# The Implementation of Prolog via VAX 8600 Microcode

*Jeff Gee, Stephen W. Melvin, Yale N. Patt*

Computer Science Division
University of California
Berkeley, CA 94720

*ABSTRACT*

We have implemented a high performance Prolog engine by directly executing in microcode the constructs of Warren's Abstract Machine. The implemention vehicle is the VAX 8600 computer. The VAX 8600 is a general purpose processor containing 8K words of writable control store. In our system, each of the Warren Abstract Machine instructions is implemented as a VAX 8600 machine level instruction. Other Prolog built-ins are either implemented directly in microcode or executed by the general VAX instruction set. Initial results indicate that our system is the fastest implementation of Prolog on a commercially available general purpose processor.

## 1. Introduction

Various models of execution have been investigated to attain the high performance execution of Prolog programs. Usually, this involves compiling the Prolog program first into an intermediate form referred to as the Warren Abstract Machine (WAM) instruction set [1]. Execution of WAM instructions often follow one of two methods: they are executed directly by a special purpose processor, or they are software emulated via the machine language of a general purpose computer.

In the case of software emulation by a general purpose machine, three levels of translation are required. Prolog code must first be transformed to WAM code, then translated to host machine code, and finally interpreted by host microcode. We can enhance performance by bypassing one or more of these levels.

The approach taken in this paper is to eliminate the host machine code level by adding microcode which interprets directly the Warren instruction set. We chose the VAX 8600 general purpose processor as the implementation vehicle.

This paper is divided into six sections. Section 2 describes the hardware and software elements of our system. Section 3 describes the Warren Abstract Machine. Section 4 describes our implementation of the Warren machine in the VAX 8600. Section 5 contains our performance results and compares these measurements with those of alternative schemes. Section 6 offers some concluding remarks.

## 2. Operating Environment

### 2.1 8600 System Architecture

We have implemented the Warren Abstract Machine on a VAX 8600 computer operating under 4.3 BSD UNIX. The VAX 8600 is a 32 bit computer designed with ECL macrocell arrays. Figure 1 shows a simplified block diagram of the 8600. The cycle time of the 8600 is 80 nanoseconds.

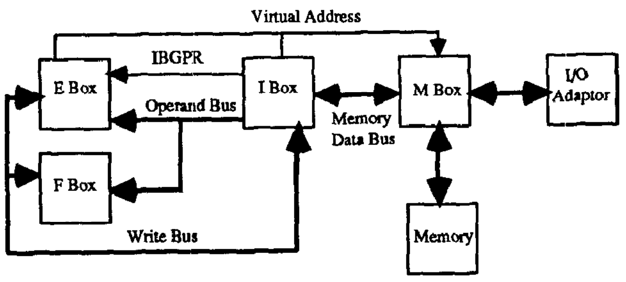

Figure 1. Simplified Block Diagram of the VAX 8600

The 8600 consists of six subprocessors: the EBOX, IBOX, FBOX, MBOX, Console, and I/O adapter. Each of the separate boxes perform fairly independent operations, thus we need not modify all 8600 subsystems for our Prolog implementation. The EBOX executes the VAX instruction set and generally supervises the entire sytsem. The IBOX is responsible for prefetching instructions and operands. Because the EBOX and IBOX operate simultaneously, the IBOX can decode and fetch operands for instructions before the previous instruction completes execution in the EBOX. The FBOX is a floating point accelerator, containing special hardware to achieve a high performance computing capability. The MBOX performs memory accesses requested by the IBOX and EBOX.

Sixteen general purpose registers are available to the programmer. Four copies of these registers are maintained to guarantee fast and flexible access to the data. Any modification updates, by means of special hardware, all copies of the registers.

Buses interconnect the various boxes. All memory and I/O accesses occur via the Memory Data Bus which connects the MBOX to the IBOX. Memory operands are passed from the IBOX to the EBOX across the Operand Bus. Operands in the general purpose registers are represented as GPR numbers passed across the IBGPR bus. Results from the EBOX or FBOX destined for memory are returned to the IBOX via the Write Bus. Any modifications to the general purpose registers are also broadcast across the Write Bus to update all other copies. The IBOX passes memory results to the MBOX via the

Memory Data Bus. The EBOX and IBOX supply virtual 32 bit addresses to the MBOX.

## 2.2 8600 Microarchitecture

All of the boxes are microprogrammed independently. Most of the microcode, including all instruction specific microcode, is contained in the EBOX. To implement the Warren instruction set, we modified only the EBOX microcode and the DRAM entries described in sections 2.3 and 4.5. Our additional microcode performs the operations required for each of the WAM constructs and for several of the Prolog built in functions which are normally represented as Warren escape sequences. The IBOX and MBOX perform the duties of instruction prefetching, operand prefetching, and memory accesses. No IBOX modifications were necessary other than the DRAM entries which are needed by the EBOX, since we use normal VAX addressing modes and the extended VAX opcodes (FD xy) to represent a WAM program.

The EBOX contains 8K x 92 bits of writable control store. The horizontal microinstruction format facilitates the implementation of a simple, but flexible data path. This flexibility accounts for much of the power of this machine.

The EBOX data path consists of a dual-ported 256 x 32 bit scratchpad register file, an ALU, and a barrel shift network. The scratchpad contains internal processor registers, temporary registers, constants, and architecturally defined general purpose registers.

The 8600 microcycle is 80 nanoseconds. In one microcycle, the machine can perform an ALU or shifter operation on two scratchpad elements and store the result back in the scratchpad. The barrel shifter works in parallel with the ALU and can select any 32 consecutive bits from a 64 bit value. Two scratchpad registers or one register concatenated with a memory operand supply this 64 bit value.

The MBOX contains a 16 Kbyte data cache to speed up memory accesses. A memory read takes two microcycles if the data is found in the cache, and seven cycles in the event of a cache miss.

## 2.3 Microcode Flow

Both EBOX and IBOX microcode share in the execution of an instruction. The IBOX microcode prefetches and decodes the instruction. Decoding is facilitated by a decode RAM containing opcode specific information. Based on the decode information, a microcode fork address is passed from the IBOX to the EBOX, directing the EBOX to the entry point of a microsequence which performs the execution phase of the particular instruction. If operands are required, the IBOX delivers a fork address causing the EBOX to loop until the MBOX can fetch the operands from memory. As the operands become available, the IBOX directs the EBOX to microcode which operates on the data. After the EBOX completes the execution phase, it may transfer results back to the IBOX. The IBOX delivers any memory result to the MBOX. Otherwise, the EBOX has already stored the result into a general purpose register, with all copies of the general registers updated via the Write Bus.

The fork addresses sent to the EBOX by the IBOX are stored in the decode RAM. The RAM contains up to eight fork entries for each instruction. As a result of each fork, the EBOX reads data, operates on data, or both. An optimization occurs when one operand is in memory and another is in a general purpose register. The IBOX delivers the memory data on the Operand Bus and the register number on the IBGPR bus.

The EBOX fork addresses are operand specific. The particular microsequence which is executed depends on the location of the data, its data type, and whether or not it is a special case handled by optimization microcode. The IBOX decodes the operand specifiers and generates the EBOX fork addresses appropriately.

## 2.4 Compilation and Assembly of Prolog Programs

Three levels of translation are required to transform Prolog programs into an executable VAX 8600 object file. First, the Prolog program is compiled into its equivalent Warren Abstract Machine form. An intermediate assembler then takes the output of the compiler and generates a VAX assembly language file. This file is then assembled into a VAX executable file.

The Prolog compiler was developed at Berkeley as a Master's Thesis by Peter Van Roy [3]. The compiler is written in Prolog, and is invoked from a Cprolog intepreter running under 4.3 BSD Unix. Input to the compiler is a set of Prolog clauses and a query. The output is the equivalent translation into the Warren instruction set.

The intermediate assembler transforms the code generated by the compiler into a VAX assembly language file. It is written in C, and performs a one to one translation of Warren code to VAX code. Each WAM instruction corresponds to a single VAX instruction. An extended VAX opcode is defined to represent each of the Warren constructs. The operands of the Warren instructions are represented as normal VAX operand specifiers. The intermediate assembler is responsible for parsing the WAM file and generating the appropriate VAX code. The assembler also creates symbol and string tables which represent Prolog atoms, lists, and structures.

The VAX/UNIX assembler **as** generates executable VAX object code from the output of our WAM assembler.

In addition, our implementation supports several built-in Prolog functions which are represented as escape sequences in the Warren Abstract Machine. These include the output functions write(X) and nl, the **is** function, the int(X) function, and the binary relational operators ==, =<, >=, <, and >.

The escape functions are implemented either directly in microcode or via the standard VAX instruction set. The relational operators and the int(X) function are implemented in microcode, each corresponding to a newly defined VAX instruction. Write(X) and nl make use of the standard C library. These escapes are represented as VAX subroutine calls to C routines compiled to the VAX instruction set. These routines are compiled separately from the Warren code, and the two files are later linked into a single VAX 8600 executable image.

The **is** escape function is more complicated. The arithmetic expression to be evaluated may require multiplication or division, and may contain nested expressions. Currently we evaluate non-nested expressions requiring only addition or subtraction directly in microcode. More complicated expressions are evaluated by C routines.

The entire compilation and assembly process is shown in figure 2.

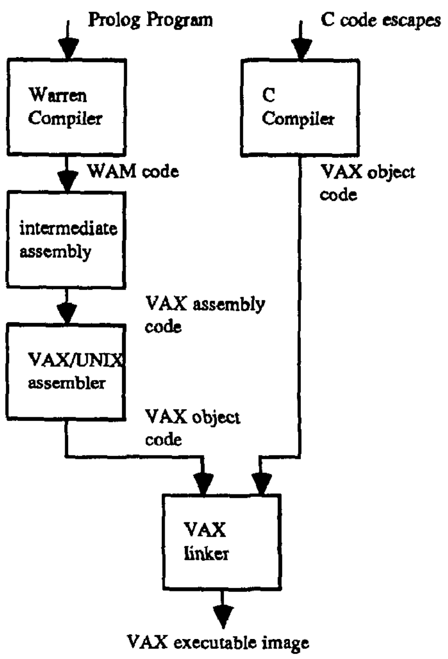

Figure 2. Prolog Compilation Process

## 3. Details of the Warren Abstract Machine

### 3.1 Data Types

Prolog manipulates four types of data: constants, variables, lists, and structures. The data type is determined by a tag field in the uppermost two bits of a 32 bit data word.

Constants can be integers, atoms, floating point values, and the special constant NIL. Integers are stored in the data word itself, atoms are represented as pointers to the item in the symbol table, and floating point values are represented as pointers to the value on the heap.

Variables can be bound or unbound. The contents of a variable point to the element to which it is bound. Unbound variables contain pointers to themselves.

Lists are represented as a data word containing a pointer to the first element of the list. Lists are cdr-coded. The car of the list is the first element; the cdr points to the remainder of the list. To improve memory efficiency, the cdr cell is not included if the rest of the list directly follows the car in memory. Otherwise, the cdr cell directly follows the car. A cdr bit in the data word detects this condition. If the cell following the car has its cdr bit set, it points to the rest of the list. Otherwise, it is the first element of the remainder of the list.

Structures are lists with principal functors. The first element of the list is the principal functor of the structure.

### 3.2 Registers

The Warren Abstract Machine architecture contains 18 special purpose registers:

| | |
|---|---|
| A1 - A8: | the Argument registers, containing the arguments of a Prolog goal. |
| P: | the Program counter, addressing the next instruction to execute. |
| CP: | the Continuation pointer, where execution continues should the current goal succeed. |
| E: | the Environment pointer, references the last environment pushed onto the stack. |
| B: | the Backtrack pointer, contains the address of the last choice point placed on the stack. |
| TR: | the Trail pointer, pointing to the top of the trail. |
| H: | the Heap pointer, pointing to the top of the heap. |
| HB: | the Heap Backtrack pointer, the value of the heap pointer when the last choice point was placed on the stack. |
| S: | the Structure pointer, pointing to the current element of a list or structure being unified. |
| PDL: | the Push Down List pointer, pointing to the last element placed on the push down list. |
| N: | the number of permanent variables in the current environment. |

### 3.3 Data Memory Allocation

The data memory is partitioned into four spaces: the Stack, Heap, Trail, and Push Down List (PDL).

The stack is used to store control information necessary for the correct execution of a Prolog program. Choice points and environments are placed on the stack by special instructions which save data needed for backtracking.

An environment contains the saved state of a Prolog clause. It contains register values and "permanent variables" which must be retained between goals in a multi-goal clause. Permanent variables are variables whose use is not restricted to the first goal in a clause. Thus, if kept in argument registers, these variables may be overwritten during execution of a subsequent clause goal. These variables are stored on the stack and retrieved when the appropriate goal is invoked. In addition, an enviroment contains the CP, E, N, and B registers which are necessary to continue computation when the last goal in a clause succeeds.

A choice point contains the information necessary to restore the process state when a goal fails. Choice points are placed on the stack whenever a procedure contains more than one clause which can unify with the current goal. Choice points contain the following register values:

| | |
|---|---|
| An: | the contents of the argument registers |
| E: | the location of the last environment |
| CP : | address to continue if the current goal succeeds |

| | |
|---|---|
| B: | location of previous choice point |
| TR: | value of the trail pointer |
| H: | top of the heap |
| N: | number of permanent variables in the current environment |
| L: | address to continue should the current goal fail |

The heap is used to store lists and structures. These data items are difficult to store on the stack. Instead, pointers to the lists and structures are stored on the stack. In addition, the heap is used to globalize variables on the stack which may become dangling references when an environment is deallocated [4].

The trail is used to store addresses of bindings which must be undone upon goal failure. When the current goal fails, the trail value saved in the current choice point is retrieved. All addresses in trail locations between this saved value and the current trail pointer are reset to unbound variables.

Finally, the PDL is a small stack used to unify nested structures and lists. Dangling references occur when unifying nested lists. During the depth first traversal of a nested list, we lose pointers to the remainder of the higher levels of the list. This occurs if the address of the cdr is not saved when a nested list is encountered. The Push Down List contains pointers to the remainder of a nested list. Unification resumes at the topmost PDL location during a depth first traversal. When the PDL is empty, the list has been traversed.

### 3.4 Instruction Set

The Warren Constructs are usually grouped into the following six classes. Details on the functions of each instruction are available in [1,4].

| Indexing | Procedure Control |
|---|---|
| switch_on_constant | try |
| switch_on_term | retry |
| switch_on_structure | trust |
| | try_me_else |
| | retry_me_else |
| | trust_me_else |
| | fail |
| | cut |
| | cutd |

| Clause Control | Get |
|---|---|
| proceed | get_variable |
| execute | get_value |
| call | get_constant |
| escape | get_nil |
| allocate | get_structure |
| deallocate | get_list |

| Put | Unify |
|---|---|
| put_variable | unify_void |
| put_value | unify_value |
| put_unsafe_value | unify_variable |
| put_constant | unify_constant |
| put_structure | unify_cdr |
| put_list | unify_nil |

The available escape sequences include:

| | |
|---|---|
| escape write | escape nl |
| escape integer | escape > |
| escape < | escape == |
| escape >= | escape =< |
| escape is | |

## 4. Implementation of the Warren Architecture on the 8600

### 4.1 Data Tags

Our method for implementing data tags is shown in figure 3. The two high order bits of a 32 bit data word specify the type of the data. The third bit supports the cdr-coding of lists. Another

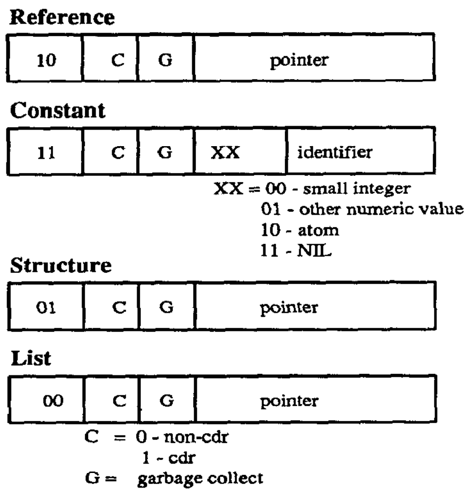

Figure 3. Data Types

bit is allocated for garbage collection, which is not yet implemented in our current system. (Our plans are, however, to handle garbage collection by means of a separate sequence of microinstructions.) Constants require two secondary tag bits which determine the type of constant.

### 4.2 Register Allocation

The architectural registers of the Warren Abstract Machine are mapped onto the sixteen VAX general purpose registers and two of the four VAX stack pointers. The argument registers, program counter, heap pointer, environment pointer, and backtrack pointer each were assigned a VAX general purpose register. VAX general purpose registers were also allocated for the CP, N, and S registers. The last available GPR is shared among the TR and PDL registers, since 16 bits of address space is sufficient for both the trail and PDL. The HB register is stored in the Executive Mode Stack Pointer. The Supervisor Stack Pointer is used for collecting performance data of the executing Prolog program; i.e., it contains the number of logical inferences made. A logical inference is defined as the total number of calls, executes, and escapes executed.

Several WAM instructions perform different functions depending on the state of two mode bits. The **cut** bit determines the proper number of choice points to discard when the Prolog cut (!) operator is executed. Normally, all choice points above the B register value saved in the current environment are discarded. However, if the current procedure has placed a choice point on the stack, then one more choice point must be discarded. The cut bit is set when a choice point is placed on the stack and cleared by a call, execute, or proceed instruction. The **read/write** bit determines the mode for the unify instructions. In write mode, a list or structure is unified with an unbound variable, and a copy of the data is written on the heap. In read mode, two lists or structures are unified, and their elements on the heap are compared. The mode bit is set to read when the argument dereferences to a list or structure, and is set to write if the argument dereferences to a variable.

The read/write and cut bits are stored directly in the VAX Processor Status Longword (PSL). The PSL negative flag implements the read/write bit, and the the PSL overflow flag implements the cut bit.

Our register allocation scheme is shown in figure 4.

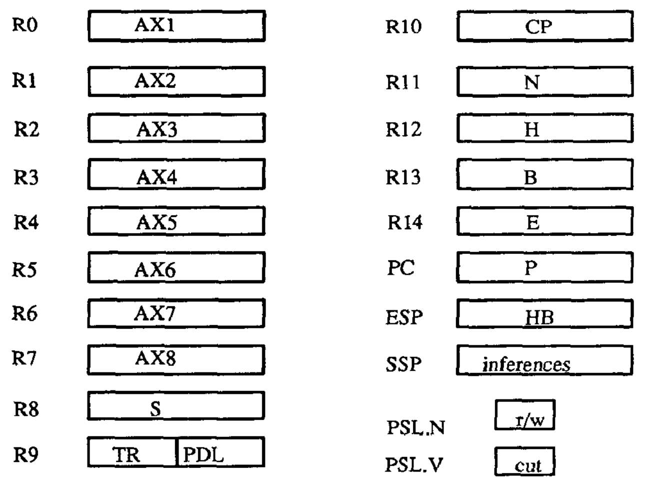

Figure 4. VAX 8600 Register Assignments

## 4.3 Memory Allocation

The VAX 8600 has 31 bits of process address space. Our Prolog implementation requires only 28 bits, due to the four bit tag in the data word. The virtual address space is allocated according to figure 5.

The code space corresponds to the size of the individual Prolog program. The heap space begins where the code space ends and grows towards high memory. The stack grows in the opposite direction. No space is allocated for the Push Down List. Instead, the PDL is stored on top of the stack, as no choice points or envirommnents will be placed on the stack while unifying two lists.

Two memory locations (7FFF 0004 and 7FFF 0008) are reserved for UNIX control data. A Prolog program is invoked by a C function call to procedure **main**, which initializes the heap pointer, saves the current frame pointer in location 7FFF 0008, and jumps to procedure **doplm**. **Doplm** stores the return address to **main** in location 7FFF 0004 and executes the Prolog program. After execution completes, **doplm** returns the result of the query to **main**. Main prints the result, restores the frame pointer from the reserved location, and returns to the calling C program.

## 4.4 Process Control

It is intended that our Prolog system will execute within a multiprogramming environment. Thus the entire Prolog process

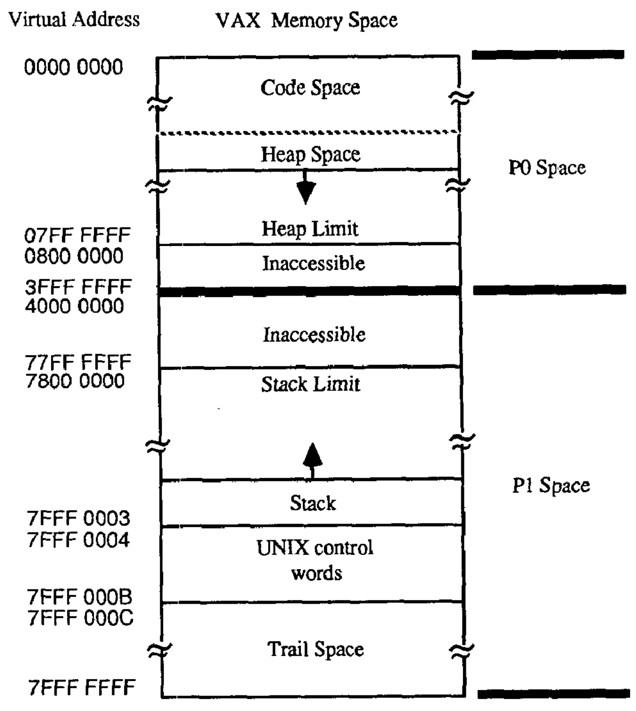

Figure 5. Memory Allocation

state is stored in the sixteen VAX general purpose registers and two of the four VAX stack pointers, all of which are saved in the process control block. Note: we were able to use the Executive Mode and Supervisor Mode stack pointers as if they were additional general purpose registers because the UNIX operating system does not require the two corresponding levels of privilege. However,. the VMS operating system has four privilege modes and requires the use of all four stack pointers. Slight changes in register allocation will be necessary to port our implementation to VAX/VMS. Possible candidates for these two registers are as follows: the logical inference count in the Supervisor Stack Pointer exists only for statistical purposes, and may be omitted. Values in the N register stored in r11 cannot exceed 256, and these 8 bits can be stored in the Processor Status Longword. The contents of the HB register in the Executive Stack Pointer can then be stored in general register r11.

Interrupts are handled between instruction boundaries. All process information is safely stored in the process control block when interrupts are executed. Many of the WAM instructions execute in non-determinate time due to the usage of the dereference, unify, bind, fail, and trail routines. When the machine unifies long lists or traces through long dereference chains, any interrupt must wait for the operations to complete, which may cause unacceptable latency for certain real-time applications. Eventually, our implementation will check for interrupts within an instruction using the VAX first part done mechanism. The process state will be preserved and execution will resume after the interrupt is processed.

Machine exceptions, such as page faults, are processed immediately. The instruction is restarted after the exception is processed. The processor registers are restored to their values before the instruction began execution. However, modifications to memory are not backed up. The microcode is designed to insure that multiple writes are atomic or to order the writes such that if a fault occurs before the instruction completes, it can restart without error.

### 4.5 Microcoding the 8600

Each of the Warren constructs plus many of the escape sequences are implemented as VAX instructions. We use extended VAX opcodes to represent each construct, with its associated microcode resident along with the host microcode. The decode RAM has been modified to provide correct fork address generation when the IBOX encounters one of the newly defined opcodes.

The operands of Warren instructions can be partitioned into four types: argument registers (Xi), permanent variables (Yi), labels (L), and constant literals (N). Operands are encoded using VAX addressing modes and conveniently evaluated by the IBOX. Argument registers are specified with register mode; permanent variables in the current environment are specified via displacement mode from the current environment pointer; labels and constants form 32 bit literals in the instruction stream.

Two instructions were added to the instruction set to facilitate execution of certain escape **is** sequences. **Plus** and **minus** are three operand instructions which dereference the first two operands, perform addition or subtraction, and store the result in the third operand. The get_value instruction completes the escape **is** sequence by unifying this result with the desired variable. Currently these instructions are hand-coded into the output of the Warren compiler. This produces faster execution times since the structure form of the arithmetic expression need not be created before the expression is evaluated. Plus and minus work only when arithmetic expressions are not nested. We plan to modify the compiler to generate these instructions automatically.

Several basic Prolog functions used by many of the Warren Constructs are also implemented in microcode. These include the dereference, decdr, unify, and fail routines. Only the fail routine is directly accessible to the user to initiate backtracking. The dereference routine follows a chain of variables until a structure, list, constant, or unbound variable is encountered. The decdr routine supports the cdr-coding of lists, and insures that a list is traversed correctly.

Over 500 lines of microcode were added to the VAX 8600 to implement the Warren Abstract Machine.

## 5. Performance Measurements

We have measured the performance of our initial implementation on two sets of benchmarks. Although our initial results indicate that our system is the fastest Prolog implementation on a commercially available general purpose computer, much remains to be done. The results reported here reflect the performance of a first-pass, unoptimized system. We believe that signficant performance gains will be achieved when the microcode is optimized, and when the remaining escape **is** sequences are moved into microcode.

Table 1 summarizes our results on the Warren Benchmark Set with those of three other systems: a Warren implementation on the NCR/32 general purpose processor; the Berkeley PLM, a special purpose processor which directly interprets the Warren instruction set; and Warren's compiled Prolog, software emulating the WAM instruction set on a DEC-2060 computer. We have not normalized the results to the cycle time of each machine.

Table 2 summarizes our results on the Berkeley Benchmark Set.

| Prolog Performance on Warren Benchmarks | | | | |
|---|---|---|---|---|
| Prolog Benchmark | VAX 8600 | NCR 32 | PLM | Classic (Warren) |
| clock cycle | 80 ns | 150 ns | 100 ns | |
| nrev | 116k | 25k | 115k | 9k |
| qs4 | 98k | 35k | 174k | 11.2k |
| palin25 | 67k | 21k | 134k | 10.5k |
| times10 | 48k | 13k | 63k | 7.7k |
| div10 | 42k | 11k | 55k | 7.8k |
| log10 | 56k | 15k | 79k | 7.8k |
| ops8 | 65k | 21k | 106k | 11.2k |
| query | 20k | 89k | 367k | 31.9k |

Table 1

| Prolog Performance on Berkeley Benchmarks | | | | |
|---|---|---|---|---|
| Prolog Benchmark | VAX 8600 | NCR 32 | PLM | Classic (Warren) |
| clock cycle | 80 ns | 150 ns | 100 ns | |
| con1 | 95k | 53k | 305k | 43k |
| con6 | 38k | 110k | 465k | --- |
| hanoi | 106k | 59k | 310k | --- |
| mumath | 73k | 17k | 89k | --- |
| pri2 | 28k | 7k | 191k | --- |
| queens | 77k | 50k | 148k | --- |

Table 2

Several comments on these results are in order. The measurements for the Berkeley PLM are simulated results, assuming the processor never waits for memory. All other results are actual performance figures. The overall results are as expected. The simulated special purpose PLM performs best, followed by the horizontally microcoded 8600, the vertically microcoded NCR/32, and the Warren software emulator last. The 8600 results consistently fell between the NCR/32 and the PLM, except on the con6 (non-determinate concat), and query benchmarks. We attribute the analmalous con6 results to some inefficiencies remaining in the microcode, and the query results to a high frequency of escape **is** sequences requiring multiplication and division.

Finally, we should point out that our performance results include the overhead associated with a real system in a real environment. That is, the VAX 8600 is a virtual memory machine operating in a multiprogramming environment. Thus the overhead due to address translation, page fault handling, and context switching is included.

## 6. Conclusions

We have described a VAX 8600 direct execution implementation of Prolog. We have mapped the Warren architecture to the 8600 in a manner which supports a multiuser environment, and translated the Warren instruction set into a form directly executable by the 8600. To our knowledge, this system is the fastest Prolog implementation on a commercially available general purpose processor. We should also point out that our implementation method produces executable images which contain both Prolog constructs and numeric operations. As a result, this may prove to be the most effective implementation method for handling computations that have substantial symbolic and numeric components.

## 7. Acknowledgements

The authors wish to acknowledge the Digital Equipment Corporation for their generous support of our research, in particular Bill Kania for providing us with the VAX 8600 in order to enhance our ability to do research in microarchitecture and microprogramming; also, Fernando Colon Osorio, Mario Troiani, Nii Quaynor, Steve Ching and Harold Hubschman, from DEC's High Performance Systems and Clusters group in Marlboro. Our work in microarchitecture is part of a larger architectural research effort at Berkeley, the Aquarius Project. We acknowledge our colleagues in the Aquarius group for the stimulating environment they provide. Finally, we acknowledge that part of this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

## 8. References

1. D.H.D. Warren, *An Abstract Prolog Instruction Set*, SRI International, Menlo Park, CA. Technical Report, (October 1983).

2. Tep Dobry, A.M. Despain, and Yale N. Patt, "Performance Studies of a Prolog Machine Architecture," *Proceedings of the 12th Intl. Symposium on Computer Architecture*, (June 1985).

3. Peter Van Roy, *A Prolog Compiler for the PLM*, University of California, Berkeley CA. Master's Report, (August 1984).

4. Barry Fagin and Tep Dobry, "The Berkeley PLM Instruction Set: An Instruction Set for Prolog," *UCB Research Report*, CS Division, University of California, Berkeley, (September 1985).

5. Tryggve Fossum, Jim McElroy, and Bill English, "New VAX Squeezes Mainframe Power Into Mini Package," *Computer Design*, (March 1985).

6. Mark T. Shaefer and Yale N. Patt, "Improving the Performance of UCSD Pascal Via Microprogramming on the PDP-11/60," *Proceedings of the 16th Annual Microprogramming Workshop*, (October 1983).