A High Performance Prolog Processor with Multiple Function Units

Ashok Singhal

Computer Science Division
University of California
Berkeley, CA 94720

Yale N. Patt

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2110

ABSTRACT

We describe the Parallel Unification Machine (PLUM), a Prolog processor that exploits fine grain parallelism using multiple function units executing in parallel. In most cases the execution of bookkeeping instructions is almost completely overlapped by unification, and the performance of the processor is limited only by the available unification parallelism. We present measurements from a register transfer level simulator of PLUM. These results show that PLUM with 3 Unification Units achieves an average speedup of approximately 3.4 over the Berkeley VLSI-PLM, which is usually regarded as the current highest performance special purpose, pipelined Prolog processor. Measurements that show the effects of multiple Unification Units and memory access time on performance are also presented.

1. Introduction

The growing interest in logic programming and Prolog, has resulted in substantial research towards the design of high performance Prolog systems by taking advantage of parallel hardware. Many research groups are trying to exploit parallelism available in Prolog programs by executing parallel processes on multiple processors [2,5,8-10]. Others have tried to exploit parallelism of a finer grain size by overlapping the execution of instructions of a single Prolog process by means of special purpose, pipelined processors [6,14]. Since the pipelines in these processors are quite short, instructions are overlapped only to a very limited extent. A large fraction of the execution time of such processors is spent on control and bookkeeping instructions, instead of unification (an operation similar to pattern matching that performs most of the "useful work" in a Prolog program). In this paper we demonstrate that a processor with multiple function units can overlap execution of many more instructions, and thus achieve a much larger speedup over sequential pipelined processors. Bookkeeping operations can overlap almost completely with unification, and several unifications can also execute in parallel. We describe one such processor, the Parallel Unification Machine (PLUM), and present performance measures obtained by register transfer level simulation. Each function unit of PLUM can be implemented by a VLSI chip of moderate complexity with a clock frequency comparable to modern single chip microprocessors.

Before proceeding with the description of PLUM, we provide the necessary background for this paper: a brief description of Prolog, a description of the execution and storage model used by PLUM, an introduction to the sources of parallelism in Prolog programs, and a summary of relevant literature. In section 2 we describe the main principles used in the PLUM design. We describe PLUM's architecture and implementation in section 3. Simulation results and analysis are presented in section 4. Section 5 concludes the paper with a summary of our results and a discussion of work in progress.

1.1. A Brief Description of Prolog

Prolog programs consist of a collection of *clauses* and a *goal*. The first goal is also called a *query*. The program is executed by the Prolog system by trying to satisfy the goal using the clauses in the program. Clauses have a *head* and an optional *body* that consists of one or more goals. Goals and clause heads are represented by *terms*. Terms may be simple or complex. Complex terms are *structures* (a list is a special case of a structure); each consists of a *functor* (name and arity of the term) and one or more *arguments*. The arguments are themselves terms. Simple terms are *atoms* or *variables*. The *unification* of two terms is the process by which the variables in the terms are bound such that the two terms become identical. Prolog finds the smallest such set of bindings (this set is unique). A goal succeeds if it unifies with a clause head and if all the goals in the body of the clause also succeed when executed in sequence. If no such clause exists, the goal fails. In order to execute a goal, Prolog tries each clause in the program in sequence. Since a goal can never unify with a clause whose head has a different functor, only clauses that have the same functor in their heads need be tried. The collection of such clauses is called a *procedure*. Thus, a goal is a procedure call. If a procedure has more than one clause

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 0884-7495/89/0000/0195$01.50

that could potentially satisfy a goal, Prolog tries the candidate clauses in sequence until a clause succeeds or there are no more clauses to try. Before trying one of several candidate clauses, the state of the program must be saved in a *choicepoint* so that the state can be restored if the clause fails and another clause must be tried. The process by which the state is restored on failure is called *backtracking*. Backtracking implements Prolog's left to right, depth first search for a solution to the query.

1.2. PLUM's Prolog Execution Model

PLUM's execution model is based on that of the Warren Abstract Machine (WAM) [17]. Conventional procedural languages, such as C, have a stack on which call frames are stored. A call frame contains information such as the return address and procedure arguments. Values of registers may also be saved in the call frame by the caller so that the registers may be restored when control returns to the caller. PLUM also has a memory area in which call frames (called *environments*) are stored. Although we refer to this area as the "environment stack", it is not really a true stack. In conventional programming languages the call frame could be deallocated from the stack when control returns from the callee, and the stack space is reclaimed. Prolog, however, may have to backtrack to the environment of the procedure to try another clause. This can happen if a choicepoint is created after the environment frame is created, and in that case the memory space occupied by the environment cannot be reclaimed when control returns from the callee. Therefore, the current environment frame is not necessarily on top of the environment area, as required by a true stack.

Prolog also needs to provide storage for choicepoints to implement backtracking. Since backtracking uses a depth first search strategy, the choicepoints may be stored on a stack. Environments and choicepoints may be placed in separate stacks (as in PLUM) or in the same stack (as in the PLM [6]).

Prolog creates it data structures, including unbound variables in a memory area called the heap. The heap space is allocated and deallocated as a stack as a simple means of garbage collection. In addition to state saved in a choicepoint, Prolog needs to keep track of all bindings of variables on the heap made after each choicepoint so that these bindings can be undone when backtracking to the choicepoint. This is accomplished by saving the addresses of the variables that are bound after each choicepoint on another stack called the *trail*. The location of the top of the trail stack at the time the choicepoint is created is saved in the choicepoint. All variables whose addresses are in the trail stack above the location saved in the choicepoint are unbound when Prolog backtracks to the choicepoint. Yet another stack, the *push down list* (PDL), is used by the unification algorithm for nested lists and structures.

Arguments could be passed to procedures either in registers (in which case the argument registers must be saved in choicepoints), or in an environment frame in memory (see [15] for a comparison of the two methods). PLUM uses argument registers for reasons that will be explained in section 2.

1.3. Fine Grain Parallelism in Prolog

Several forms of parallelism can be exploited in Prolog programs [13]. AND parallelism is exploited when several goals of a clause are executed in parallel. OR parallelism is exploited when several clauses of a procedure are tried in parallel. The AND and OR branches of the solution tree are usually exploited by parallel processes. Parallelism of a finer grain is also present in Prolog. Unification parallelism is exploited when several arguments of the goal are unified in parallel with corresponding arguments of a clause head. Bookkeeping and control operations, such as choicepoint creation and environment allocation, can execute in parallel with unification. Since unification, bookkeeping and control operations usually execute in far fewer cycles than an AND or OR process, parallelism among them must be exploited with far less overhead in order to be useful. In PLUM fine grain parallelism is exploited by multiple function units.

1.4. Related Work

PLUM evolved out of experiments with PUP [3], which also used multiple function units to exploit fine grain parallelism in Prolog, and with HPS [12], a restricted data flow architecture that uses the Tomasulo algorithm [16] to control multiple function units. PLUM's register set and pipeline control design benefited from the design of the POPE processor [1]. POPE exploits fine grain parallelism only across procedure boundaries by executing each procedure on a separate processor. PLUM's storage model and abstract machine are based on the Warren Abstract Machine (WAM) [17]. Ito et al [11] and Hasegawa et al [9] have proposed data flow machine for logic programming languages that exploit fine grain parallelism. Citrin [4] proposes a static data dependency analysis to determine which unifications of a clause head are known to be independent at compile time, and can be scheduled to run in parallel.

2. Design Philosophy

PLUM's design is based on three main principles: using multiple function units to execute instructions in parallel, using data driven control of the function units so that operations may execute whenever their operands are available, and partitioning memory to increase available memory bandwidth and reduce the sharing of memory among function units. In this section we justify these principles. In addition to this design philosophy, PLUM is designed to eliminate stalls in instruction dispatch pipeline wherever possible by providing architectural support for static branch prediction. Static branch prediction in PLUM requires very little extra hardware because the mechanism is similar to choicepoint creation and backtracking.

2.1. Multiple Specialized Functional Units

The short pipelines of most processors allow only limited overlap of operations. Figure 2.1 illustrates how multiple function units increase the overlap of operations that require multiple cycles to execute. On an average, the Berkeley PLM instructions execute in about 7 cycles [6], but consecutive instructions are overlapped by only one cycle (microinstruction execution is also overlapped by pipelining in the microen-

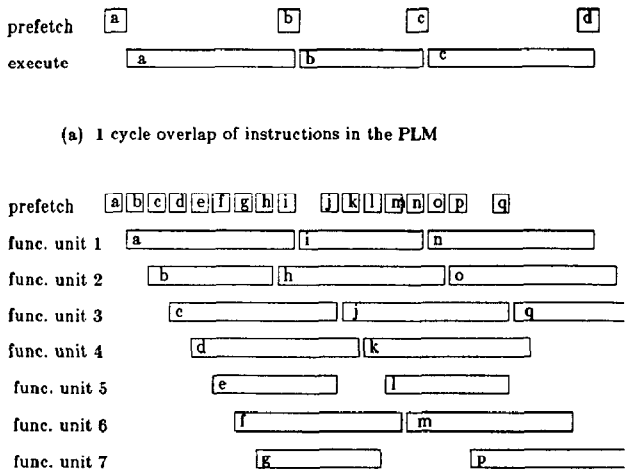

Figure 2.1: Multiple Functional Units Increase Overlap

gine). Potentially, therefore, a speedup of up to 7 could be achieved by multiple function units. However, because of stalls due to data dependencies and branches in the execution stream, we do not expect to achieve complete overlap of instructions. Each function unit in PLUM has specialized hardware that enables it to execute a particular set of tasks efficiently.

2.2. Data Driven Control

PLUM uses data driven control to resolve data dependencies between instructions. An instruction can execute when its operands are available. Instructions may execute out of order. Instruction dispatch can, therefore, continue beyond a stalled instruction. Additional parallelism is exploited because subsequent instructions may be independent of the stalled instruction and can execute on other function units.

As mentioned earlier, PLUM uses registers to pass arguments to procedures instead of passing them in a call frame. There are two reasons for this choice. First, access to registers is faster than access to memory. Second, data driven control requires some hardware to indicate whether or not an argument is valid, usually a "valid bit" for each datum. Valid bits for the entire memory are expensive and slow (since the valid bits must be reset whenever stack space is reclaimed). Valid bits for a small number of registers are easier to implement. The disadvantage of registers is that they have to be saved in memory in choicepoints and environments. With suitable buffering, and with parallel execution of choicepoint and environment instructions, copying and restoring registers from memory is overlapped with other operations and usually does not slow down execution of a program.

2.3. Partitioned Memory

Since Prolog execution is memory intensive, high performance Prolog processors must provide a high bandwidth access to memory. With multiple function units executing in parallel, PLUM's memory bandwidth requirement is even greater. In order to provide this bandwidth, each function unit has its own port to memory. A shared memory that can be accessed in parallel through multiple ports is either expensive or slow. In PLUM, the memory is partitioned so that each type of specialized function unit accesses only one partition of memory, and there is no sharing among partitions. This means that the memory can be easily implemented as multiple modules, one for each partition, that can be accessed in parallel.

3. Architecture and Implementation

3.1. Architecture Description

PLUM's architecture is similar to the WAM. Like the WAM, data types are specified by tag fields in data words. A PLUM data word is 32 bits wide. Each data word has a 4-bit type tag and 28-bit value field. The types are listed in table 3.1. List and structure data types contain pointers to lists and structures respectively. Lists consist of elements and links (which have list tags). A list ends with a word in the link position that does not dereference to a list. Although PLUM's storage model is also similar to the WAM, PLUM's memory partitioned as described section 3.1.1. PLUM's registers organization is quite different from the WAM and is described in section 3.1.2. The PLUM instruction set is described in section 3.2.

Table 3.1: Data Types in PLUM

tag (hex)	type
0	unbound variable
1	bound variable (reference)
2	list
3	structure
4	integer
5	atom
6 - E	(currently undefined)
F	nil (special constant)

3.1.1. Memory Organization

The PLUM has 3 separate address spaces (memory partitions): the choicepoint stack, the environment stack, and a global address space. The global address space contains 4 memory areas: code, heap, trail and system memory. The system memory is used by the operating system. The PLUM architecture does not specify a memory area for the *push down list* (PDL), a stack used in the unification of nested lists and structures. A PLUM implementation provides memory space for one or more PDLs either in a separate memory area or in a part of shared memory. Multiple PDLs are useful because several Unification Units could unify nested structures in parallel.

Tick's measurements [15] show that about 50 percent of all data memory accesses in Prolog programs are to the choicepoint stack and about 25 percent to the environment stack. Separate address spaces for the choicepoint and environment areas greatly reduce the memory traffic that would otherwise compete with accesses to the global memory. In our implementa-

tion, only the global memory is accessed by multiple function units. Overheads due to cache coherence will only apply to this area which accounts for only about 25 percent of data accesses.

3.1.2. Register Sets

The instruction set of the PLUM is based broadly on the WAM but registers are treated quite differently so that procedure executions are pipelined as in POPE [1]. A procedure has access to two register sets: a source or input register set, and a destination or output register set. A procedure only writes to the output register set. The output register set of one procedure becomes the input register set of the next procedure. The number of register sets is not specified by the architecture. In fact the architecture may assume a very large number of sets and the implementation must ensure that it appears that way.

Table 3.2: PLUM Registers

Name	Register
R0-R7(in,out)	Argument registers(input, output)
CP(in,out)	Continuation pointer
E(in,out)	Environment pointer
TE(in,out)	Top of Environment stack
B(in,out)	Backtrack pointer
TR(in,out)	Trail Pointer
H(in,out)	Heap pointer
L(in,out)	Alternate address
P	Program counter

Table 3.2 lists the registers in each PLUM register set. Apart from the input and output argument registers, there are registers in the input and output set with special functions. The Continuation Pointer (CP) contains the address of the next instruction to execute should the current goal succeed. The Environment Pointer (E) points to the current environment on the environment stack. The TE register points to the top of the environment stack. Note that, unlike the PLM, the PLUM has separate stacks for the environments and choicepoints. The TE register is necessary because the environment stack is not a true stack and the current environment may not be on the top of the environment stack. The backtrack Pointer (B) points to the last choicepoint on the choicepoint stack. The Trail Pointer (TR) points to the top of the trail stack. The Heap Pointer (H) points to the top of the heap. The L register contains the address of the next instruction to execute should the current goal fail. In addition to these registers, there is a Program Counter (P). Memory addresses that appear as arguments in PLUM instructions are offsets from the current value of P.

An important feature of a register set is that it acts as a buffer for the environment and choicepoint because its registers are only written once. Thus, environment and choicepoint instructions can execute after the rest of the instructions for the register set have completed.

3.2. The Instruction Set

Table 3.3 lists the PLUM instructions. They are similar to the Berkeley PLM instructions. We describe them briefly here (see [7] for more details on the PLM instruction set).

Table 3.3. PLUM Instruction Set

Indexing	Load and Save

swot Reg,Lv,Lc,Ll,Ls
swoc Reg,Hashtable
swos Reg,Hashtable
	load Reg,Y
	save Reg,Y
	asave Reg,Y

Get	Put
getval(type) R1,R2 getconst(type) R,C getlist(type) R,L getstruct(type) R,S	putval R1,R2 putconst R,C putlist R,L putstruct R,S putvar R
Procedure Control	Clause Control
tryelse T,L retryelse T,L predictelse T,L trust T fail cut cutd nocp	proceed execute P dexecute P calls P acalls P allocate N
Miscellaneous (incomplete list)	

add R1, R2, Rd
inc R1, Rd
mul R1, R2, Rd
cgtz R1
cgtr R1,R2
	sub R1, R2, Rd
	dec R1, Rd
	deref R1, R2
	clsz R1
	ceql R1, R2

3.2.1. Indexing, Clause and Procedure Control Instructions

The indexing instructions are used to filter the set of candidate clauses based on the type and value of input argument registers. The procedure control instructions create and manipulate choicepoints. The *predictelse* instruction is used for static branch prediction (to select one of several clauses to try). The branch destination is checked during subsequent head unification of the clause. Whenever possible, a compiler should use static branch prediction instead of indexing instructions, since the indexing instructions cause instruction dispatch to stall (unless the implementation also supports dynamic branch prediction of indexing instructions). The *nocp* (no choicepoint operation) instruction is used if there is no choicepoint or prediction instruction to load the output B register. The clause control instructions deal with environment allocation and deallocation, and control transfer associated with procedure calls and returns. The *acalls* instruction is similar to the *calls* (procedure call) instruction except that it does not transfer the input E and TE registers to the output set like the *calls* instruction. It is used if there is an *allocate* instruction preceding it in the current set which loads new values of the E and TE registers into the output set. The *dexecute* instruction is similar to the *execute* instruction except that it also deallocates the current environment.

3.2.2. Get and Put Instructions

The *get* instructions unify arguments of the clause

head with the arguments of the goal (available in the input argument registers). The *put* instructions are used to load arguments of a goal or procedure. The *get* instructions have two "type" attributes. The "shared" attribute implies that the unification must get exclusive access to every variable that it binds because that variable could be shared with another unification. Static analysis of programs, as proposed by Citrin [4], can be used to determine which unifications could potentially share unbound variables with other unifications, and only these unifications need incur the overhead of synchronization before binding variables. The "check" attribute requires that the instruction first check that the type of the input argument is appropriate. The "check" attribute is used to check if a predicted branch destination is correct.

3.2.3. Get and Put for Lists and Structures

Unlike the WAM and the PLM, the *getlist, putlist, getstruct* and *putstruct* instructions are not followed by *unify* instructions. Instead, each list and structure unification is treated as a single instruction. Each instruction contains a pointer to a list of words in code space that describe elements of the list or structure. These list and structure descriptions are different from other instructions in that they are not dispatched to Unification Units, but rather they are fetched directly from memory by the Unification Unit that executes the list or structure get or put instruction.

3.2.4. Load and Save Instructions

Unlike the Berkeley PLM, the get and put instructions cannot have permanent variables in the environment as arguments. This allows the environment memory area to be treated as a separate address space inaccessible to the Unification Units. The load instruction loads an argument register with a permanent variable from the environment, and the save instruction saves an argument register in the environment as a permanent variable. The *asave* instruction is similar to the *save* instruction except that it is used if an environment has been allocated in the current set (in which case the input E register does not point to the current environment, but the input TE register does).

3.2.5. Miscellaneous Instructions

The miscellaneous instructions include arithmetic and logic operations, as well as simple general purpose instructions that could be used to implement builtin operations of Prolog. Instructions such as *cgtz* (which succeeds if the argument is greater than zero and fails otherwise) can also be used to check that a particular clause was correctly predicted.

3.3. Implementation

3.3.1. Overview

Figure 3.1 is an overview of a PLUM implementation. A Prefetch Unit fetches, buffers and dispatches instructions to appropriate functional units. The Choicepoint Unit and Environment Unit access and manipulate the choicepoint and environment stacks respectively. Several Unification Units execute

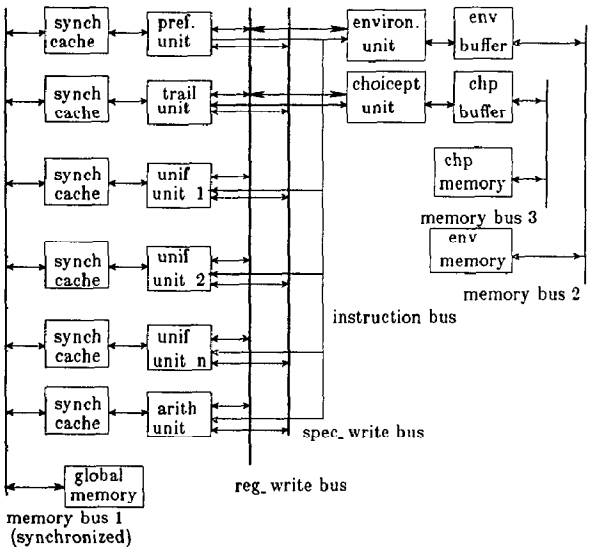

Figure 3.1: Overview of the PLUM microarchitecture.

unification instructions as well as some simple arithmetic instructions. An Arithmetic Unit performs more complicated arithmetic operations (such as floating point instructions). A Trail Unit trails variables bound by the Unification Units and performs the detrail operation during backtracking.

3.3.2. Multiple Register Sets

The PLUM architecture assigns a new register set for the outputs of instructions each time a procedure boundary is crossed. A practical implementation can provide only a limited number of register sets. The microarchitecture described in this section provides a few (we think 4 are sufficient) register sets, simulating a large number of register sets by re-using them. The register set numbers "wrap around", and each register set is reset so that all its registers are marked invalid before it is re-used. The Prefetch Unit appends the physical set number to each instruction that it dispatches to functional units.

3.3.3. Data Driven Control

Each register has a valid bit associated with it, and the valid bits are used to implement the data driven control. An instruction can only execute when its input operands are valid. Some instructions need to wait for implicit operands (that are not specified explicitly in the instruction). For example, a unification cannot allocate space on the heap until all the heap space for the previous goal has been allocated in order to prevent interleaving of data for different goals on the heap. The Unification Unit must wait until the input heap register for its set is valid, and the input heap register is loaded by the previous goal when it requires no more heap space.

Each functional unit contains a shadow copy of all the register sets (we refer to all the register sets together as the register file). The microarchitecture maintains consistent copies of the register file in all the functional units by insisting that registers in the file can only be written over shared buses. The PLUM

has two buses: the register_write bus is used to write the argument registers, and the special_write bus is used to write the other registers. The functional units arbitrate for the use of a bus one cycle before they use the bus.

4. Simulation Results and Analysis

We have written a register transfer level simulator for PLUM in order to estimate the performance of the implementation and evaluate various design choices. The measurements described below demonstrate that PLUM achieves an average speedup of 3.4 over the Berkeley VLSI-PLM, a specialized pipelined processor for Prolog.

4.1. Assumptions and Benchmarks

The PLUM simulator accepts the access times of each memory port and the number of Unification Units as inputs. The shared memory (connected to the ports of the Prefetch Unit, Unification Units, and Trail Unit) is treated as a multi-port memory for purposes of simulation. Since a multi-port memory is expensive, an actual implementation would use one of several memory systems, depending on the desired cost and performance, to allow parallel access to a shared memory. For example, one option is to use caches at each port connected to a shared bus. A cache coherence protocol can be used to ensure that shared data are kept consistent. Another option is to have multiple memory modules connected to the processors by an interconnection network such as a cross-bar switch. These and other options have lower performance than a multi-port memory with the same access time, but we believe that the performance degradation is small. Memory traces from the simulator can aid in evaluating performance degradation with various memory system designs, but that is beyond the scope of this paper.

The simulator models a branch target instruction buffer (4 lines, 16 words per line) and a 16 word prefetch buffer in the Prefetch Unit. The Trail Unit contains an 8 word trail buffer and each Unification Unit contains an 8 word prefetch buffer to hold elements of lists and structure unifications from the code space. They have been included in the simulator so that the performance measurements are not degraded by factors that can easily be eliminated by small and simple buffers that are common in modern VLSI processors. At the same time, the simulation models the performance degradation that can be expected due to misses in buffers. Since our simulations are run with cold starts (the buffers are initially empty), the performance of a program with a short execution time is usually degraded more than that of a longer program.

We present measurements on four benchmarks that have commonly been used in comparing the performance of Prolog systems. *Concat* is a small program that concatenates a list of 3 elements to a list of 6 elements. *Hanoi* computes the solution to the "towers of Hanoi" puzzle for 8 disks , *nrev1* reverses a list of 30 elements, and *qs4* sorts a list of 50 integers using the quicksort algorithm.

4.2. Effect of Multiple Unification Units

In figure 4.1 we plot PLUM's performance (relative to the performance of PLUM with 1 Unification Unit and 1 cycle memory access) for each benchmark, and figure 4.2 is a similar graph for the average of all the benchmarks. Performance is measured as the reciprocal of the number of execution cycles, and the number of cycles for the average is the sum of the cycles for each benchmark. Figures 4.1 and 4.2 show that PLUM's performance improves with additional Unification Units, but the performance improvement is small beyond three Unification Units for the benchmark programs chosen. Programs with more unification parallelism can be expected to benefit more from multiple Unification Units. Such programs usually have a large number of complex argument unifications in each goal.

4.3. Effect of Memory Access Time

Figure 4.3 shows how PLUM's performance (relative to the performance of PLUM with 1 Unification Unit and 1 cycle memory access time) on each benchmark is affected by memory access time. In these measurements we assume that the memory access time on all memory ports is the same. Figure 4.4 shows the effect on the average performance. The figures show that PLUM's performance degrades slowly with increasing memory access time. This suggests that a PLUM implementation will perform quite well even if the memory system's effective memory access time is more than 1 cycle (for example, due to cache misses and synchronization for shared data).

The *hanoi* benchmark behaves differently from the other benchmarks. Unlike the other benchmarks, unification is not the bottleneck to *hanoi*'s performance, and none of the Unification Unit instructions in the benchmark access memory. As memory access time increases, therefore, the performance is determined almost completely by the Choicepoint, Environment and Prefetch Units, and the curves for various numbers of Unification Units merge.

4.4. Comparison with the Berkeley VLSI-PLM

Table 4.1 compares the performance of the VLSI-PLM (using a simulator that assumes a 1 cycle memory access and 100nsec clock cycle) with that of PLUM (using a simulator with 3 Unification Units, 1 cycle memory access and 100nsec clock cycle). On some benchmarks, PLUM's speedup over the VLSI-PLM cannot be attributed only to fine grain parallelism. The VLSI-PLM has little support for arithmetic operations and comparisons. For example, the VLSI-PLM's performance on the *hanoi* benchmark can be improved by approximately 0.7 millisec (13.4 percent) by an improved instruction set for arithmetic. Table 4.1 shows that PLUM achieves a speedup of 3.42 over the VLSI-PLM averaged over the benchmarks chosen.

Table 4.1. Comparison of VLSI-PLM and PLUM

	Execution Time (millisec)		Speedup
Benchmark	VLSI-PLM	PLUM	
concat	0.035	0.015	2.33
hanoi	5.211	1.331	3.91
nrev1	2.116	0.788	2.68
qs4	4.304	1.28	3.36
average	11.67	3.414	3.42

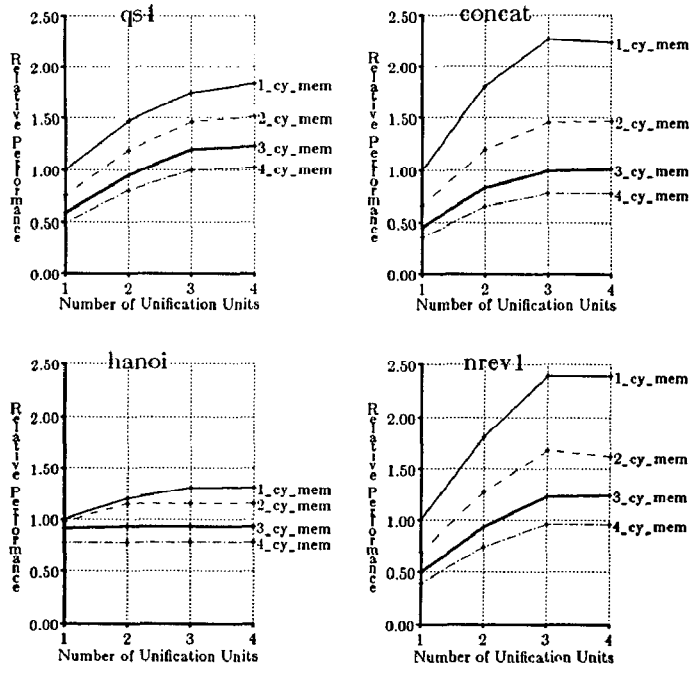

Fig. 4.1: Effect of Multiple Unification Units on Performance

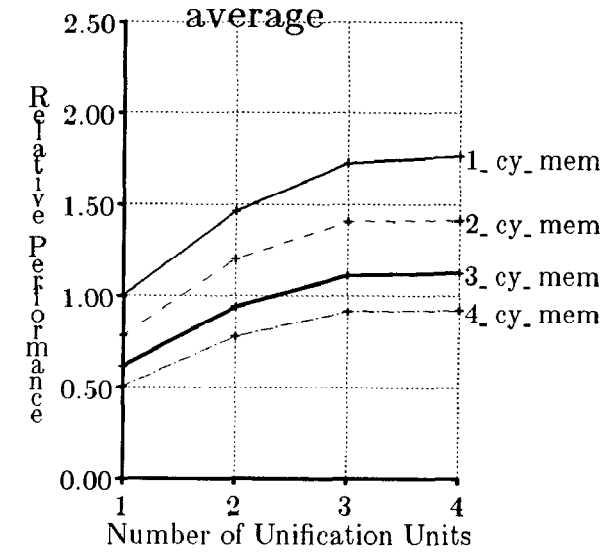

Fig. 4.2: Effect of Multiple Unification Units on Average Performance

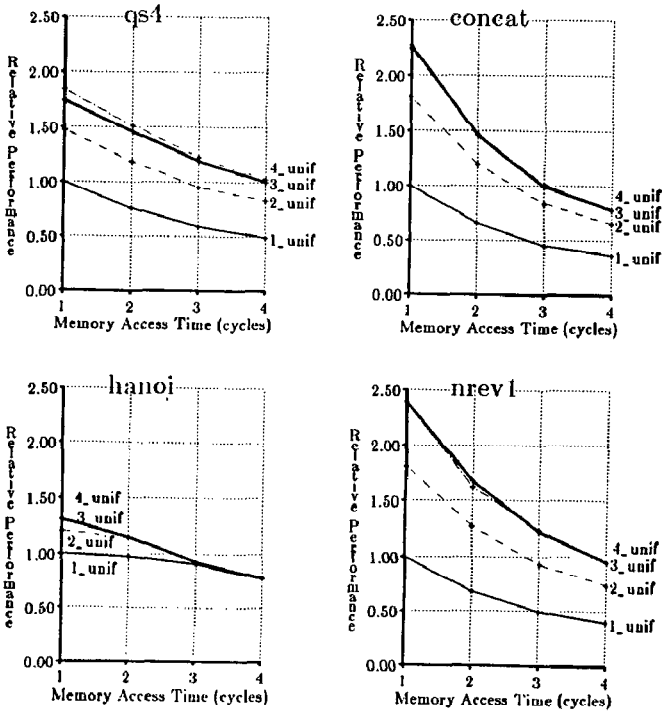

Fig.4.3 Effect of Memory Access Time on Performance

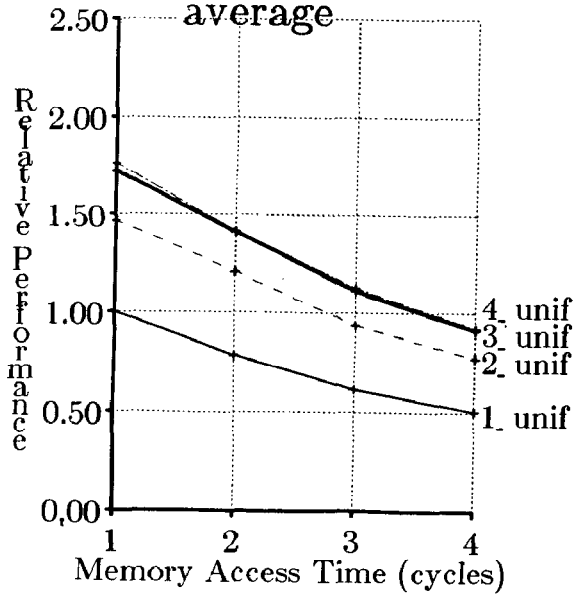

Fig. 4.4: Effect of Memory Access Time on Average Performance

Although we feel that execution times are more appropriate measures of Prolog processor performance, it has unfortunately become customary to report the LIPS (Logical Inferences Per Second) that a processor can achieve. Table 4.2 lists the KLIPS (Kilo LIPS) that PLUM with 3 Unification Units achieves on the benchmark set using different clock cycle lengths and memory access times. With current CMOS technology and memory speeds, an implementation of PLUM with 50 nsec clock cycle and 1 cycle memory access is feasible. With these assumptions, PLUM achieves 1102 KLIPS.

Table 4.2. KLIPS Rates for PLUM (with 3 Unif. Units)

Clock Cycle	100nsec		50nsec	
Mem. Access (cycles)	2	1	2	1
Benchmark				
concat	302	470	604	940
hanoi	508	576	1016	1152
nrev1	444	631	888	1262
qs4	399	477	798	954
average	449	551	898	1102

5. Summary and Conclusions

We have described the architecture and implementation of PLUM, a high performance processor for Prolog that exploits fine grain parallelism by executing instructions in parallel on multiple specialized function units, each of which can be implemented on a single VLSI chip. PLUM achieves a speedup of approximately 3.4 over the Berkeley VLSI-PLM averaged over a set of benchmarks. Performance of PLUM improves with additional Unification Units but the performance improvement is small beyond three Unification Units for the benchmarks chosen. The amount of parallelism that can be exploited by multiple Unification Units varies from program to program. However, even programs without sufficient unification parallelism perform well because of parallel execution of choicepoint and environment instructions. PLUM's performance degrades slowly with increasing memory access time, indicating that it will perform well with a wide range of memory systems.

We are currently optimizing the microcode for each function unit and running simulations on a larger and more diverse benchmark set. We are confident that simulations on the larger benchmarks will result in comparable or greater speedup than that observed in this paper.

Acknowledgement

Sponsored by Defense Advanced Research Projects Agency under Contract Nos. N0039-84-C-0089 and N00014-88-K-0579

The authors also wish to thank Zycad Corporation for the use of their Endot N.2 hardware simulation tools that greatly simplified the task of simulating PLUM.

References

1. J. Beer, Concepts, Design, and Performance Analysis of a Parallel Prolog Machine, *PhD thesis, Technical University*, Berlin, .
2. 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.
3. C. Chen, A. Singhal and Y. N. Patt, PUP: An Architecture to Exploit Parallel Unification in Prolog, *(submitted for publication)*, 1988.
4. W. Citrin, Parallel Unification Scheduling in Prolog, *PhD thesis, University of California, Berkeley*, Berkeley, California, 1988.
5. 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.
6. T. Dobry, A High Performance Architecture for Prolog, *PhD thesis, University of California, Berkeley*, Berkeley, California, 1987.
7. B. Fagin and T. Dobry, The Berkeley PLM Intruction Set: An Instruction Set for Prolog, *Report No. UCB/Computer Science Dpt. 86/257, Computer Science Division, University of California, Berkeley*, September 1985.
8. 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.
9. 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.
10. M. V. Hermenegildo, An Abstract Machine for the Restricted AND-Parallelism of Logic Programs, *Third International Conference on Logic Programming*, July, 1986, 25-39.
11. 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.
12. 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.
13. 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.
14. E. Tick and D. Warren, Towards a Pipelined Prolog Processor, *1984 International Symposium on Logic Programming*, February 1984.
15. 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.
16. R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development 11* (1967).
17. D. H. D. Warren, An Abstract Prolog Instruction Set, *Technical Report 309, Artificial Intelligence Center, SRI International*, 1983.