since the instructions have a sufficiently coarse granularity.

2.2. Out of Order Execution

Several operations execute concurrently in the PUP. However, some of these operations, for example unification, can take a varying number of cycles to complete. Therefore, it is quite possible that operations can complete before those that occur earlier in the instruction stream. Also, some operations may have to stall for data from previous operations, while other operations that follow may be ready to execute. In order to take advantage of the available parallelism, the PUP allows out of order execution. Instructions may execute when all their input data are available.

2.3. Dynamic Scheduling

In general, it is not possible to determine the execution time of unification operations at compile time. Static analysis and scheduling as proposed by Citrin [6] results in worst case schedules and low utilization of unification units (since all parallel operations on the unification units must complete before any of the units can start executing the next group of parallel operations). Extensive static analysis of programs is required and the analysis is ineffective when the program or data are modified by asserts or retracts. To overcome these problems, the PUP schedules operations dynamically among unification units. An operation that is ready to execute is dispatched to the first free unification unit. This scheduling policy is not optimal but is simple to implement (see Section 3) and produces near-optimal schedules most of the time [1]. Optimal schedules cannot be computed since we do not know the execution time of all the operations.

Static schedules proposed by Citrin ensure that only one parallel unification operation will try to bind an unbound variable. With out of order execution and dynamic scheduling, however, we have to provide synchronization mechanisms to ensure that unbound variables that are shared among several unification operations will be bound by only one of the operations. The reasons for this synchronization requirement will be explained in Section 3.

3. Description of the PUP

The instruction set architecture of the PUP is based on the WAM [23] and the Berkeley PLM [10]. New features have been included to enable parallel execution by an implementation based on the design principles outlined in Section 2. In this section, we describe the architecture and an implementation. An example to illustrate the operation of the PUP is also presented.

3.1. Instruction Set Architecture

3.1.1. Data Types

Data elements of the PUP are organized as 32-bit words. Each word is made up of 4 fields (see figure 1): a 3-bit tag, a cdr bit, a 27-bit value field, and a bit for garbage collection (currently unused). The tag field specifies the type of the words. Possible data types are *bound variable, unbound variable, structure, list, constant integer, constant floating point,* and *nil* (a special constant). The value field of a bound variable contains a pointer to the data to which it is bound. Value fields of structures and lists contain pointers to the first element of the structure or list. Lists are cdr-coded as in the PLM. If the cdr bit of a word is set, the word represents the last element of the list. Cdr-coded lists require less storage space than traditional linked lists that require two words for every element of the list (one for the element itself and one for the link). Figure 2 explains how lists are coded using the cdr bits.

3.1.2. Processor State

The PUP registers are summarized in table 2. The argument registers X0 - X7 are used to hold the arguments to procedure calls. The write-once registers W0-W7 contain variables that appear in more than one argument (shared variables), as well as variables that appear within complex arguments (list or structure arguments). As the name suggests, write-once registers are only written once during the unification of the head of a clause. They are used to synchronize access to shared variables and to enable unification units to unify elements of complex arguments. The function and operation of write-once registers will be described later in greater detail. The continuation pointer (CP) stores the return address of a procedure call and points to code that must be executed after the current clause succeeds. The environment pointer (E) points to the location of the previous environment on the stack. The backtrack pointer (B) points to the location of the previous choicepoint. The trail pointer (TR) points to the top of the trail stack and the heap pointer (HP) points to the top of the heap. The stack pointer (T) points to the top of the stack that

contains environments and choicepoints. Readers familiar with the WAM or PLM architectures will notice that there is no structure pointer (used to point to elements of lists or structures on the heap during unification). This is because the unification of a list or structure is treated as a single instruction in the PUP. The WAM and PLM have separate "unify" instructions for each element of the list or structure. The structure pointer is, therefore, not specified in the instruction set architecture (although the implementation may have structure pointers to implement list and structure unifications).

3.1.3. Memory Organization

Data memory in the PUP is divided into a stack for environments and choicepoints, a heap, and a trail. The push down list is not specified in the instruction set architecture since list and structure unifications are treated as single instructions. Sublists or substructures that are pushed on to the push down list during unification are not visible to the programmer at the instruction set level. The implementation may provide memory or separate storage for the push down list.

3.1.4. Instructions

A complete description of all the instructions of the PUP is inappropriate for this paper. In this section we describe the instructions very briefly to illustrate the operation of the PUP. The simulator described in Section 4 does not implement all the instructions that a real machine would have, but only those required to run the benchmark suite and demonstrate the feasibility of such an architecture.

The instruction set consists of six groups of instructions: indexing, clause control, procedure control, get, put and miscellaneous instructions. These groups are summarized in table 3.

3.1.4.1. Indexing Instructions

Indexing instructions are used to filter a set of clauses that are candidates for unification. They eliminate clauses that cannot possibly unify with the current goal since there is a mismatch in at least one argument. The *switch-on-term* (swot) instruction branches to one of four destinations based on the type of the argument. Separate destination addresses are provided for unbound variable, constant, list and structure types. For example, if the argument is a list, all clauses that require a structure or constant in that argument position can be eliminated. If the argument is an unbound variable, it could possibly unify with all the clauses. Code at the destination for unbound variables must, therefore, attempt all the clauses. The *switch-on-constant* and *switch-on-structure* instructions

use hash tables to filter clauses even further if the arguments are constants or structures respectively.

3.1.4.2. Procedure Control Instructions

Procedure control instructions create, modify and delete choicepoints on the stack. The *tryelse* instruction creates a choicepoint and transfers control to the first clause to be tried (specified by the first address in the instruction). The second address points to the code that must be executed if the first clause fails. The *retryelse* instruction is used to try the next clause and modify the choicepoint to point to the code that must be executed if the the clause fails. It is used for all clauses except the first and the last. The *trust* instruction deletes the choicepoint and tries the last clause. The *fail* instruction causes the clause to fail and backtrack to the last choicepoint.

The *cut* instruction implements the cut operator (!) of Prolog. It discards all the choicepoints of the current procedure and its goals. The *cuts* is a special case of the cut instruction and is used as an optimization when it is known that the current clause has not called any goals. The *cutd* is used to correctly implement disjunctions in Prolog. Explanation of the cutd instruction is rather involved and interested readers may refer to [10] for more details.

3.1.4.3. Clause Control Instructions

This group of instructions is responsible for control transfer and environment allocation associated with procedure calls and returns. The *allocate* and *dealloc* instructions allocate and deallocate environments. The *call* is used to call a goal that is not the last goal in the body of the clause, whereas the *execute* instruction is used to call the last goal of the clause. The *call* instruction saves the return address in the CP register. The *proceed* instruction is a return from a procedure call. If the procedure was called using the *call* instruction, control returns to the caller so that the next goal can be executed. However, if the procedure was invoked using the *execute* instruction, control is transferred to the return address in the last environment on the stack.

3.1.4.4. Get Instructions

The *get* instructions are used to unify the input arguments of a procedure with the arguments of the head of the clause currently being executed. The *getconst* instruction tries to unify the argument register with a constant and is used when for constant arguments in the clause head. The *getvalx* tries to unify the two argument registers specified. The *getvaly* tries to unify the specified permanent variable (a variable that is stored in the

environment) with the specified argument register. The *getwx* tries to unify the specified write-once register and argument register, and the *getwy* tries to unify the specified write-once register and permanent variable. The *getvarx* instruction is used for variables in the clause head and copies the first argument register to the second. Similarly, the *getvary* instruction transfers the contents of the argument register to the permanent variable. The *getlist* and *getstruct* instructions are used when the argument in the clause head is a list or structure respectively. In the PLM and WAM, these instructions are followed by a series of *unify* instructions for each element of the list or structure. In the PUP, however, the instructions contain pointers to a description of the list or structure, and the unification of the entire list or structure is treated as one instruction. The description could contain sub-lists and substructures. They may also unify elements with write-once registers. If the argument register that is being unified with the list or structure is an unbound variable, the instruction first constructs the list or structure on the heap and then binds the variable to it. The description also contains a field that specifies the size of storage that is required on the heap so that the instruction can request a chunk of heap space of the appropriate size. This is not necessary in sequential execution since heap space cannot be requested for more than one list or structure at a time so there is no danger of elements of two or more lists or structures being interleaved on the heap. For parallel execution, allocation of shared heap space requires synchronization and it would be expensive to allocate space for each element separately. In addition, if space for each element were allocated separately, structures and lists could be interleaved with each other and explicit linking of structures and lists would be required.

3.1.4.5. Put Instructions

The *put* instructions are used to move data into the argument registers before calling or executing a goal. The *putconst* instruction loads a constant into an argument register. The putvarx instruction creates an unbound variable on the heap loads a pointer to this variable in the argument registers. The *putvary* instruction creates an unbound variable in the environment. The *putlist* and *putstruct* instructions create lists and structures on the heap and load pointers to them into an argument register. As with the corresponding *get* instructions, the *putlist* and *putstruct* instructions contain pointers to a description of a list or structure. The *putval* instructions are identical to the corresponding *getvar* instructions. The *putwx* and *putwy* instructions are used to copy the contents of a write-once register into an argument register or permanent variable.

3.1.4.6. Miscellaneous Instructions

Other instructions of the PUP include instructions for magnitude comparison, arithmetic and evaluation of expressions. The list of miscellaneous instructions shown is by no means complete. Floating point instructions are an example of further additions to the instruction set. However, the instructions shown suffice to run the benchmark suite and evaluate the architecture.

The *synch* instruction is used to reset the write-once registers so that they may be used by the next procedure call. It will be discussed in greater detail along with synchronization issues.

3.1.5. Synchronization for Shared Variables

3.1.5.1. Write-Once Registers

Consider the goal and clause shown below. In the head of the clause for the "concat" procedure, the first and third arguments are lists that would be unified in parallel on the PUP with the input arguments of the procedure. The variable *X* in these lists is shared by the two unifications operations.

```
?-concat([a,b,c,], L, [d,e,f]).
concat([X|L1], L2, [X|L3]):- concat(L1, L2, L3).
```

Clearly, the clause head shown should not unify with the goal arguments. However, if the unification operations for the two lists try to unify the variable *X* at the same time, the first would try to bind *X* to *a* while the second tries to bind *X* to *d*. If both perform the binding, they would both succeed producing incorrect results. To solve this problem, we need to synchronize bindings of shared variables. In the PUP, each shared variable in a clause is allocated a write-once register. The architecture guarantees that a write-once register is written at most once and by a single unification operation (until the register is reset by a *synch* instruction). All other unification operations unify with the value written into the write-once register and do not try to write it. The write-once register can then be copied into an argument register using the *putwx* instruction if it is required as an argument for another goal.

The *synch* instruction is executed before the first use of a write-once register in a clause. It causes the processor to wait for all previous unifications to complete and then resets the write-once registers so that they can be reused.

Write-once registers are also used for other variables (even if they are not shared) that occur in lists and structures. In the example above, *L1* and *L3* in the clause head would be allocated a write-once register each. This is because descriptions of lists and structures for the *getlist, putlist, getstruct* and *putstruct* instructions are not allowed to refer to argument registers. This decision was made so that data dependencies between instructions could be handled based on the argument registers and permanent variables they write, without reading descriptions of lists and structures. The reasons will be clearer when the algorithm that is used to satisfy data dependencies (the Tomasulo algorithm [22]) is discussed.

3.1.5.2. Shared Variables in Memory

Write-once registers are used to synchronize binding of shared variables that appear explicitly in the code. However, two or more unification units may also try to bind the same unbound variable on the heap. This situation is illustrated in the following example.

```
?- a([X, L1], [X, L2]).
a([1, 2], [3, 4]).
```

From clause *a* it does not appear that there are any shared variables being bound. However, one unification operation tries to bind variable *X* to the constant 1, while the other tries to bind it to the constant 3. The architecture must guarantee that only one unification operation binds the variable and all others unify with the value of the bound variable. Several methods can be used to ensure synchronization and the method is not part of the specification of the PUP architecture (since the method is not visible at the instruction set level). Our implementation of the PUP uses *dereference locks* for synchronization of shared variables in memory and will be explained later.

3.2. An Implementation of the PUP

Figure 3 shows a block diagram of the system. It consists of a prefetch unit (PFU), a control unit (CU), a node table and global push-down list (NTPDL) and several unification units(UUs). The PFU, CU and each of the UUs may access shared memory over the memory bus. The the CU, NTPDL and the UUs communicate over a synchronous bus called the distribution bus. All units that require the distribution bus during a particular cycle must arbitrate for it in the previous cycle. A single bus arbiter is used. The PFU fetches instructions from the memory, and ships their partially decoded form to the CU on demand over a dedicated, synchronous prefetch bus. The CU executes some instructions and dispatches others (those involving integer arithmetic and unification are executed on

UUs) to the node table in the NTPDL over the distribution bus. The node table facilitates out of order execution and dynamic scheduling of operations for the unification units. The Tomasulo algorithm [22] is used to implement out of order execution. The algorithm was first used in the floating point unit of the IBM 360/91 and is currently being used for the entire execution unit in the HPS project at Berkeley [19].

Instructions are dispatched from the CU even if their arguments are not valid. (since the arguments are the results of previous operations that have not yet completed). The destination registers of instructions are marked as invalid and are assigned tags (the hardware has a special valid bit and tag field for each register). These instructions wait in the node table until their arguments become valid (are computed and distributed over the distribution bus by previous operations). Once the arguments of an instruction in the node table are valid, the instruction may be dispatched to the next unification unit that requests another instruction. When the results of that instruction are ready the UU distributes the result, along with the tag associated with the destination, over the distribution bus. All invalid registers and arguments in the node table that match the tag are then loaded with the value on the distribution bus and marked valid.

The Tomasulo algorithm is not applied to the write-once registers in the PUP. This is because the write-once registers may be specified as destinations within list or structure descriptions. These descriptions are not seen by the CU, which assigns tags for the Tomasulo algorithm. Operations that use write-once registers as a source wait in a unification unit until the register is written.

Each of the unification units has a local push-down list that is used to store sub-lists or sub-structures encountered during unification of a list or structure. The local push-down lists overflow into the global push-down list in the NTPDL. Sublists or sub-structures in the global PDL may be unified by any UU, not necessarily the UU that overflowed. By this mechanism we achieve a simple form of load balancing.

Each of the functional units will be described in greater detail in the next few sections.

3.2.1. The Prefetch Unit

The prefetch unit not only prefetches PUP instructions and aligns operands, but also executes unconditional branch operations. The program counter of the PUP system is maintained by the PFU. When the PFU encounters an **execute** instruction, it resets the program counter (P register) to the destination address and starts prefetching from the new target. For a **call** instruction, the PFU passes the continuation pointer (the return address) to the CU and starts prefetching from the target address. The CU stores the continuation

pointer in the CP register and sends it back to the PFU when it executes a **proceed** instruction. The PFU then resets the P register to point to the return address that it receives from the CU. Similarly, for the **tryelse** instruction, the PFU passes the **else** address (the second argument of the instruction) to the CU which then stores it in the choice point. The PFU resets the P register to point to the target address (the **try** address) and starts prefetching from there. When backtracking is necessary the CU returns the **else** address to the PFU which then loads the P register with this value. When the PFU encounters a **switch-on-term** instruction, it passes the argument register number to the CU. The CU dereferences the register and returns the type of the dereferenced result to the PFU over the prefetch bus. The PFU then loads the P register with the target address corresponding to the type of the argument and starts prefetching from there.

Two versions of the PFU were designed. In the first version instructions were stored in a circular prefetch queue. After each instruction was sent to the CU, the instruction bytes were removed from the head of the queue. Whenever free memory cycles were available, new instructions were loaded into the tail of the queue. On a branch the queue was emptied and reloaded with the new target byte at the head. This version of the PFU worked well for sequential instructions but the penalty for branches was too high since the buffer had to be refilled with data from the memory on every branch. The memory traffic was also at least as high as the number of instruction bytes executed.

In order to reduce the penalty for branches and the memory traffic due to instruction fetches, we added a target instruction cache (TIC) to the PFU. The current version of the PFU has a target instruction cache as well as the circular prefetch buffer as shown in figure 4. Each line of the TIC contains the first 16 bytes following a branch. On a branch, if there is a hit on the TIC, the circular buffer gets loaded with the bytes in the TIC and instruction dispatch can begin right away instead of waiting for memory access to fill the buffer. Memory accesses then only have to fill the rest of the buffer. If there is a miss on the TIC the circular buffer is reset. A line of the TIC is then filled along with the circular buffer so that it will be available the next time there is a branch.

At present the TIC is two-way set-associative and each set consists of 32 lines, each of which is 16 bytes long. The circular buffer is 32 bytes long. The length of a TIC line, the number of lines in a set, and the size of the circular buffer may be varied in the simulator.

3.2.2. The Control Unit

The control unit performs two basic functions: bookkeeping and instruction issuing. Bookkeeping operations consist of trailing variable bindings, creating choice points and environments, and backtracking on failure. The data path of the CU is shown in figure 5.

3.2.2.1. Instruction Issuing

The CU gets zero to two arguments and an opcode from the PFU for each PUP instruction. The two arguments are latched and the opcode is used to set the microprogram counter in the microsequencer. The *get*, *put* and arithmetic instructions are dispatched to the node table and bookkeeping instructions are executed by the CU. The CU issues instructions to the node table sequentially but since data dependencies are resolved by the node table and the unification units, out of order execution can occur. If an instruction modifies an argument register, the CU marks that register as invalid and assigns a result tag to it. This result tag is then shipped to the node table in place of the destination register number. The unification unit that produces the result distributes it along with the result tag and the CU loads the register with the value from the distribution bus.

3.2.2.2. Trail Buffer

The CU also contains a buffer for the trail stack. Trail requests are made by the UU's over the distribution bus when they bind variables. Instead of writing to the trail stack, the CU writes to the trail buffer. If the memory bus is available, the buffer is also copied to the trail stack in memory. Thus, when the buffer gets full, the bindings that have already been copied into memory can be overwritten to accommodate new trail requests. If a clause fails, all bindings that were trailed by the clause must be undone. Instead of reading from the trail stack in memory, the CU can pop entries from the trail buffer. However, if the buffer is empty, entries are copied back into the buffer from memory.

3.2.2.3. Choice Point Buffer

There are two sets of argument registers in the register file of the CU. One set is visible to the programmer. The other, a back-up set, is used for choice point buffering. At failure the back-up set is made the current set. Thus backtracking is fast. To create a choice point, all pending writes to the register file are completed and then the current set is made the back-up set. All subsequent writes are directed to the other set, which is the new current set, and the corresponding registers are marked dirty. A read from a register will return the register value in the back-up set if the corresponding register in the current set is clean. Otherwise the register value is read from the current set. Whenever the memory bus is not being used, registers in the back-up set are copied to the memory allocated to the choice point. Thus choice point creation and shallow backtracking are fast. If we have to backtrack through two or more choice points, the registers are copied from the choice points in memory.

3.2.3. The Unification Unit

Special hardware unification units have been proposed elsewhere, for example [3,24]. These units, however, do not perform parallel unification with synchronization for shared variables. Special nodes for unification have also been proposed for dataflow machines [16], where shared variables are handled by *check consistency* operators. We use a different approach to synchronization of shared variables, and our unification units execute WAM based unification instructions. Figure 6 shows our unification unit data path. The UUs execute all **put, get** and integer arithmetic instructions. Whenever a UU is free, it requests instructions from the NTPDL. The arguments and result tag are latched and the opcode sets the microprogram counter in the microsequencer. When the instruction is completed, the result, if any, is latched into the distribution latch and distributed over the distribution bus.

3.2.3.1. List and Structure Unification

The getlist and putlist instructions require the unification units to obtain chunks of heap space of the size of the list or structure being unified. This size is available from the size field in the descriptions of lists or structures in code space. In order to obtain the heap space, the UU maintains a heap pointer, H. It tries to distribute the value of the present heap pointer added to the size of heap space required. At the same time it watches the distribution bus for a new value of the heap pointer. If it can distribute its new H value then it has succeeded in obtaining the required heap space. If it sees a new H value being distributed before it has a chance to distribute it, the H must be updated to the value seen on the distribution bus and the new H value must be recomputed before retrying.

Nested lists and structures are unified using the push down list. When a sublist or substructure is encountered, the addresses of the rest of the list or structure are pushed onto the push down list and the sublists or substructures are unified. After this is completed, the addresses are popped off the PDL and the rest of the list or structure is unified.

The unification unit uses a small local PDL. When this overflows, the top of the PDL is popped and pushed into the global PDL in the NTPDL over the distribution bus. Entries in the global PDL may be unified by any unification unit and this allows simple load balancing if deep lists or structures are used. It should be noted that a stack discipline for the PDL is not necessary. It is convenient for the case when no overflow occurs but it is not necessary to unload the bottom of the local PDL onto the global PDL when overflow occurs. It is simpler to pop the top of the local PDL and push it onto the global PDL.

3.2.3.2. Write-once Registers

As explained in earlier, the write-once registers are an important architectural innovation. They allow unification instructions that share variables to be dynamically scheduled in parallel. Elements of lists and structures are fetched by the unification units directly from memory. The control unit cannot assign tags and resolve data dependencies caused by variables shared by two or more lists or structures. The write-once registers resolve these dependencies. Each unification unit has a shadow copy of the write-once registers. These registers must be written using the distribution bus so that other units can update their write-once register sets.

The first implementation of the unification unit had a single set of write-once registers. The CU stopped dispatching instructions when a *synch* instruction was received. After all pending instructions had completed, the CU reset the write-once registers and resumed dispatching instructions. The *synch* instruction thus stalled the pipeline and degraded the performance of the PUP.

In order to reduce the number of pipeline stalls due to the synch instruction, the current version of the unification unit has two sets of write-once registers. In this implementation each instruction dispatched by the CU has a set number (1 or 0) indicating the set of write-once registers associated with that instruction. Suppose the CU has been issuing instructions marked with set 0. When a *synch* instruction arrives, the CU waits until all pending instructions marked with either set 1 or set 0 are completed. The CU then resets the write-once registers for the set that is completed and issues instructions marked with that set. Now if another synch instruction arrives, the CU will have to wait for all the instructions marked with set 0 to complete before issuing new instructions. Separate signal lines for set 0 and set 1 from the CU and the UUs indicate when there are no instructions pending.

3.2.3.3. Dereference Locks

Write-once registers are used to ensure that only one UU binds a register. We described an example earlier that demonstrated the need for a mechanism to synchronize bindings of shared variables in memory. The architecture does not specify the mechanism to be used. In this implementation of the PUP we use dereference locks to ensure that only one UU binds an unbound variable in memory.

Dereference locks are a set of associative registers. There are shadow copies of this set in each of the UUs. Whenever a UU finds that a variable dereferences to an unbound variable, it first does an associative search in its copy of the dereference locks for the address of the unbound variable. If the search succeeds some other UU is in the process of

binding the variable and the UU must wait until the lock is released by the other UU. Then it must dereference the variable again since there may be a new binding. If the associative search does not succeed, the UU tries to distribute the address of the unbound variable to lock it. If it does not get the distribution bus it must monitor the bus to see if some other UU distributes the address of the same unbound variable. After acquiring the lock the UU may bind the variable. The dereference lock is released when the trail request is distributed. At present there are four dereference locks.

3.2.4. The Node Table and Global Push Down List

The node table stores instructions dispatched by the CU. Some of these instructions do not have valid operands. Such operands are marked as invalid and an operand tag is stored in place of the operand itself. An instruction may have two operands. An opcode, result tag and write-once register set number are also stored with each instruction. The node table monitors the distribution bus continuously. If the distribution bus cycle is used to distribute a result of a previous operation, the node table associatively compares the tag of the result to the operand tags of all the invalid operands. If any tags match the result is loaded into the operand and the operand is marked as valid. If an instruction has all its operands valid, it can be dispatched to a UU that requests an instruction. The current version of the node table contains 16 entries.

When the local PDL in a UU overflows, the UU pushes the top of the local PDL onto the global PDL by distributing the data and requesting a push operation. When a UU is free it requests the distribution bus for an instruction fetch cycle. If this request is granted, the NTPDL uses the cycle to output (fire) an instruction. If the global PDL is not empty the PDL is popped and the instruction on the top is output. If the global PDL is empty the next instruction that has its operands valid is dispatched from the node table. The cycle is unused if no instruction is ready. The global PDL is rarely used since lists and structures are usually not very deep and at present it has 64 entries.

3.3. An Example

In this section we describe the flow of instructions through the system for a single iteration of a simple program. The instruction trace does not correspond to an actual cycle-by-cycle flow and is only meant to illustrate the interaction between the various components of the hardware and the function performed by each of them.

The Prolog program described appends the list [d,e] to the list [a,b,c] to give the list [a,b,c,d,e]. The Prolog code is as follows.

```
main :- concat([a,b,c],[d,e],X).
concat([],L,L).
concat([X|L1],L2,[X|L3]) :- concat(L1,L2,L3).
```

The PUP code corresponding to the main call, which involves setting up the arguments and calling the concat procedure is as follows. Alphanumeric arguments are used for readability. For simplicity only one set of write-once registers are used.

```
main:   putlist    (l1,x1)
        putlist    (l2,x2)
        putvarx    (x3,x3)
        call       (concat)
ret:    quit
l1:     list       (size 4)
        const      (a,cdr 0)
        const      (b,cdr 0)
        const      (c,cdr 0)
        nil        (cdr 1)
l2:     list       (size 3)
        const      (d,cdr 0)
        const      (e,cdr 0)
        nil        (cdr 1)
```

The prefetch unit fetches the instructions and passes them on to the CU. The first instruction, *putlist(l1,x1)*, has X1 as the destination register. The CU assigns a unique tag, *a* to X1 and marks it invalid. The instruction *putlist(l1,a)* is then dispatched to the node table. There are two unification units, UU1 and UU2, in this example, and both are initially idle. Since the instruction *putlist(l1,a)* is ready, the NTPDL dispatches it to the first UU that requests it, say UU1. Similarly, the next instruction is dispatched via the node table to the second unification unit, UU2. The CU also assigns a tag to the destination register of the third instruction, X3, and dispatches the instruction to the node table. Since both the UUs are now busy, however, this instruction waits in the node table until one of the UUs finishes an instruction and requests another. This state of the PUP is shown in figure 7(a).

The PFU then decodes the *call(concat)* instruction and starts prefetching instructions starting at the address *concat*. The continuation pointer, CP, in the CU gets loaded with the return pointer, *ret*, when the CU executes the *call* instruction. This address is supplied

by the PFU along with the *call* instruction. UU1 requests 4 words of heap space by adding 4 to its copy of the heap pointer, HP, and distributing the result. On seeing the distributed HP value, all the other units with HP copies (the CU and UU2) update their copies. UU1 then creates a list on the heap and distributes a pointer to the beginning of that list with a type tag of *list*. The distribution result is identified with tag *a* and the CU loads X1 with the pointer and type tag being distributed since X1 has the result tag *a*. UU1 then requests a new instruction from the NTPDL and gets the instruction *putvarx(c)*. Figure 7(b) illustrates the state of the PUP at this point. All the arguments are now ready for the *concat* procedure.

Similarly, UU2 creates a list on the heap as well and distributes a pointer to it as a result and this pointer gets loaded into X2. UU1 also creates an unbound variable on the heap and distributes a pointer to it. This gets loaded into X3. Meanwhile, the PFU decodes the first instruction at address *concat*. The code for the procedure *concat* is given below.

```
concat:   swot      (1,lv,lc,ll,ls)
lv:       tryelse   (lc,f)
lc:       getvalx   (x2,x3)
          getconst  (nil,x1)
          proceed
f:        trust     (ll)
ll:       synch
          getlist   (l3,x1)
          getlist   (l4,x3)
          putwx     (w1,x1)
          putwx     (w3,x3)
          execute   (concat)
ls:       fail
l3:       list      (size 2)
          bvar      (cdr 0,w4)
          bvar      (cdr 1,w1)
l4:       list      (size 2)
          bvar      (cdr 0,w4)
          bvar      (cdr 1,w3)
```

The first instruction of the *concat* procedure is *swot(1,lv,lc,ll,ls)*. The PFU decodes this instruction and passes the argument register number, 1, to the CU. The CU dereferences

the argument and passes back the result type, in this case *list*, to the PFU. The PFU then starts to prefetch instructions starting at the address *ll*. The first instruction at this address is *synch*. This instruction causes the CU to wait until all the previous instructions are done and then reset the write-once registers. The state at this stage is shown in figure 7(c).

The next two instructions, *getlist(l3,list1)* and *getlist(l4,bdvar8)* are dispatched by the CU to the node table in place of the instructions *getlist(l3,x1)* and *getlist(l4,x3)* since X1 contains the pointer to the list starting at address 1 and a type tag of *list*, and X3 contains a type tag *bdvar* (bound variable) with an address 8. These two instructions are dispatched to the UUs. In figure 7(d), the CU has also dispatched the instruction *putwx(w1,x1)* to the node table as *putwx(w1,a)* and assigned X1 the tag *a*. It has just fetched the instruction *putwx(w3,x3)* from the PFU.

The PFU decodes the *execute(concat)* instruction and starts prefetching from the *concat* address while UU1 unifies the list at 1 with the list at *l3* and UU2 unifies the bound variable at 8 with the list at *l4*. Figure 7(e) shows the state of the PUP after UU1 has completed the *getlist(l3,list1)* instruction. Write-once register W4 gets loaded with the first element of the list at 1, a constant *a*. Write-once register W1 gets loaded with a list tag and a pointer to the rest of the list.

The argument of the *getlist* instruction in UU2 is a bound variable that dereferences to an unbound variable at address 8 on the heap. This is unified by UU2 to the list at *l4*. Two heap space locations are requested since the size field at *l4* is 2. Then the location 8 is written with a *list* tag and a pointer to the first of the 2 heap locations just allocated. Since W4 has already been written, it is copied into the first element of the list on the heap. The second element is marked as a *cdr* element (its **cdr** bit is set) and and written with tag of unbound variable. This state is shown in figure 7(f).

The subsequent putwx instructions load the X1 register with a *list* tag and an address 2, and the X3 register with a *list* tag and an address 10. The arguments for the next iteration of the *concat* procedure are now ready and the state is shown in figure (g).

4. Measurements and Analysis

Like most complex systems, optimization of the PUP architecture is a problem involving several variables that interact in ways that are difficult to model. We have developed a register transfer level simulator to verify the correctness of the architecture and to measure the results of our design choices. This section describes the simulator and reports the measurements taken on a set of standard Prolog benchmarks running on the simulator.

4.1. The Simulator

A register transfer level simulator is required to verify the implementation of the architecture and to get detailed traces and measurements. We used a design environment provided by Endot Inc. called N.2. This includes a compiler and simulator for the ISP hardware description language and tools to write an assembler and linking loader. The simulator includes an interactive command set, tracing facilities and tools to aid in collecting statistics.

The PUP simulator includes mechanisms to vary memory access time and arbitration policy, number of unification units (1 to 4) and number of write-once register sets (1 or 2). The target instruction cache in the PFU can also be enabled or disabled. These mechanisms allow us to observe the effects of changing system parameters on the performance and identify bottlenecks.

4.2. Benchmarks and Assumptions

Our measurements were obtained by running benchmarks from the Warren and Berkeley Benchmark sets. It is important to note that all goals that perform input and output have been deleted from these benchmarks, since these goals require interaction with the operating system. Most of these benchmarks are rather small. The **con1** benchmark concatenates a list of two elements to a list of three elements. The **con6** program finds all pairs of lists which, when concatenated, result in a given list of five elements. The **nrev1** benchmark reverses a list of 30 elements and **hanoi** solves the Towers of Hanoi problem for 8 discs. The **qs4** program sorts a list of 50 elements using the quicksort algorithm and **queens** solves the queens problem for 4 queens on a 4 x 4 chess board. The **palin25** benchmark detects a palindrome that is 25 characters long.

While we believe that execution time is the primary metric that should be used to measure performance, we also include the equivalent LIPS (logical inferences per second) rate because it has unfortunately become customary to do so. Figures in parenthesis are KLIPS rates in all the performance tables. Execution time is presented in cycles. We expect a 1.6 micron CMOS implementation to easily achieve a clock rate of less than 80ns. In our LIPS measures, however, we use a 100ns clock which is the same as that used for the PLM.

It should be noted that all PUP measurements include the effects of bus contention and stalls due to the PFU buffer being empty or a miss in the target instruction cache. The PLM measurements are for single cycle memory and do not include delays due to an empty prefetch buffer.

In the performance tables and figures the following notation is used. PUP(2f1s) is the PUP with 2 unification units, fixed priority scheme (UUs get first priority, the CU is next and PFU gets the memory bus only if no other unit requests it), 1 cycle memory access and pipelined **synch** instructions (2 write-once register sets). PUP(4r3n) is the PUP with 4 UUs, round-robin memory arbitration, 3 cycle memory access, one set of write-once registers and no target instruction cache in the PFU.

4.3. Number of Unification Units

Figure 8 shows the performance of the PUP as a function of the number of unification units for different memory access times. Two write-once register sets and a TIC (target instruction cache) are assumed for all measurements. The performance figures are relative to the PUP(4f1s) which is the fastest configuration for the benchmark set as a whole. The total execution time is the sum of the execution times in cycles for each of the benchmarks.

We make two important observations from figure 8. The first is that the performance increases dramatically when the number of unification units is increased from one to two but less significantly when the number of unification units is increased from two to three and negligibly when a fourth UU is added. The second observation is that the performance increase when a unification unit is added is greater at higher memory speeds. This occurs because contention for the memory bus is greater for slower memories and therefore additional UUs are less effective. Memory contention can also explain the first observation, although there are other important reasons as well. Contention for the distribution bus, the limited parallelism available due to inherent dependencies in the program, and the limited capacity of the CU to dispatch instructions are other reasons.

In some cases the performance may degrade slightly rather than improve when more UUs are added. This occurs because the sequence of instructions can change when more UUs are added and the execution path at the microinstruction level may be different. This phenomenon occurs rarely and is observed only if the memory is slow.

4.3.1. Utilization of CU and UUs

Table 4 shows the utilization (fraction of time the unit is busy) of the CU and each of the UUs for different configurations with varying numbers of unification units and memory speeds. Two write-once register sets and a TIC are assumed for all systems. Idle time for each of the units is measured by counting the number of cycles during which the unit is waiting for an instruction. The CU waits for instructions from the PFU, and the UUs wait for instructions from the NTPDL. Utilization is computed as the fraction of the total number of cycles that the unit is not idle. Time spent waiting for a memory access to

complete is not considered idle time since the unit is not free to do other useful work while it it is waiting. The cycle counts used are the sum of the cycle counts for each of the benchmarks. From table 4 it is apparent that only two unification units can be utilized more than 30% of the time.

4.4. Effect of Memory Access Time

Table 5 shows the relative performance (PUP(4f1s) = 1.0) of various PUP configurations and memory speeds. All PUP configurations in this table have two write-once register sets and a TIC. The performance of the PUP with 2 cycle memory is around 80% of that with 1 cycle memory irrespective of the number of UUs. Performance degradation is almost the same when 2 cycle memory is replaced with 3 cycle memory.

4.5. Effect of Memory Arbitration Algorithm

Memory arbitration algorithm has very little effect on the performance of the PUP as can be seen from table 6. Relative performance (PUP(4f1s) = 1.0) of systems with four UUs and one set of write-once registers are shown for fixed priorities and round-robin arbitration. In the fixed priority scheme, the UUs have the highest priority followed by the CU and then the PFU.

4.6. Effect of the Target Instruction Cache

As shown in table 7, the target instruction cache results in a substantial improvement in performance. Further tests need to be run on larger benchmarks and for a wider range of memory speeds in order to optimize the number of lines per set and the number of bytes in each line. The choice of a TIC and prefetch buffer in preference to an instruction cache also needs to be justified by a separate study. Some of the tradeoffs involved are presented in [15] for a single chip RISC processor. However, the tradeoffs may be different for a multiple chip implementation.

4.7. Effect of Multiple Write-Once Register Sets

The measured performance improvement due to a second set of write-once registers is surprisingly small. Table 8 compares relative performance (PUP(4fls)= 1.0) of systems with one and two write-once register sets and lists the performance improvement due to the additional set. The performance improvement is less than 3% and is higher when more UUs are used. We conclude that the stalls caused by the **synch** instructions do not degrade performance significantly.

4.8. Comparison With the Berkeley PLM

Table 9 compares the performance of the PUP with that of the Berkeley VLSI-PLM and the TTL-PLM. The PUP consistently outperforms the TTL-PLM by factors ranging from 1.39 to 2.44 and the VLSI-PLM by factors ranging from 1.26 to 2.22. The TTL-PLM requires the host to execute numerical computations. These computations are called *escapes*. The VLSI-PLM is an improved version of the TTL-PLM and it executes several numerical computations in microcode.

The PLM figures for both the TTL and VLSI versions were taken from simulation measurements that assume single cycle memory and no stalls due to instructions not being available in the prefetch unit. The PUP simulator takes into account stalls due to instructions not being available in the PFU. This is especially important for the smaller benchmarks like concat and nconcat since the PUP incurs misses in the target instruction cache initially and the time taken to handle the misses is a substantial fraction of the total execution time. KLIPS figures are given in parentheses and assume a 100 ns clock for all processors although we believe that a faster clock cycle is possible for the PUP using a 1.6 micron CMOS implementation.

5. Conclusions

The TTL-PLM was the first special purpose Prolog machine of the Aquarius project at Berkeley. The VLSI-PLM evolved from the TTL version and the PUP is based on what we learned from these earlier machines. The PUP exploits fine grain parallelism (which includes unification parallelism and parallelism among arithmetic and bookkeeping operations). Since it is not possible in general to predict how long a unification operation will take at compile time, we believe that dynamic scheduling of unification operations and out of order execution can result in greater speedup than static scheduling. The speedup achieved by the PUP is a lower bound on what can be achieved with dynamic scheduling of parallel unification and arithmetic operations and overlapped bookkeeping operations. The node table, write-once registers and dereference locks are the hardware structures used by

the PUP to resolve data dependencies while allowing dynamic scheduling and out of order execution. We have shown that these structures allow us to achieve substantial speedup.

The goal of the Aquarius project is a Prolog system that exploits parallelism at all levels in the execution hierarchy. The PUP effectively exploits fine grain parallelism. Eventually, we will incorporate suitable extensions and modifications to the PUP so that it can be used to exploit AND and OR parallelism as well.

Acknowledgements

This project evolved out of discussions with our colleagues in the Aquarius project and we thank them for their suggestions and criticisms. We would also like to thank Endot Inc. for the use of their N.2 design environment.

This work was partially sponsored by Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871, Monitored by Space & Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

References

1. T. L. Adam, K. Chandy and J. R. Dickson, A Comparison of List Schedules for Parallel Processing Systems, *Communications of the ACM*, December, 1974, 685-690.

2. J. Beer, Concepts, Design, and Performance Analysis of a Parallel Prolog Machine, *PhD thesis, Technical University*, Berlin, .

3. G. Boriello, A. Cherenson, P. B. Danzig and M. Nelson, RISCs or CISCs for Prolog: A Case Study, *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, October, 1987.

4. M. Carlton and P. V. Roy, A Distributed Prolog System with AND-Parallelism, *Proceedings of Hawaii International Conference on System Science 88*, Honolulu, Hawaii, January, 1988.

5. J. H. Chang, A. M. Despain and D. DeGroot, AND - Parallelism of Logic Programs Based on a Static Data Data Dependency Analysis, *Digest of Papers, Spring COMPCON 85*, February 25 - 28, 1985, 218 - 226.

6. W. Citrin, Parallel Unification Scheduling in Prolog, *PhD thesis, University of California, Berkeley (expected 1988)*, Berkeley, California, 1988.

7. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

8. J. S. Conery, The AND/OR Model for Parallel Interpretation of Logic Programs, *PhD thesis, Dept. of Information and Computer Science, University of California, Irvine*, 1983.

9. A. M. Despain and Y. N. Patt, The Aquarius Project, *Digest of Papers, COMPCON*, Spring 1984, 364-367.

10. T. Dobry, A High Performance Architecture for Prolog, *PhD thesis, University of California, Berkeley*, Berkeley, California, 1987.

11. C. Dwork, P. C. Kanellakis and J. C. Mitchell, On the Sequential Nature of Unification, *The Journal of Logic Programming 1* (June 1984), 35-50.

12. B. S. Fagin, A Parallel Execution Model for Prolog, *PhD thesis, Computer Science Division, Univ. of California, Berkeley*, November, 1987. Available as Tech. Report UCB/Computer Science Dpt./87/380.

13. R. Hasegawa and M. Amamiya, Parallel Execution of Logic Programs based on Dataflow Concept, *Proceedings of the International Conference on Fifth Generation Computer Systems, 1984*, 1984, 507-516.

14. M. V. Hermenegildo, An Abstract Machine for the Restricted AND-Parallelism of Logic Programs, *Third International Conference on Logic Programming*, July, 1986, 25-39.

15. M. D. Hill, Aspects of Cache Memory and Instruction Buffer Performance, *PhD thesis, University of California, Berkeley*, Berkeley, California, November, 1987. Also available as Report No. UCB/Computer Science Dpt. 87/381.

16. N. Ito, H. Shimizu, M. Kishi, E. Kuno and K. Rokusawa, Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog, *New Generation Computing 3* (1985), 15-41, OHMSHA, LTD and Springer-Verlag.

17. T. Moto-oka, H. Tanaka, H. Aida, K. Hirata and T. Maruyama, The Architecture of a Parallel Inference Engine - PIE, *Proceedings of the International Conference on Fifth Generation Computer Systems, 1984*, 1984, 479-488.

18. H. Mulder and E. Tick, A Performance Comparison Between PLM and an MC68020 Prolog Processor, *Technical Note no. CSL-86-302, Computer Systems Laboratory, Stanford University*, Stanford, California, September, 1986.

19. Y. N. Patt, W. Hwu and M. C. Shebanow, HPS, A New Microarchitecture: Rationale and Introduction, *Proceedings of the 18th International Microprogramming Workshop*, Asilomar, California, December, 1985.

20. J. Syre and H. Westphal, A Review of Parallel Models for Logic Programming Languages, *Technical Report CA-07, European Computer Industry Research Centre, GmbH, Arabellastr, 17, D-8000 Muenchen 81, West Germany*, 10 June 1985.

21. E. Tick, Studies in Prolog Architectures, *Phd thesis (also Technical Report No. CSL-Tech. Rep.-87-329, Computer Systems Laboratory, Stanford University)*, Stanford, California, June, 1987.

22. R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development 11* (1967).

23. D. H. D. Warren, An Abstract Prolog Instruction Set, *Technical Report 309, Artificial Intelligence Center, SRI International*, 1983.

24. N. S. Woo, The Architecture of the Hardware Unification Unit and an Implementation, *Proceedings of the 18th Annual Workshop on Microprogramming*, December 3-6, 1985, 89-98.

25. H. Yasuura, On Parallel Computational Complexity of Unification, *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984.

Percent of Time by Instruction Class					
Benchmark	U	I	P	C	O
boyer	36.37	16.37	14.64	9.05	23.57
browse	65.47	3.06	13.39	10.46	7.62
con1	62.88	7.81	0	10.54	18.77
con6	42.81	4.84	10.14	14.90	27.31
hanoi	42.36	3.24	22.34	19.42	12.64
mumath	65.98	7.32	14.85	10.29	1.56
nrev1	83.04	11.7	0	4.73	0.53
palin25	70.58	2.8	7.75	11.36	7.51
qs4	55.78	4.4	12.4	17.01	10.41
queens	58.69	0.78	16.03	13.60	10.9

U = (unification) get, put and unify

I = (indexing) switch-on -term, -constant, -structure

P = (procedure control) try, retry, trust, cut, cutd

C = (clause control) allocate, deallocate, call, execute, proceed, fail

O = (others) escape, arithmetic, etc

Table 1: Time spent by the VLSI-PLM on various instruction types

PUP Register Set	
X0-X7	argument registers
W0-W7	write once registers
P	program counter
CP	continuation pointer
E	environment pointer
B	backtrack pointer
TR	trail pointer
H	heap pointer
T	top of stack pointer

Table 2: PUP Registers

get	put
getvarx(x,a) getvary(y,a) getvalx(x,a) getvaly(y,a) getconst(c,a) getwx(w,a) getwy(y,w) getlist(l,a) getstruct(s,a)	putvarx(x,a) putvary(y,a) putvalx(x,a) putvaly(y,a) putconst(c,a) putwx(w,a) putwy(y,w) putlist(l,a) putstruct(s,a)
clause control	procedure control
allocate(n) dealloc call(p) execute(p) proceed fail	tryelse(t,e) retryelse(t,e) trust(t) cut cuts cutd(l)
indexing	arithmetic and miscellaneous
switch-on-term(x,v,c,l,s) switch-on-const(h,n) switch-on-struct(h,n)	synch is(x1,x2) eval(x1,x2) add(x1,x2) sub(x1,x2) gt(x1,x2) lt(x1,x2) gte(x1,x2) lte(x1,x2) escape(n)

Table 3: Partial List of Instructions

Utilization of CU and UUs				
		Memory Speed (cycles)		
No. of UUs	Unit	1	2	3
1	CU	0.8935	0.8844	0.8802
	UU0	0.6409	0.6629	0.6570
2	CU	0.8510	0.8404	0.8240
	UU0	0.4257	0.4660	0.4633
	UU1	0.5040	0.5025	0.4995
3	CU	0.8748	0.8132	0.7501
	UU0	0.1863	0.2131	0.2733
	UU1	0.3637	0.4185	0.3901
	UU2	0.4797	0.4753	0.4762
4	CU	0.8865	0.8143	0.7114
	UU0	0.1078	0.1530	0.1696
	UU1	0.1516	0.2062	0.2977
	UU2	0.3952	0.4098	0.3741
	UU3	0.4198	0.4212	0.3977

Table 4: Utilization of CU and UUs

Relative performance vs. Memory access time (PUP(41fs)=1.0)					
	Memory access time (cycles)				
No. of UUs	1	2	3	2cy/1cy perf.	3cy/2cy perf.
1	0.7677	0.6053	0.4952	0.788	0.818
2	0.9691	0.7711	0.6171	0.796	0.800
3	0.9933	0.8038	0.6019	0.809	0.749
4	1.0000	0.8062	0.6214	0.806	0.771

Table 5: Effect of Memory Access Time on Performance

Relative Performance. PUP(4f1s) = 1.0			
	Memory access time (cycles)		
arbit. alg.	1	2	3
fixed	0.9740	0.7867	0.5992
round-robin	0.9689	0.8141	0.5925

Table 6: Effect of Memory Arbitration Algorithm

system	total cycles without TIC	total cycles with TIC	percentage improvement
PUP(1f2)	166284	142710	14.2%
PUP(2f2)	141217	114437	19.0%
PUP(3f2)	135407	109884	18.8%

Table 7: Effect of Target Instruction Cache on Performance

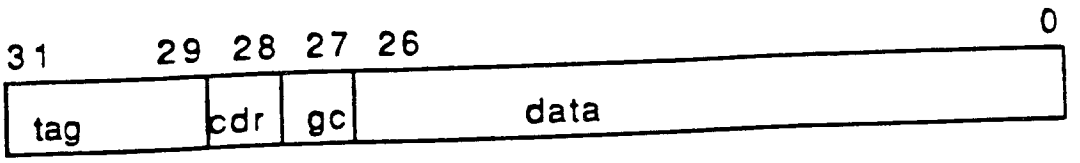
Relative Performance. PUP(4fls) = 1.0				
No. of UUs	Mem. Speed (cycles)	1 W-reg set	2 W-reg sets	improvement
1	1	0.7668	0.7677	0.0009
	2	0.6039	0.6053	0.0014
	3	0.4947	0.4952	0.0005
2	1	0.9501	0.9691	0.0190
	2	0.7530	0.7711	0.0181
	3	0.6005	0.6171	0.0166
3	1	0.9650	0.9933	0.0283
	2	0.7842	0.8038	0.0196
	3	0.5983	0.6019	0.0036
4	1	0.9740	1.0000	0.0260
	2	0.7867	0.8062	0.0195
	3	0.5992	0.6214	0.0222

Table 8: Effect of Multiple Write-Once Register Sets on Performance

Performance in cycles (KLIPS)					
benchmark	TTL-PLM	VLSI-PLM	PUP(4fls)	VLSI-PLM/PUP	TTL-PLM/PUP
con1	200(200)	180(222)	143(280)	1.26	1.39
con6	1187(50)	872(69)	487(123)	1.79	2.43
nrev1	24124(206)	21130(235)	15554(319)	1.36	1.55
hanoi	64865(118)†	52334(147)	28122(273)	1.86	2.30
qs4	47224(129)†	47709(128)	21434(285)	2.22	2.20
queens	50138(77)†	35435(109)	20521(188)	1.73	2.44
palin25	31296(95)†	26684(112)	14281(209)	1.86	2.19
total	219034(117)	184344(139)	100055(256)	1.84	2.18

† The TTL-PLM simulator does not count cycles for numerical escapes.
The actual cycle counts would therefore be higher than these.
In hanoi 6% of the instructions are escapes, in qs4 3.5% are escapes,
queens contains 11% escapes and in palin25 2.7% of the instructions
are escapes.

Table 9: Comparison with the Berkeley PLM



tag	type
000	unbound variable
001	bound variable
010	structure
011	list
100	constant atom
101	constant integer
110	constant (other)
111	constant nil

cdr	
0	not end of list
1	end of list

gc = garbage collection bit
(not used)

Figure 1: Fields of a data word

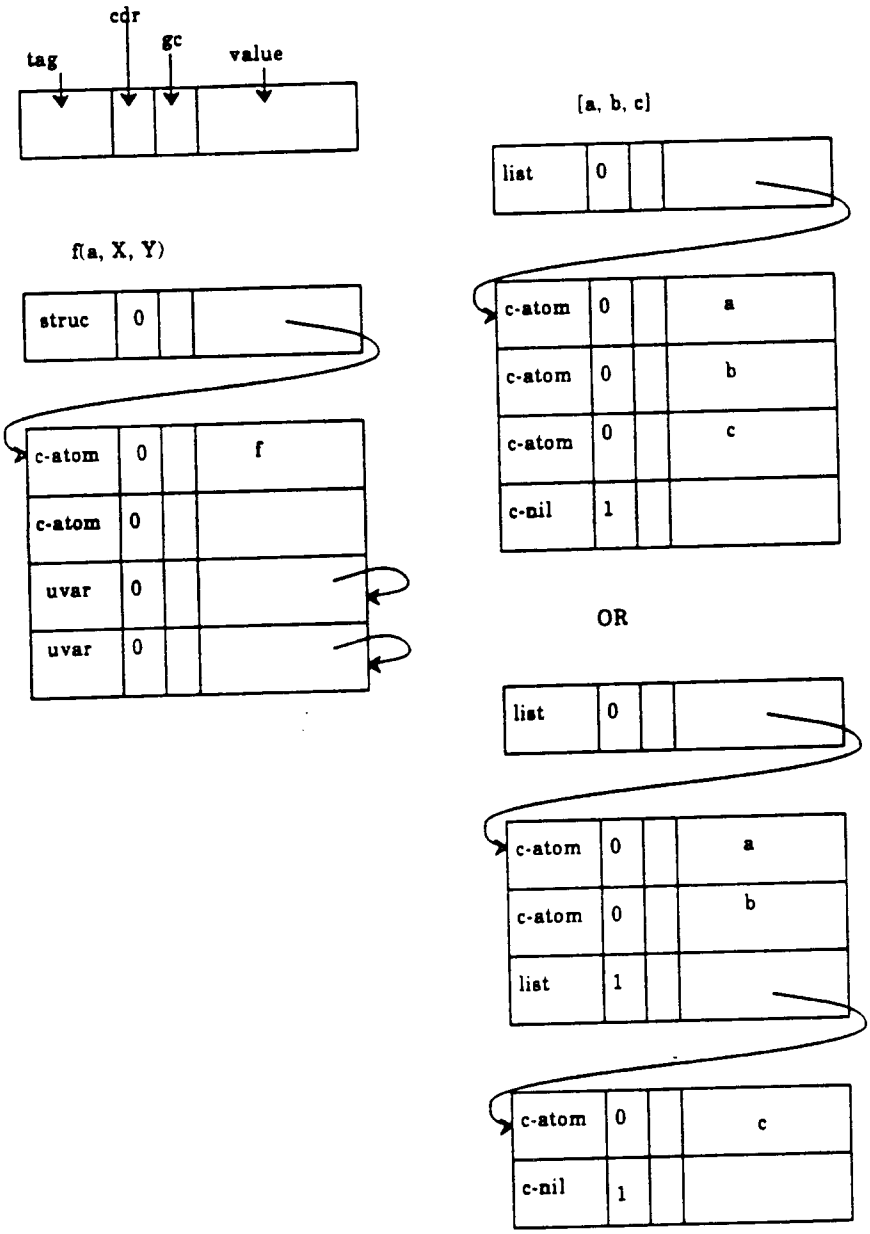


Figure 2: Cdr Coded Representation

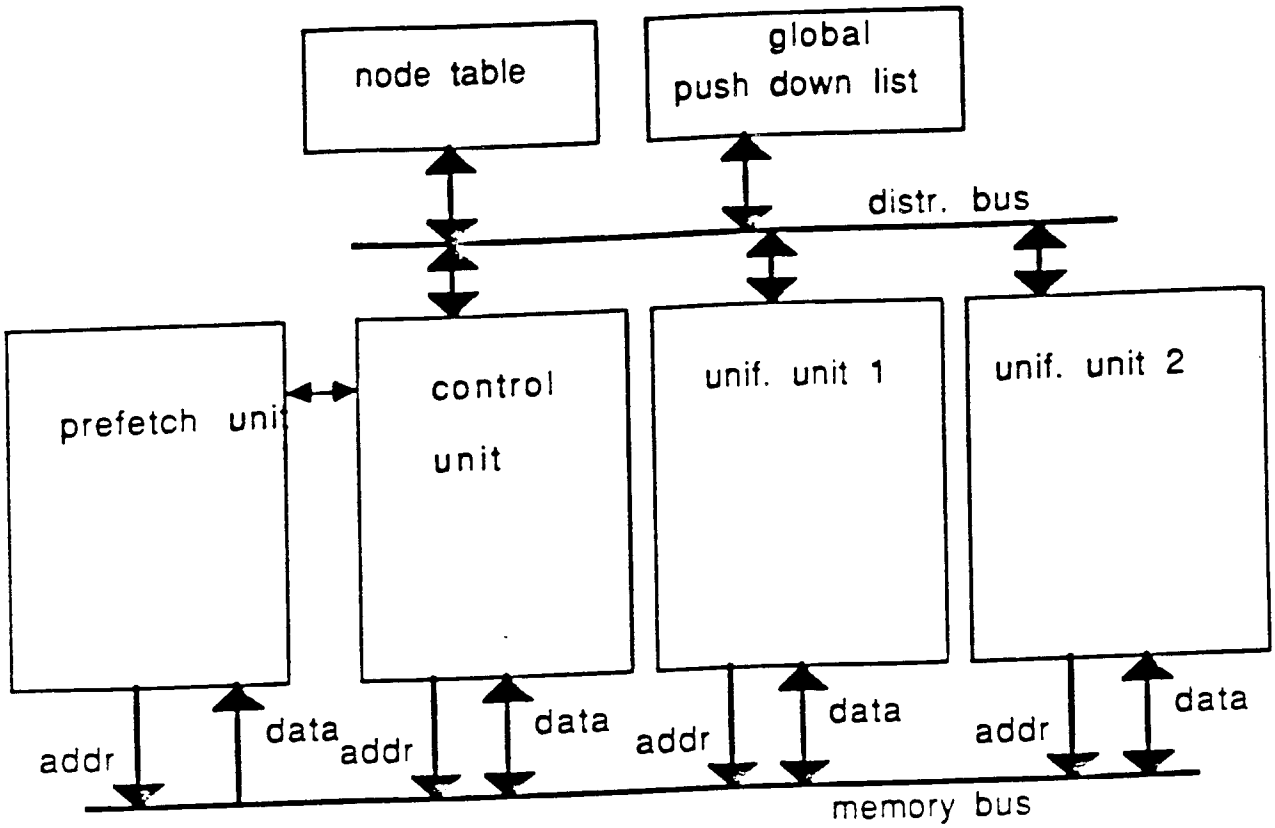
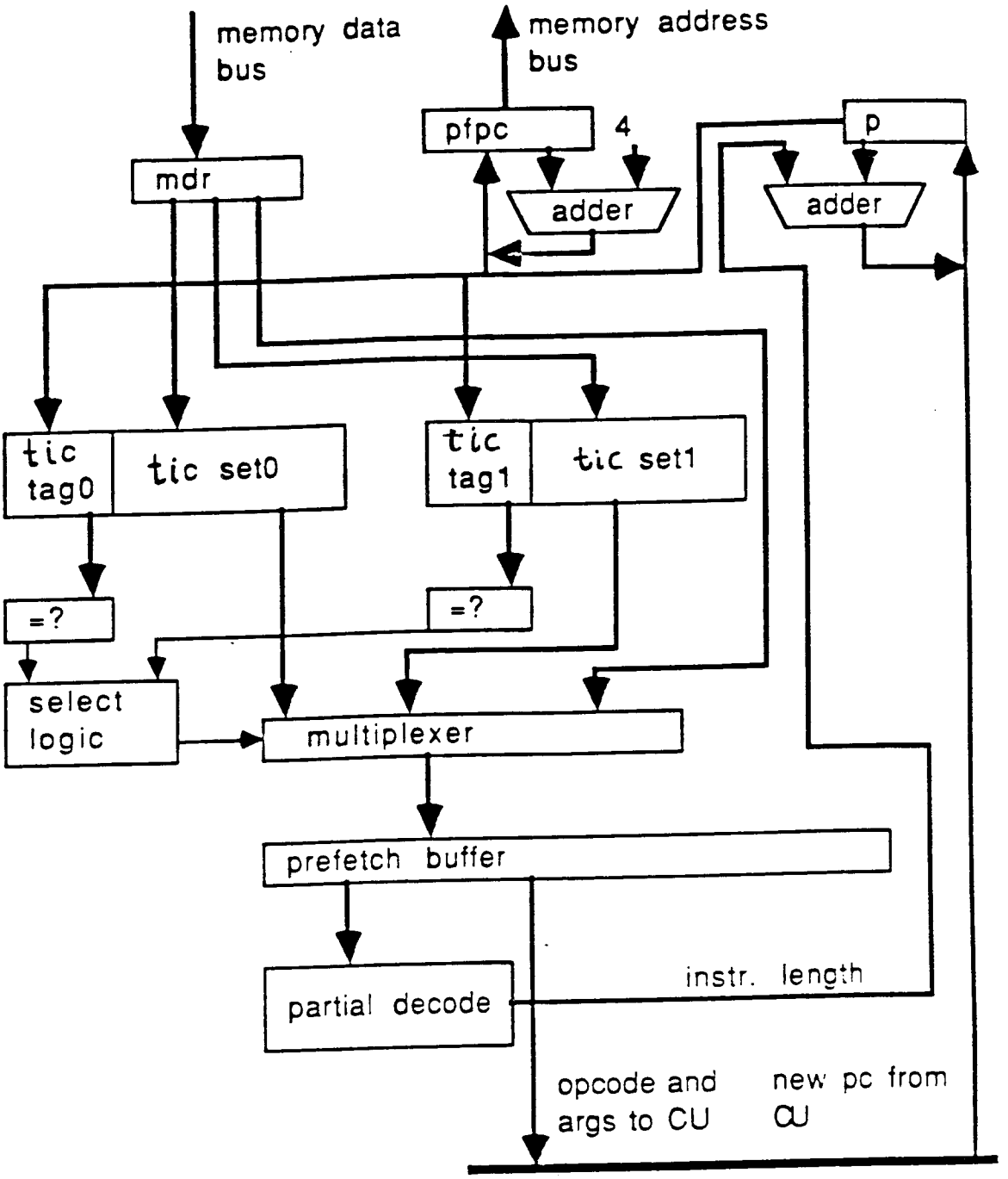


Figure 3: System Overview



p = program counter
pfpc = prefetch program counter
tic = target instruction cache

Figure 4: Prefetch Unit block diagram

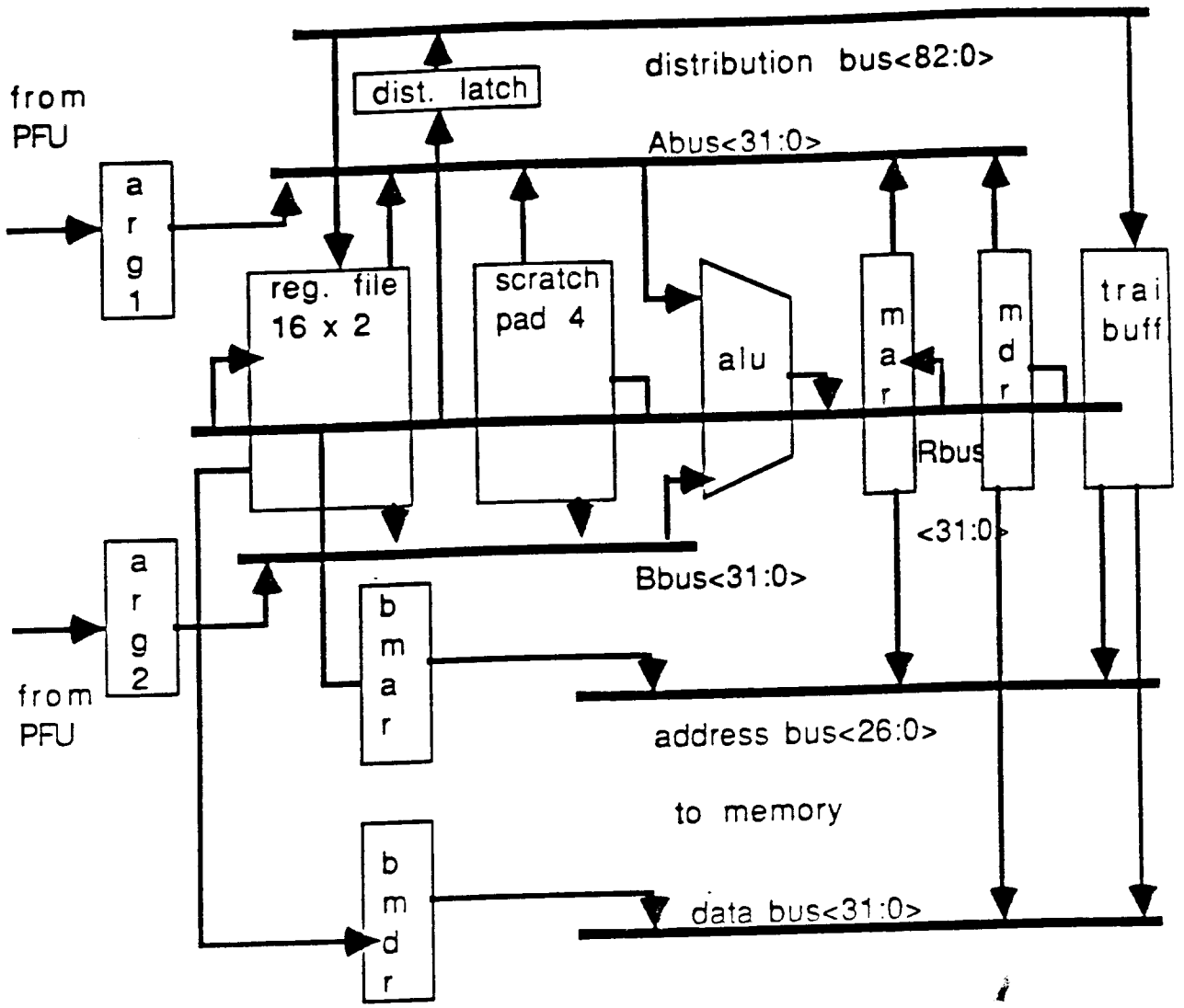


Figure 5: Control Unit Data Path

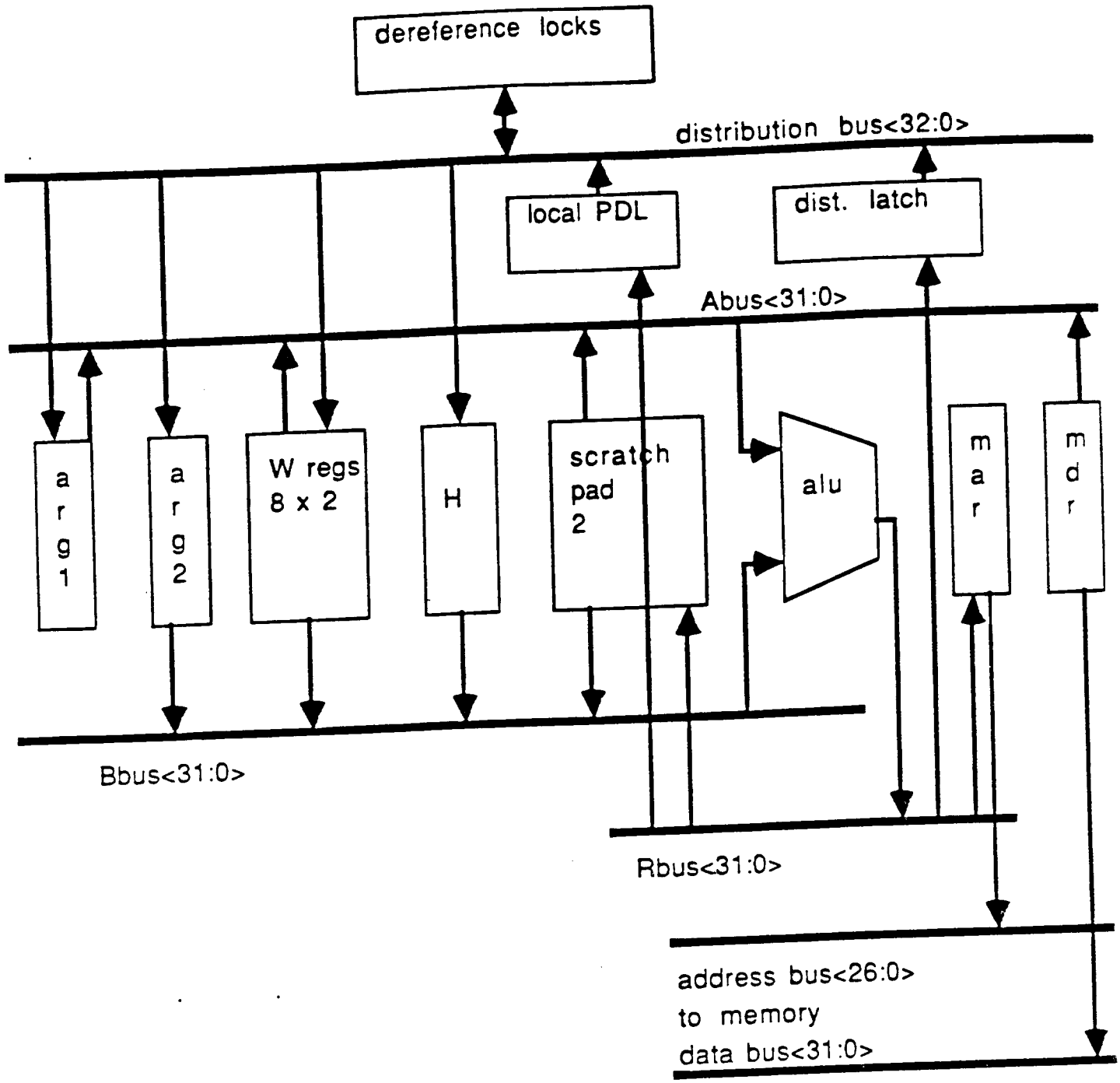


Figure 6: Unification Unit Data Path

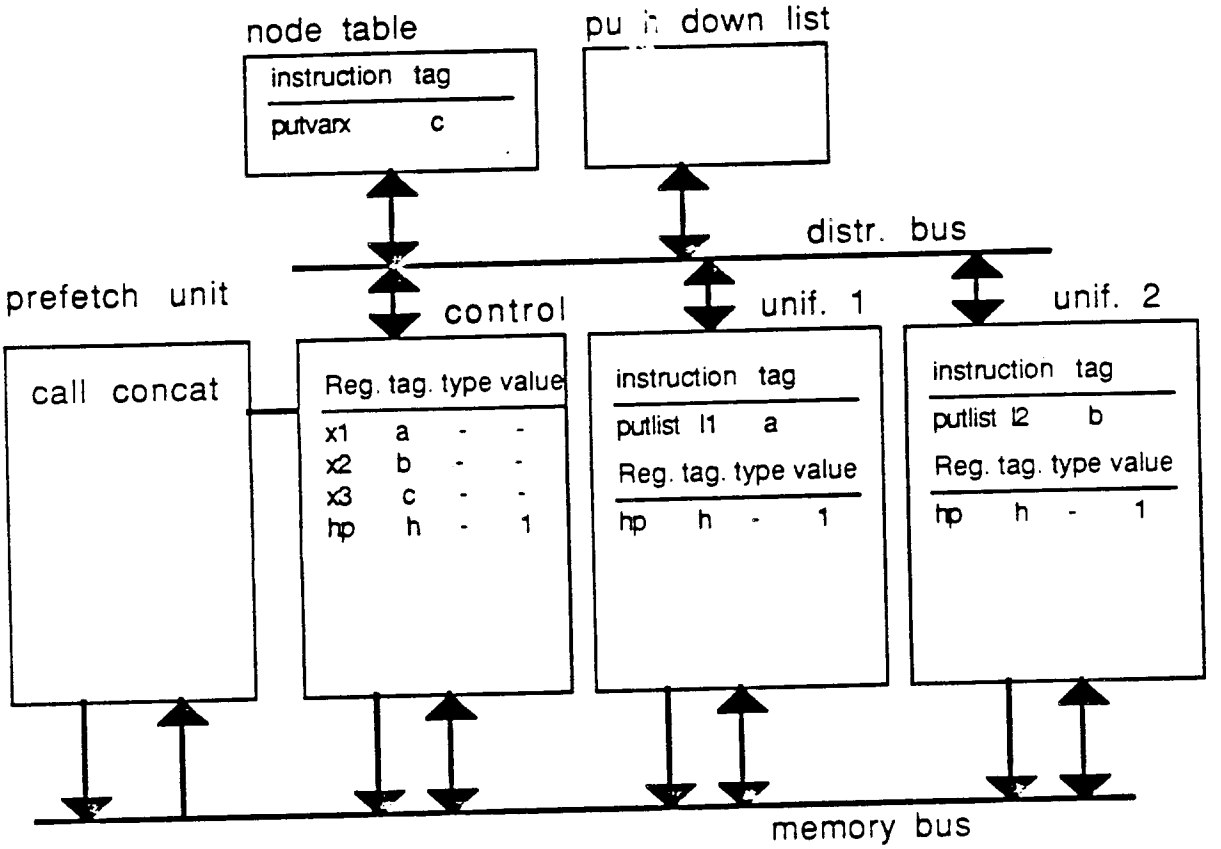
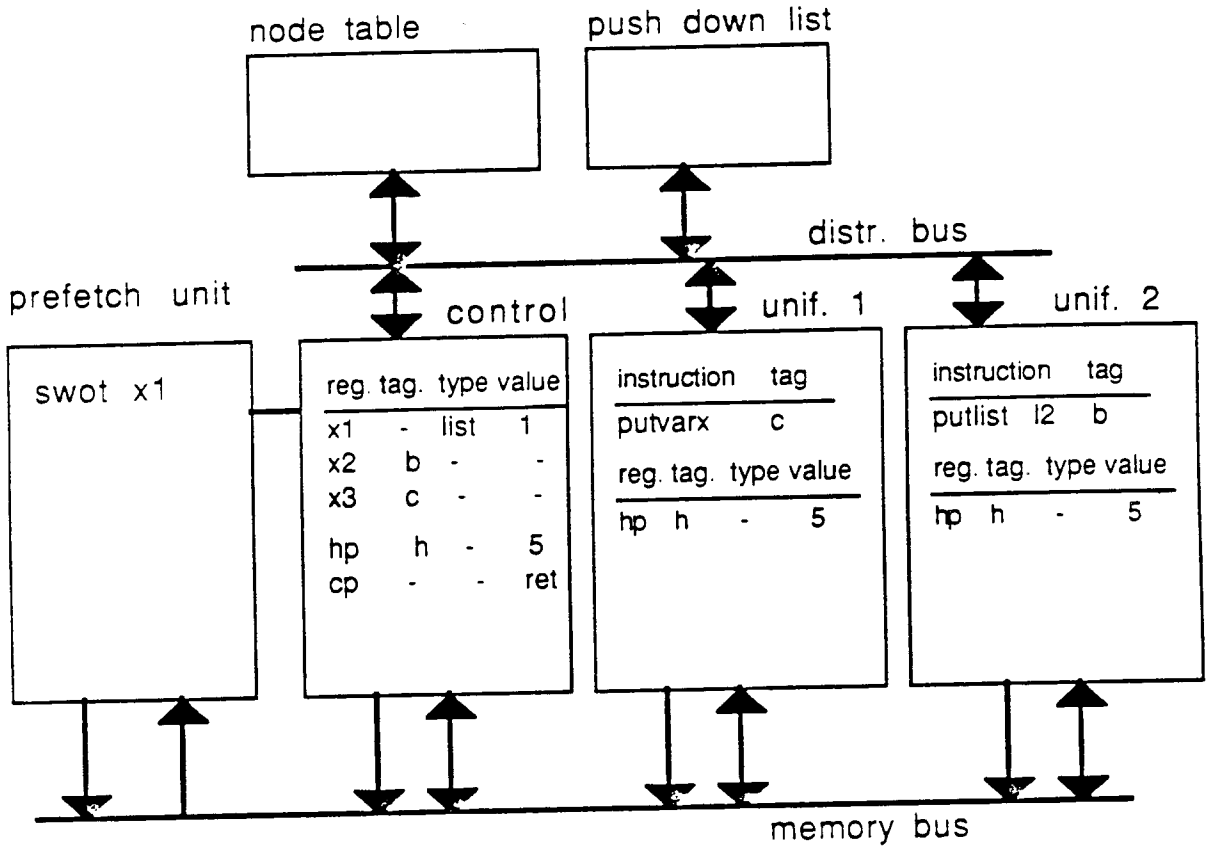


Figure 7(a): Example

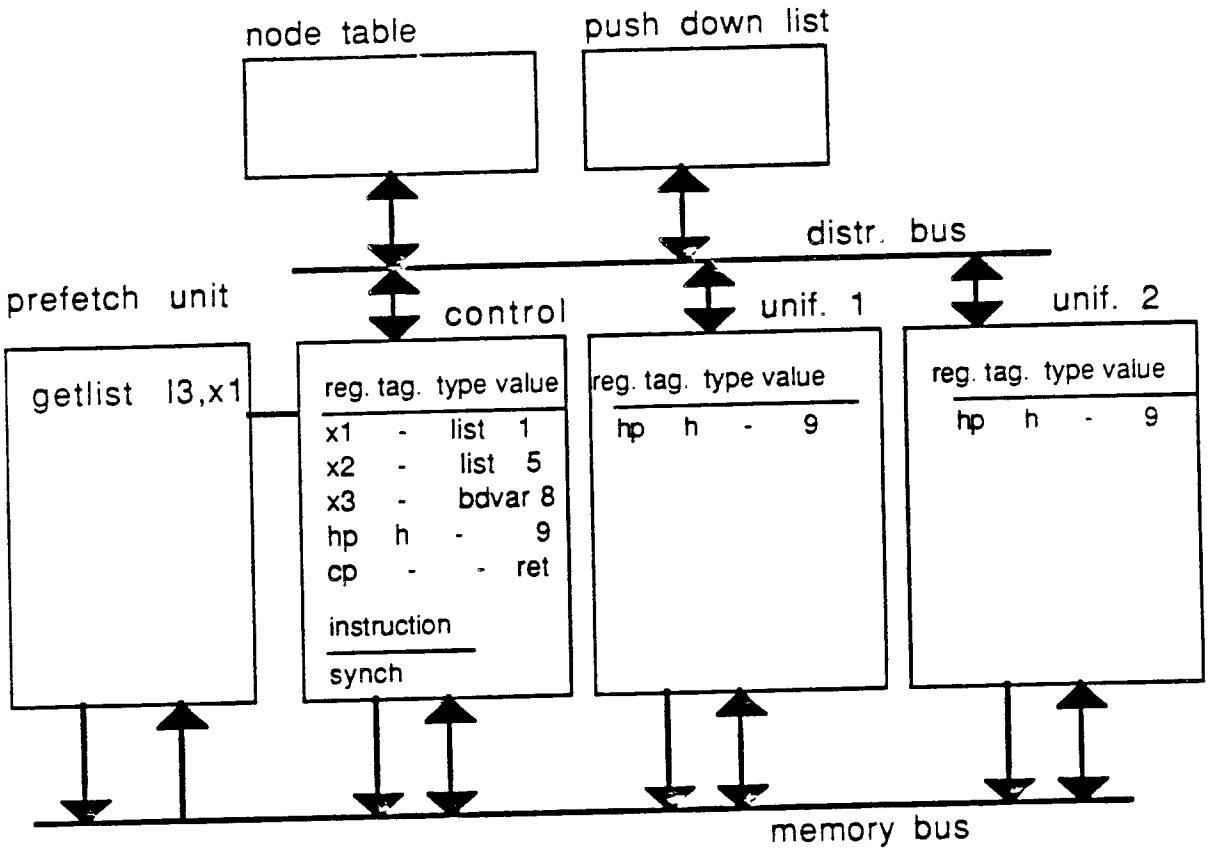


HEAP

1 : const a
2 : const b
3 : const c
4 : nil cdr 1
5:

cp = continuation pointer
hp = heap pointer

Figure 7(b): Example

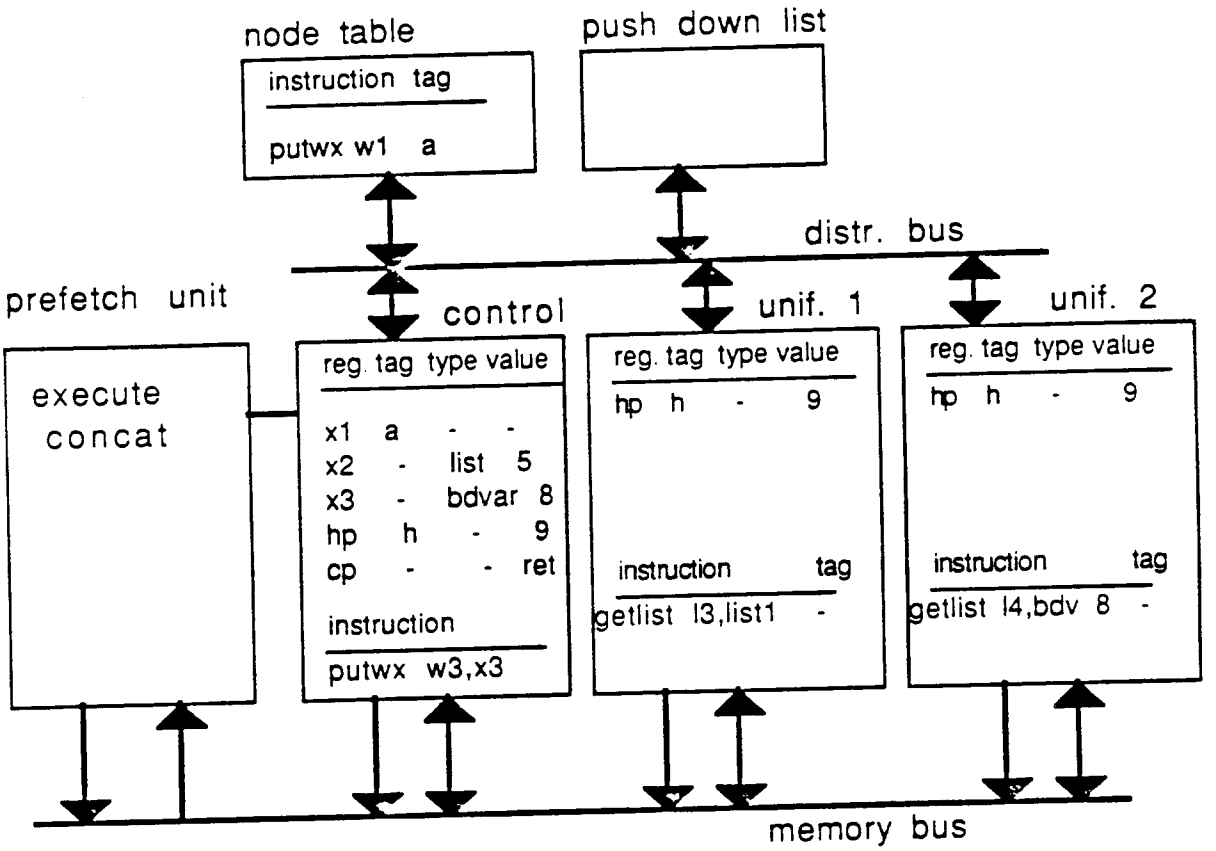


HEAP

1 : const a
2 : const b
3 : const c
4: nil cdr 1
5: const d
6 : const e
7 : nil cdr 1
8 : ubd var 8
9:

cp = continuation pointer
hp = heap pointer

Figure 7(c) Example

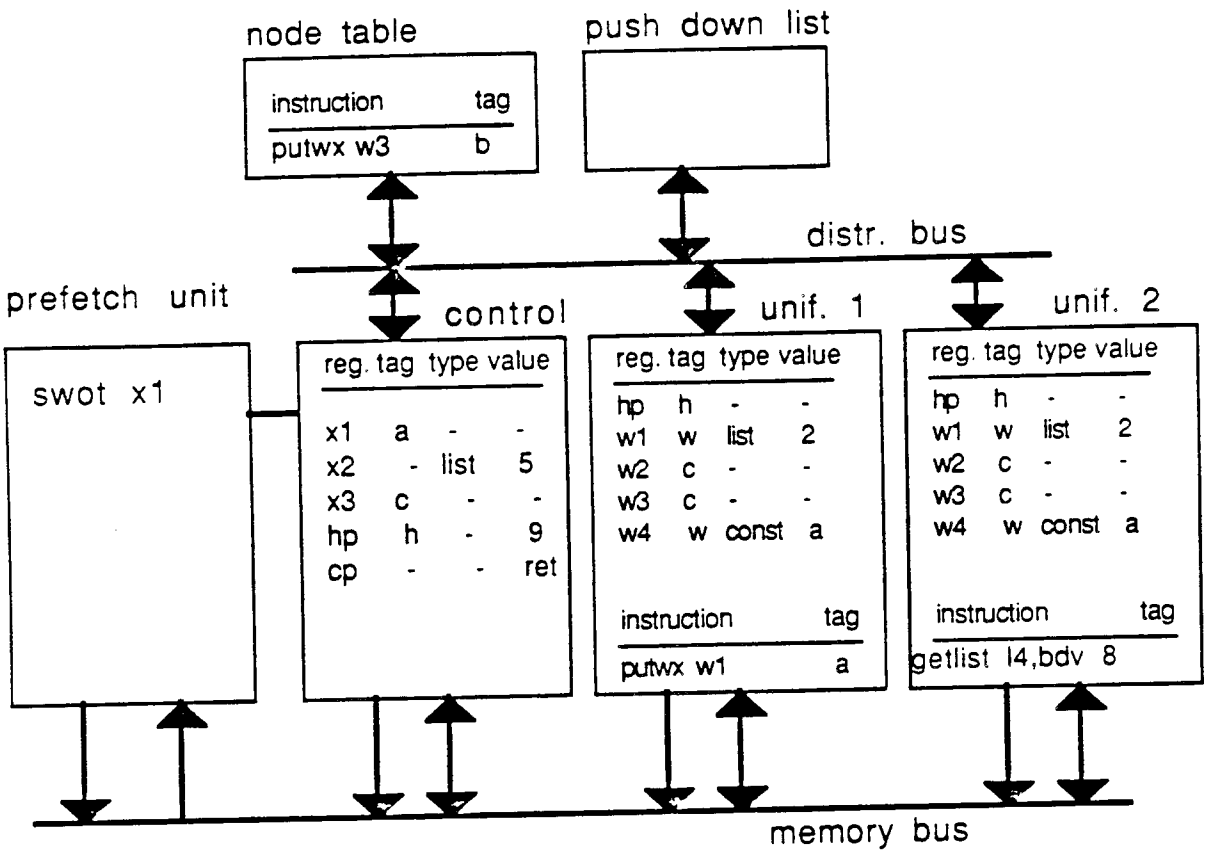


HEAP

1 : const a
2 : const b
3 : const c
4 : nil cdr 1
5 : const d
6 : const e
7 : nil cdr 1
8 : ubd var 8
9 :

cp = continuation pointer
hp = heap pointer

Figure 7(d): Example

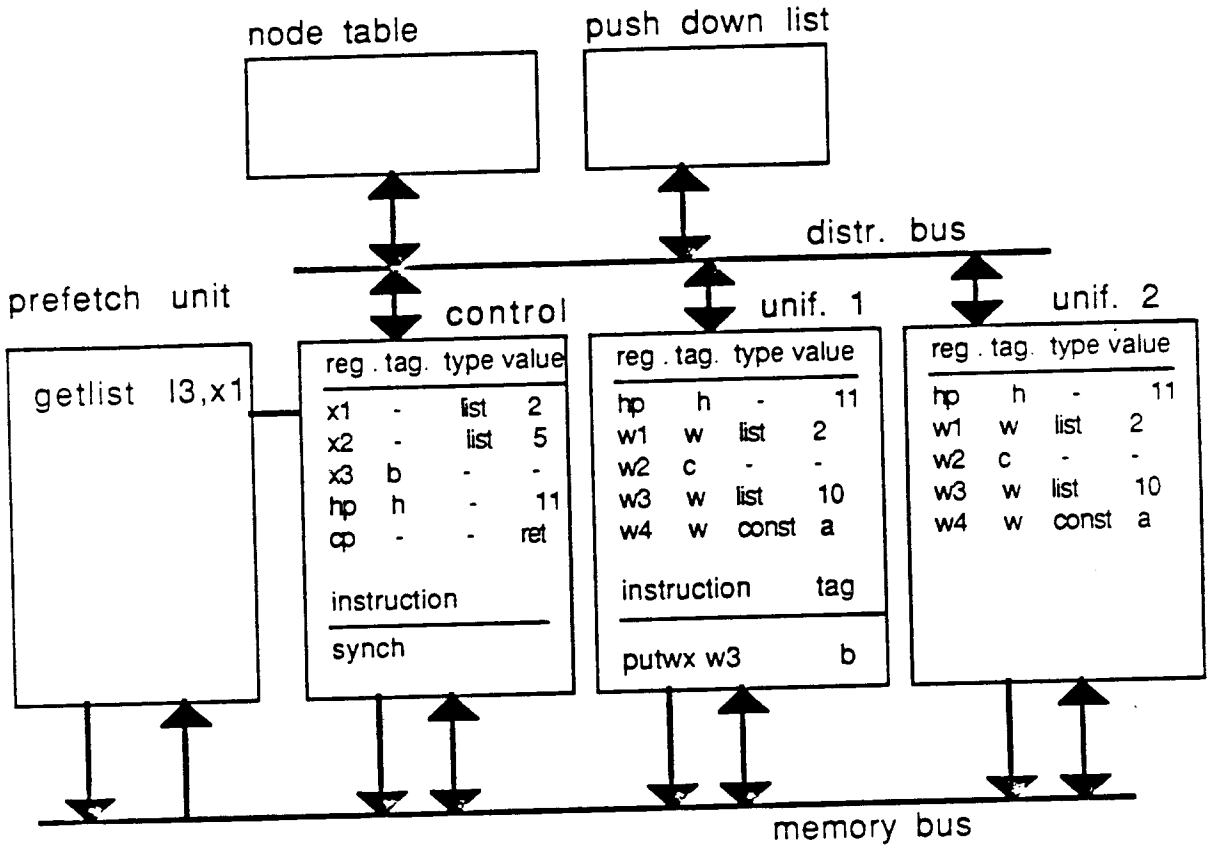


HEAP

1 : const a
2 : const b
3 : const c
4 : nil cdr 1
5 : const d
6 : const e
7 : nil cdr 1
8 : ubd var 8
9:

cp = continuation pointer
hp = heap pointer

Figure 7(e): Example

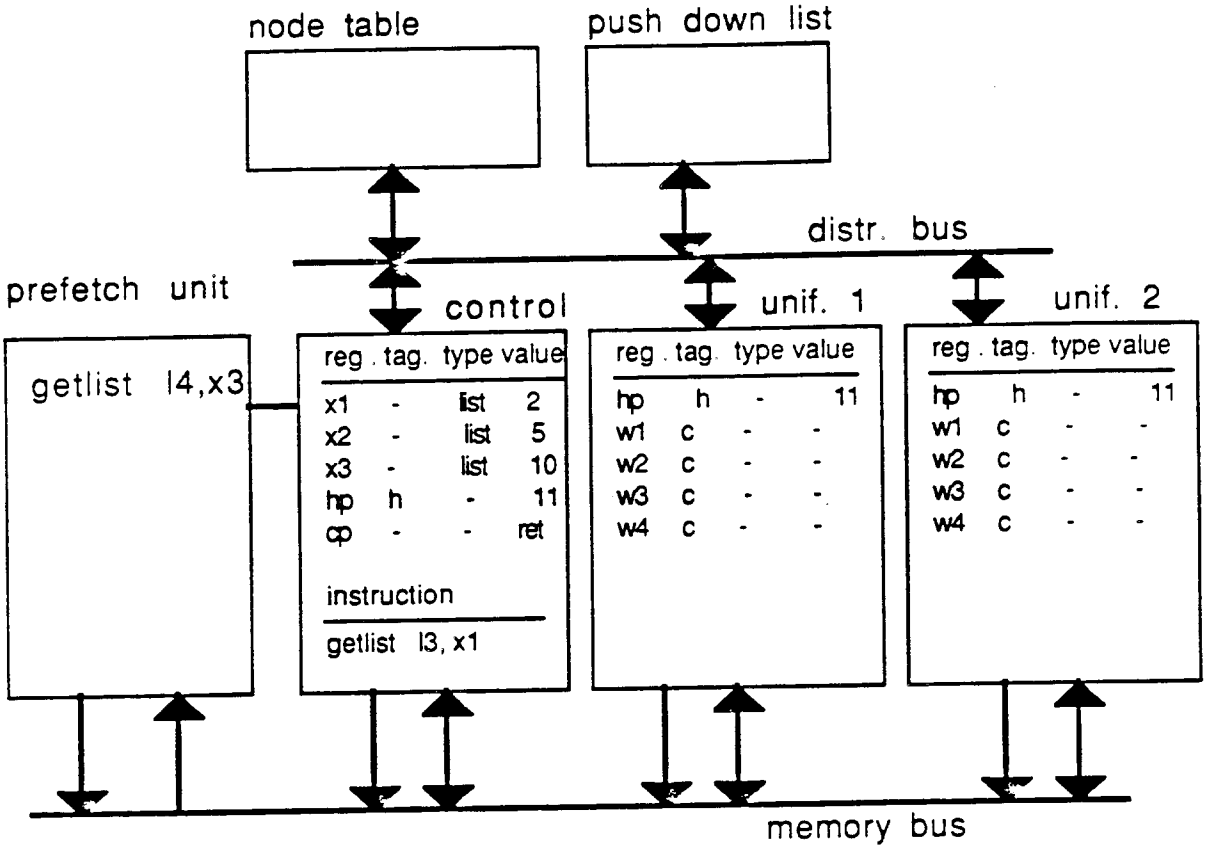


HEAP

1 : const a
2 : const b
3 : const c
4 : nil cdr 1
5 : const d
6 : const e
7 : nil cdr 1
8 : list 9
9 : const a
10: ubd var 10 cdr 1
11:

cp = continuation pointer
hp = heap pointer

Figure 7(f): Example



HEAP

1 : const a
2 : const b
3 : const c
4 : nil cdr 1
5 : const d
6 : const e
7 : nil cdr 1
8 : list 9
9 : const a
10: ubd var 10 cdr 1
11:

cp = continuation pointer
hp = heap pointer

Figure 7(g): Example

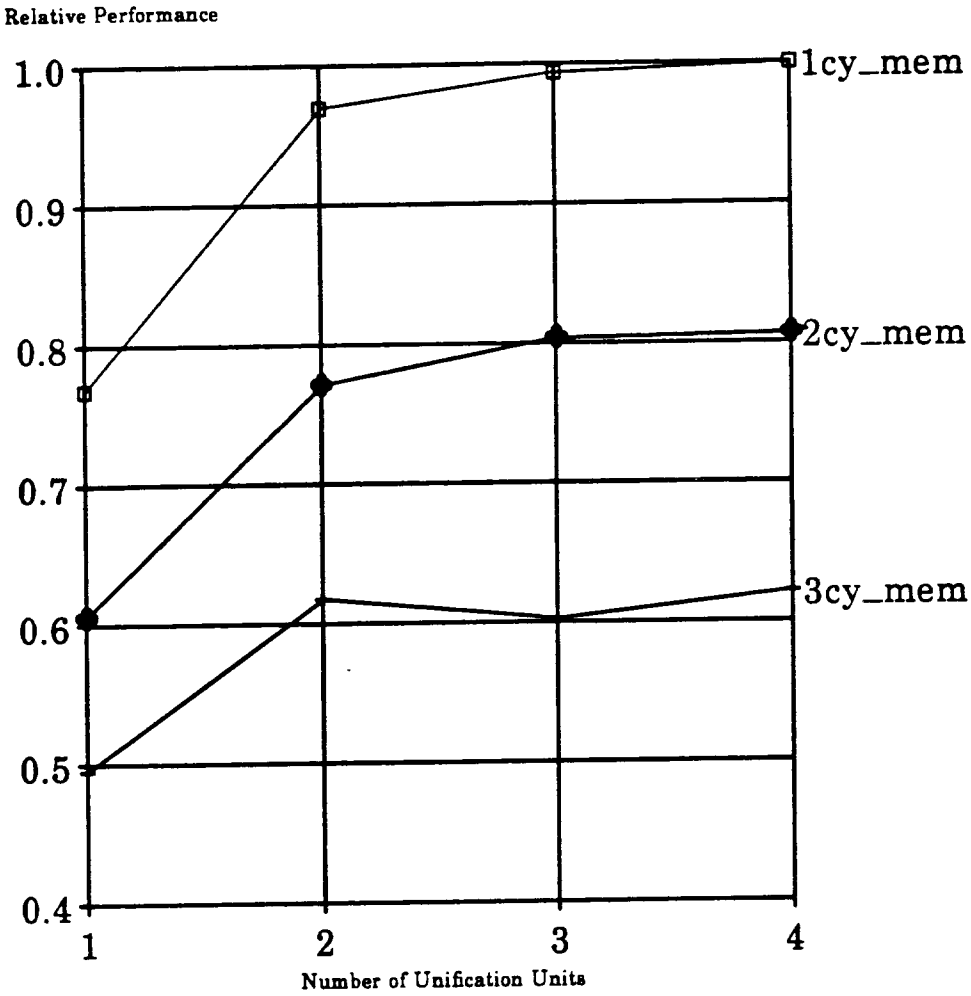


Figure 8: Performance vs. number of unification units