# Compiling Prolog Into Microcode: A Case Study Using the NCR/32-000

*Barry Fagin*
*Yale Patt*
*Vason Srini*
*Alvin Despain*

Division of Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

A proven method of obtaining high performance for Prolog programs is to first translate them into the instruction set of Warren's Abstract Machine, or W-code [1]. From that point, there are several models of execution available. This paper describes one of them:- the compilation of W-code directly into the vertical microcode of a general purpose host processor, the NCR/32-000. The result is the fastest functioning Prolog system known to the authors. We describe the implementation, provide benchmark measurements, and analyze our results.

## 1. Introduction

Substantial current interest in the high performance execution of Prolog programs demands investigation into the various alternative models of execution. The classical scheme, implemented by Warren [1] among others, involves translating the Prolog program first to an intermediate form usually referred to as the instruction set of Warren's Abstract Machine, and from there to the machine language (ISP) of the host processor. Machine instructions are then interpreted by host microcode, which controls the data path of the host microengine. This process is shown in figure 1.

Since three levels of transformation exist between the Prolog application program and the host microprogram, it is reasonable to ask what improvement can be obtained by eliminating one or more of these levels of transformation. One approach employed by Dobry et. al [2] (see figure 2), was to eliminate the general purpose host ISP level and translate the W-code directly into the microcode of a special purpose host designed specifically to interpret W-code instructions. The performance advantage of this approach is significant, as will be discussed in section 6. The disadvantage, obviously, is the cost of a special purpose processor.

An alternative approach, the subject of this paper, is to again eliminate the ISP level translation, but to compile directly into the microcode of a general purpose host. (Figure 2 also illustrates this scheme). This paper describes the results of one implementation of this approach, using the vertically microprogrammable NCR/32-000 microprocessor as the host microengine.

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.

A Method for Executing Prolog

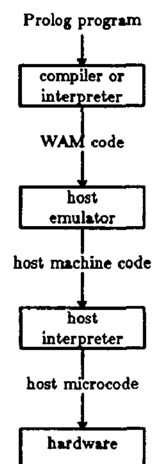



figure 1

Three Translation Schemes for Executing Prolog

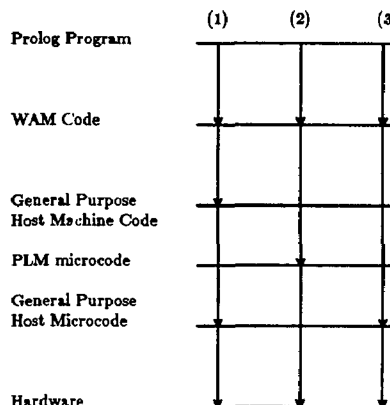



figure 2



© 1985 ACM 0-89791-172-5/85/0012/0079$00.75

This paper is divided into seven sections. Section 2 provides a short overview of the hardware and software elements that comprise our system; Section 3 discusses the Warren Abstract Machine mapped onto our host system. Section 4 describes the host hardware in more detail. Section 5 describes the implementation. Section 6 delineates the measurements which we performed with our system, compares these measurements to the alternative schemes shown in figure 2, and analyzes the results of this comparison. Section 7 offers a few brief concluding remarks.

## 2. An Overview of the System

### 2.1. Hardware

We carried out the implementation on an NCR 9300 system, containing a 64K x 16 writable control store, two megabytes of memory, and the NCR/32-000 processor (see figure 3). The NCR/32-000 is a 32-bit NMOS VLSI microprocessor. It executes microinstructions fetched from control store, communicating with it over the 16-bit IBUS. Communication with main memory takes place over a 32-bit address/data bus, the PMBUS. The 9300 system is connected to a device port on a VAX 11/750, runing 4.3 BSD UNIX.

The NCR 9300

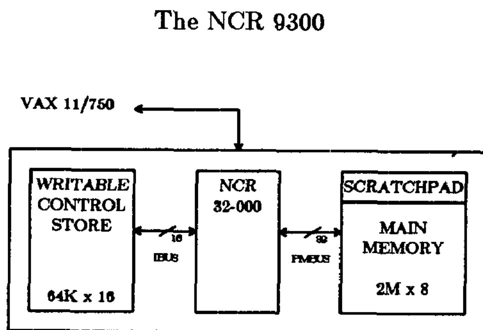



figure 3

### 2.2. Software

Three significant pieces of software are used in transforming Prolog programs to executable NCR/32-000 microcode: a Prolog compiler, a microcode compiler, and an assembler. The entire translation process, from Prolog to microcode, is shown in figure 4.

The Prolog compiler takes Prolog programs and compiles them to produce W-code. This W-code is translated into NCR/32-000 microinstructions by the microcode compiler. The resulting file is then assembled into a binary object file by the microcode assembler, which is downloaded into control store.

The Prolog compiler was developed at Berkeley as a Master's Thesis by Peter Van Roy [3]. It is written in Prolog, and is invoked from a Cprolog interpreter under 4.3 BSD UNIX. Considerable documentation on the compiler is available elsewhere [3] , [4].

The second piece of software, the microcode compiler, is the heart of the Prolog implementation. It is written in C, and expands the macroinstructions produced by the Prolog compiler into NCR/32-000 microinstructions. Each W-code instruction corresponds to a sequence of NCR/32-000 microoperations. The microcompiler reads in a W-instruction and prints out the appropriate microcode flow. Thus the output of the microcompiler is an ascii file of NCR/32-000 microinstructions corresponding to the W-coded version of the original Prolog program.

The third piece of software is the microcode assembler. It is written in C, and transforms the output of the microcode compiler into an executable file. This file may then be downloaded into the control store of the NCR/32-000 and executed.

Compilation and Assembly

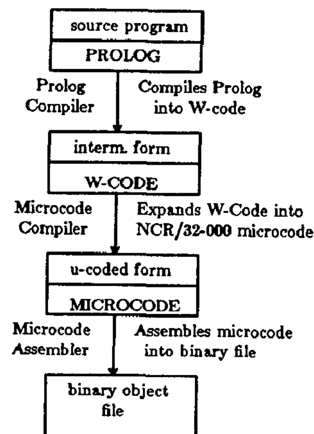



figure 4

Some of Warren's instructions invoke subroutines that perform basic Prolog functions (for example, unification). In addition to providing these subroutines, any implementation of Prolog must also support a minimal subset of builtin functions that are not part of the pure logical language. The microcode flows for all these operations are stored in a file called "basics.ncr". This file is written in NCR/32-000 microcode, since most builtin functions cannot be written with Warren's instruction set. It must be resident in WCS whenever Prolog programs are executed.

The builtin function file takes up 3% of WCS; the space taken up by the various classes of routines within the file is shown in table 1.

Percent of Reserved Control Store Area Occupied By Builtin Functions

| | |
|---|---|
| arithmetic routines | 77% |
| unify | 10% |
| set, access | 8% |
| fail | 3% |
| trail | .9% |
| output routines | .7% |
| bind | .4% |

Table 1

*Arithmetic builtins include the usual arithmetic and* comparison operations. Because arithmetic can be performed on expressions, the arithmetic category of builtins also includes code for evaluating structures. The set and access builtins are simple versions of the Prolog functions "assert" and "retract".

Finally, there must also be a program running on the VAX while Prolog programs are being executed on the NCR/32-000. This program is written in C, and is called "ncrmon". It monitors the execution of compiled Prolog programs on the NCR/32-000, assisting in I/O, making performance measurements, and providing a user interface for the system.

### 3. The Abstract Resources of Warren's Machine

The Warren machine [1] is an abstract architecture; its abstractions must be mapped onto the concrete resources of a concrete machine (in this case the NCR/32-000 micromachine). These abstractions include data types, special registers, and special areas of memory. This section explains the purpose of each of these and how they are used.

#### 3.1. Data Types

Prolog manipulates four kinds of data types: structures, lists, variables, and constants. The type of a data word is indicated by an appropriate tag. Warren's machine specification leaves the representation of each type unspecified; thus other alternatives exist to the scheme shown here.

##### 3.1.1. Constants

Constants can be of several types, including integers, atoms, floating point values, and the special constant NIL. Small integers are stored directly in the data word itself, while atoms and floating point values contain pointers to the appropriate item in memory.

##### 3.1.2. Variables

A variable is simply a data word with the variable tag in the most significant byte, whose contents are an address of some other data word. Unbound variables are represented by pointers to themselves.

##### 3.1.3. Lists

Lists are represented by a word with the list tag, pointing to the first entry of the list. List entries are one word long; our implementation of them uses cdr-coding to improve memory efficiency [5]. Conventional list representation uses two words for each entry: the car, which contains the list entry itself, and the cdr, which points to the remainder of the list. With cdr-coding, if the cdr cell corresponding to a list entry represented conventionally would point to the next word in memory, then that cell is omitted. When this is not the case, the cdr cell is left in memory, with a bit set to indicate that the word is an explicit cdr cell. This bit is called the cdr bit. Thus, to determine the location of the next entry of a list, one simply examines the next contiguous word in memory. If the cdr bit is off, then that cell is the next entry. If it is on, then the location of the next entry is pointed to by the contents of the cell.

##### 3.1.4. Structures

Structures are simply lists with principal functors. They are represented by a word with the structure tag, whose contents are a pointer to the principal functor of the structure, followed by the arguments of the structure.

#### 3.2. Special Registers

The current state of a Prolog computation on Warren's machine is defined by certain registers containing pointers to memory. Any memory subsystem supporting Warren's abstract machine must have separate address spaces for data and code; with one exception, the special registers point into the data space. This memory space is in turn divided into four areas: the heap, the stack, the trail, and the PDL. The purpose of each will be explained in more detail after the registers are discussed.

Warren's machine makes use of the following special registers:

| | |
|---|---|
| A1 - An: | the Argument registers |
| P: | the Program counter |
| CP: | the Continuation Pointer |
| E: | the Environment pointer |
| B: | the Backtrack pointer |
| TR: | the Trail pointer |
| H: | the Heap pointer |
| HB: | the Heap Backtrack pointer |
| S: | the Structure pointer |

The purpose of each of these registers is explained below.

A1-An
: The Argument registers : contain the arguments of a Prolog goal. For example, to execute the Prolog query "d(4,5,6)?", registers A1, A2, and A3 would be loaded with the tagged words representing the constants 4, 5, and 6 respectively, and then the code for procedure d would be entered. For our implementation of Prolog, n = 8.

P
: The Program pointer: contains the address of the next instruction to execute.

CP
: The Continuation Pointer: contains the address of the next instruction to execute should the current goal succeed. For example, when execution begins for the code for procedure "h" in the clause "f(X) :- g(X), h(X), i(X)", the CP would contain the address of the code corresponding to the call to i. In other words, the CP functions like a return pointer for a subroutine call.

E
: The Environment pointer: contains the address of the last "environment" pushed on the stack. (Environments will be explained shortly).

B
: The Backtrack pointer: contains the address of the last "choice point" pushed on the stack. (Choice points will also be explained shortly).

TR
: The Trail pointer: points to the top of the trail.

H
: The Heap pointer: points to the top of the heap.

HB
: The Heap Backtrack Pointer: the top of the heap at the time the last choice point was placed on the stack (i.e. the value of H corresponding to B).

S
: The Structure Pointer: Used to address elements of structures and lists on the heap. Points to the current element of a structure or list being addressed.

#### 3.3. Data Memory Allocation

The data memory is partitioned into four stacks: the control stack, the heap, the trail, and the push-down list, or PDL.

##### 3.3.1. The Control Stack

The control stack (hereafter called "the stack") is the area in memory used for storing control information. Two kinds of objects may appear on the stack: environments, and choice points.

#### 3.3.1.1. Environments

An environment represents the saved state of a Prolog clause: it contains pertinent register values, and what are known as "permanent" variables. Permanent variables are variables needed by more than one goal in the body of a clause; they must be saved so that succeeding goals can access them.

For example, consider the following Prolog clause:

f(X,Y) :- g(X), h(X,Z).

At the beginning of executing the code corresponding to this clause, an environment will be allocated for it on the stack, and the data word representing the variable X would will be stored within it. Thus, after executing the code for the procedure g, h will be able to access X by referring to the location of X on the stack. X is a "permanent" variable because it occurs more than once in a clause, and its last occurrence is after the first goal. If its value were not saved in an environment on the stack, other goals would not be able to reference it.

By contrast, the clause

f(X,Y) :- g(X), h(Z).

has no permanent variables, because the second occurrence of the variable X is in the first goal. The clause

f(X,Y) :- g(Z), h(W).

similarly has no permanent variables, because no goal requires access to the variables of another. Formally, a variable is temporary if it occurs in at most one goal of a clause, where the head is considered part of the first goal. All variables that are not temporary are permanent.

Environments also contain the values of certain registers, to enable restoration of the state of a computation when the last goal in the clause succeeds. Environments contain the following register values:

CP : where to continue once clause succeeds
E : location of last environment on stack
N : size of last environment
B : location of last choice point

#### 3.3.1.2. Choice Points

A choice point is a group of data words containing sufficient information to restore the state of a computation if a goal fails, and to indicate the next procedure to try. Choice points are placed on the stack by special instructions when a procedure is entered that contains more than one clause that can unify with the current goal. For example, as the following Prolog program fragment is executed

g(X) :- f(X), h(X, X).
g(X) :- a(X), b(X, Y).

g(X)?

a choice point would be placed on the stack when the first clause is entered, because should it fail an alternative clause exists which is to be tried as well.

Choice points contain the following register values:

An: the contents of the argument registers
E : location of last environment
CP : address of next clause to execute should this one succeed
B : location of previous choice point
TR: the value of the trail pointer when choice point built
H: the top of the heap when choice point built
N: the number of permanent variables in the environment
L: address of next clause to try should current goal fail.

### 3.3.2. The Heap

The heap is the area of data memory used for the storage of lists and structures, which are too cumbersome to be kept in environments on the control stack. The primary purpose of the heap is for the storing of lists and structures. The choice of name for this area of memory is unfortunate, because it is actually allocated incrementally like a stack, and deallocated in variable size blocks.

### 3.3.3. The Trail

When a variable becomes bound during the course of a Prolog program, it may become necessary to undo the binding when backtracking is done. Thus some method is needed for keeping track of all bindings that are to be undone when the current goal fails, so that the variables they refer to can be unbound again. For example, in the Prolog program

f(a).
f(b).
g(b).

f(X), g(X)?

X would first be bound to a, but "g(a)" would have no solution. Thus the binding of X to a must be undone. X will then unify with b, and "g(b)" will succeed.

A small stack called the trail is used to handle the necessary bookkeeping for bindings that will have to be undone upon goal failure. This stack is addressed by the TR register. When a binding is trailed, a pointer to the variable just bound is pushed onto the trail, (in the previous example a pointer to the variable X), and the TR register incremented. Upon goal failure, all variables pointed at by pointers on the trail, from the top of the trail down to the previously saved TR value in the current choice point, are reset to unbound variables. This is done as part of the 'fail' operation, explained in the section on basic operations.

It should be noted that not all bindings need to be trailed, and hence some runtime optimization is possible. Suppose the variable being bound is on the stack. If it is located above the current choice point (assuming the stack grows upwards) then it will be thrown away on goal failure; hence the binding will not have to be explicitly undone. Similarly, if the variable being bound is on the heap, then if it is located above the address in the HB register (assuming the heap grows upwards) then it will be discarded on goal failure. Thus whenever a binding is made, we compare its address with the appropriate register (B or HB). Only if the address is less than the B register (for a variable located on the stack) or the H register (for variables located on the heap) is the address of the variable pushed onto the trail. This reduces both trail space and memory traffic, at the cost of extra microcycles when trailing bindings. Special comparison logic could further reduce this cost.

### 3.3.4. The PDL

The PDL is a small stack created for the unification of nested structures and nested lists. Consider the problem of unifying the lists '[a,[b,c,d],e]' and '[a,[b,c,d],f]'. Both objects are lists, and their first elements match. The second element in each object is also a list, and as we traverse down it we find that each of their elements match. However, elements 'e' and 'f' are now inaccessible. They are pointed to by the cdr cell after the sublist '[b,c,d]', whose address we neglected to save. This problem is solved by pushing pointers to points where unification of a nested data object is to continue onto a stack; in Warren's machine, this stack is the PDL. When the end of a substructure is encountered, the topmost entry on the PDL is popped off and unification continues at the point that entry

indicates.

Either depth first or breadth first traversal of nested structures is possible. However, since Prolog structures tend to be long rather than deep, depth first traversal uses less PDL space and is hence preferable. With depth first traversal, the maximum value of the PDL will be the maximum depth of nesting of a structure in the program, whereas with breadth first it will be the maximum number of arguments of a structure or entries in a list. The former tends to be much smaller than the latter.

For more information on Warren's machine, see [1] , [6].

## 4. The NCR 9300 System

A diagram of our host system, the NCR 9300, is shown in figure 4.1.[1] The major components of the system are the NCR/32-000 CPC (Central Processor Chip), the ISU (Instruction Storage Unit), and main memory [7].

### 4.1. The CPC

The datapath of the NCR/32-000 CPC is shown in figure 4.2. The CPC is controlled by a vertically encoded 16 bit microinstruction, shown in figure 4.3. The G field always contains part of the opcode. The H and I fields may contain either an extended opcode, or may specify operand selection. The J and K fields are register specifiers that determine the operands of the instruction. A few instructions require a trailing 16-bit literal; this is supplied as the L field.

The CPC contains sixteen 32-bit general-purpose registers, referred to collectively as the RSU, or Register Storage Unit. Four of these registers are byte-addressable. The source or sink of a microinstruction is usually an RSU register. These registers are the most important resource of the NCR/32-000, as we shall see.

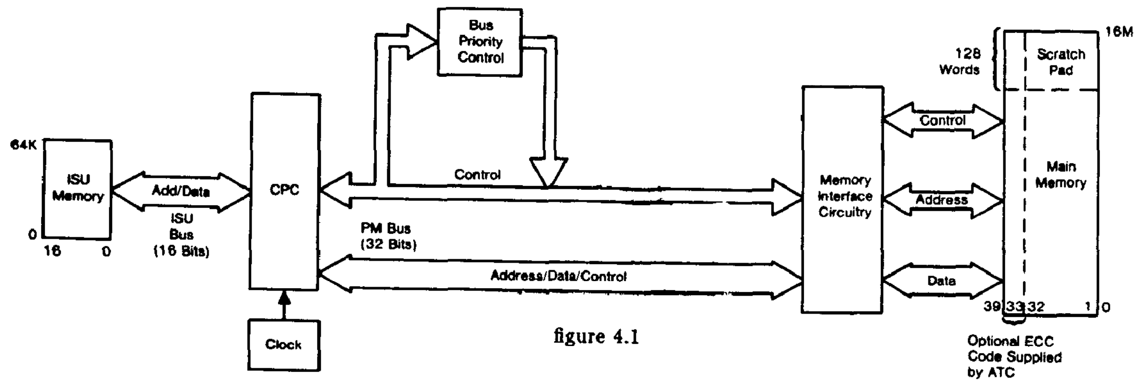



figure 4.1

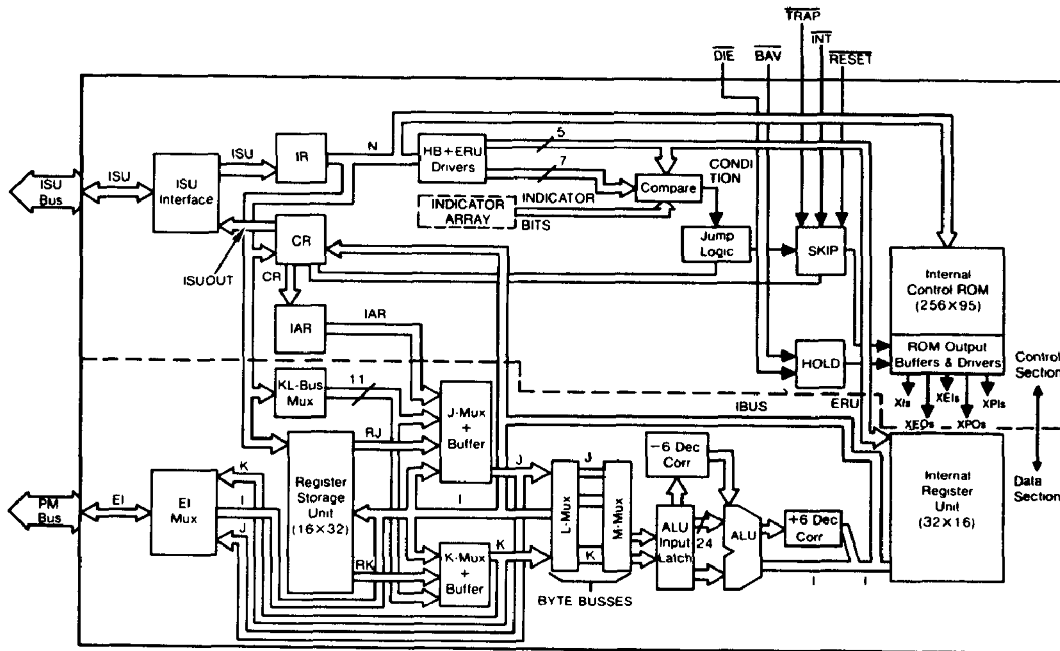



figure 4.2

[1]The NCR 9300 also includes an Extended Arithmetic Chip, for floating point calculations, and an Address Translation Chip, for virtual memory support. These chips were not part of our implementation, and are not shown in the figure.

NCR/32-000
Microinstruction Format

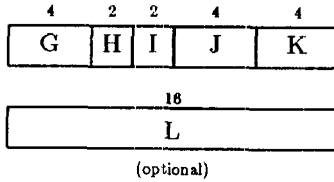



figure 4.3

The CPC also contains thirty-two special-purpose registers, called IRUs, or Internal Register Units. These include a builtin stack pointer, an indicator array for preserving the results of conditional tests, and jump registers for holding branch addresses.

There are ninety-six other register assignments, external to the CPC, called ERUs or External Register Units. (ERU's are not shown in the figure). These units include special registers that provide support for indirect accessing of the top 128 locations of main memory. These locations are referred to as the scratchpad; their use will be explained further in the section on main memory.

An instruction is fetched from ISU at the address stored in the CR, or control register. The fetched instruction is stored in the IR, or instruction register. The G field of the instruction is sent from the IR to address a small on-chip nanocode ROM, which drives various control points. Other fields of the instruction address the RSU to determine which registers, if any, are to be used. The outputs of the RSU may be sent out to the PMBUS for memory accesses, or sent to the ALU for computation. The output of the ALU may be sent back to the CR if it is used for determining the address of the next instruction, or it may be used to address the IRU. The results of ALU operations affect the indicator array, whose contents may be tested with bit patterns supplied from the current microinstruction. The result of such a comparison can be used to modify the contents of the CR.

The NCR/32-000 has a three stage pipeline, in which instruction fetch, decode, and execution are overlapped. Thus the control path includes "skip" logic, to void the pipeline when necessary.

The processor has a 150ns, two-phase clock.

### 4.2. ISU

The ISU, or Instruction Storage Unit, is a 64K x 16 writable control store. The CPC accesses the ISU through a 16 bit ISUBUS, multiplexing addresses and data. To run Prolog programs, assembled code is downloaded directly into the ISU, for execution by the CPC.

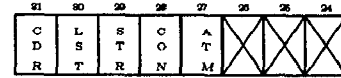
### 4.3. Main Memory

Currently, our system uses 2 megabytes of main memory, organized into 32-bit words. The CPC communicates with main memory over the PMBUS, a 32-bit address/data bus. If a memory location is to be accessed that is not part of the scratchpad, its address must be supplied from an RSU specified by the microinstruction. Thus for most of main memory, extra instructions are required to generate the address and place it in the correct RSU. Address generation for the scratchpad, however, is faster, as we shall see in the next section.

## 5. Allocating the Resources of the NCR 9300 System

We now show how the concrete resources of our host system were used to support the abstractions of Warren's machine. Since the NCR/32-000 has a 32-bit data path, the main data element was chosen to be a 32-bit word, with the upper byte reserved for the tag. Figure 5 shows the tagging scheme used. While this scheme wastes space, it offers the advantage of fast determination of the type of a data element, using the byte addressing and masking capabilities of the NCR/32-000.

Tagging Scheme




VAR = x0000000
LIST = x1000000
STRUCT = x0100000
CONST(INT) = x0010000
CONST(ATOM) = x0011000

figure 5

Our 9300 system contains two megabytes of memory, used for the trail, the stack, and the heap. Because we anticipated needing far more heap space than stack or trail space, main memory was allocated according to figure 6. The trail grows down, while the stack and the heap grow up.

Main Memory
Allocation

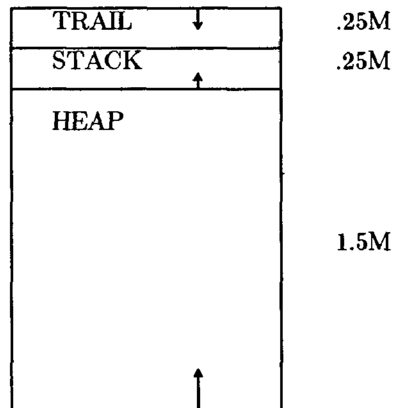



figure 6

To determine the most efficient way to map Warren's registers onto those of the NCR/32-000, we examined approximately 600,000 register references made during execution of two benchmark sets of Prolog programs, using our Prolog simulator. The first benchmark set is from Warren's thesis [8]; we refer to it as the Warren Benchmark Set. The second set has been developed at Berkeley for exercising all the instructions of Warren's machine; we refer to it as the Berkeley Benchmark Set. Both benchmark sets are shown in tables 2 and 3.

The Warren Benchmark Set

| Name | Task |
|---|---|
| nrev | reverse 30-element list |
| qs4 | sort 50-element list |
| palin25 | itemize 25-element list |
| times10 | symbolic differentiation |
| div10 | " |
| log10 | " |
| ops8 | " |
| query | extract info from database |

Table 2

The Berkeley Benchmark Set

| Name | Task |
|---|---|
| con1 | concat [a,b,c] to [d,e] |
| con6 | nondeterministic concat |
| hanoi | tower of hanoi, 8 disks |
| queens | queens problem, 4 x 4 board |
| pri2 | find all primes < 98 |
| mutest | prove a theorem |
| ckt2 | design 4-1 MUX using NANDs |

Table 3

The percentage breakdown of register references for these benchmarks is shown in table 4:

Register References

| | |
|---|---|
| An | 27.7% |
| P | 22.1% * |
| H | 12.8% |
| E | 7.5% |
| B | 7.0% |
| CP,N,S,TR,HB | 5% or less |

Table 4

* P is the program counter, implicit in the NCR/32-000.

Some registers were needed to hold intermediate results, and others to pass parameters to microroutines. Thus the results in the above table led to the allocation of registers shown in figure 7. Registers 0 and 1 were used as parameter registers for microsubroutines. Registers 2 and 3 were used for temporary values, as were registers E and F. Registers 4 through 7 were the Warren argument registers A1-A4, while the rest of the registers were assigned to the more important principal registers of Warren's machine.

NCR/32-000
Register Assignment

| | | | |
|---|---|---|---|
| 0 | parameter register 0 | 1 | parameter register 1 |
| 2 | scratchpad register 0 | 3 | scratchpad register 1 |
| 4 | A1 | 5 | A2 |
| 6 | A3 | 7 | A4 |
| 8 | E | 9 | CP |
| A | B | B | H |
| C | TR | D | HB |
| E | scratchpad register 2 | F | scratchpad register 3 |

figure 7

Because there are more special registers in Warren's machine than will fit in the NCR/32-000 register file, the scratchpad memory of the 9300 was also used. Unlike the rest of main memory, which requires an address stored in an RSU to specify the desired location, the scratchpad can be addressed either indirectly through special ERU's or directly through a field in the microinstruction. Since scratchpad locations do not require extra microinstructions to set up the desired address, it was treated specially in the implementation, and was allocated as shown in figure 8. Locations 0-4 are reserved for use by ECD, the Extended Console Debugger booted out of PROMs when the system is powered up. Location 4 holds the address of the base of the stack. Location 5 holds the cut flag, a flag needed for implementing the Prolog 'cut' operator. Locations 6 and 7 are the N and S registers, location 8 holds the mode bit (used to implement structure copying), and locations 9 and A are unused. Location B holds the time of execution of the program (used for performance measurements), and locations C-F hold the remaining argument registers A5-A8. Finally, locations 10-3F are used for the PDL, while 40-5F are used for a microsubroutine stack. The topmost location of this stack is indicated by IRU26, the NCR/32-000 stack pointer. The remaining scratchpad locations are unused.

Scratchpad
Allocation

| | |
|---|---|
| 0 | reserved by ECD |
| 1 | reserved by ECD |
| 2 | reserved by ECD |
| 3 | reserved by ECD |
| 4 | stack base (x180000) |
| 5 | cut flag |
| 6 | N |
| 7 | S |
| 8 | MODE |
| 9 | unused |
| A | unused |
| B | program execution time |
| C | A5 |
| D | A6 |
| E | A7 |
| F | A8 |
| 10-3F | PDL |
| 40-7F | microsubroutine stack |

figure 8

### 5.1. Two Sample Microcode Flows

We consider now two examples of microroutines produced by the microcode compiler: the routines for the W-code instructions "put_variable Yi,Aj" and "call proc,n". The form of "put_variable" instruction discussed here must put an unbound variable at location Yi, and place a pointer to it in argument register Aj. The "call proc,n" instruction must load the number n into the N register, save a pointer to the following code in the CP register, clear the cut flag, and branch to the address represented by "proc".

Both of these flows contain macros referring to various abstractions of Warren's machine; "er" for E register, "a1" for register A1, and so forth. These macros are translated by the microcode assembler into the appropriate bit patterns. To avoid confusion, the registers of Warren's machine will be referred to as abstract registers.

Finally, the microinstructions used in the flows have the following meanings:

```
aw      add word
bew     boolean exor word
djor    delayed jump on reg
lit     literal value
lrhc    load right halfword,
          clear left halfword
s       store
sl      store literal
          (used for scratchpad)
tw      transfer word
```

### 5.1.1. The Flow For 'put_variable Yi,Xj'

Let us assume that i and j are both 1. The microcode compiler will accept the instruction "put_variable Y1,X1" and expand it to:

```
; put_variable Y1,X1

tw      er,screg0
lrhc    screg1
lit     10
aw      screg0,screg1
tw      screg0,a1
tw      screg0,screg1
s       F,screg1
```

The first instruction transfers the contents of abstract register E to scratch register 0. Next, the hexadecimal literal 10 is added to it, computing the location of the variable Y1. This address is transferred to abstract register a1, and to temporary register 1 as well. The last instruction stores the contents of the specified register at the address stored in the register with which it is paired. In the NCR/32-000, odd numbered registers are paired with their immediate predecessors, so scratch registers 0 and 1 are paired. Thus the final instruction stores the address of Y1 in the location of Y1, making Y1 point to itself. This is how unbound variables are represented.

### 5.1.2. The Flow For 'call proc,n'

Let us assume that n = 2. Upon reading "call proc,2", the microcode compiler generates the following:

```
; call proc,2

lrhc    screg1
lit     2
sl      nr,screg1

lrhc    cpr
lit     $+6
lrhc    screg0
lit     proc
djor    0,screg0
bew     screg1,screg1
sl      cutflag,screg1
```

First, the number 2 is loaded into a temporary register, and then stored into abstract register N. Note that since N is actually in scratchpad memory, its address does not have to be computed ahead of time. An "sl" instruction can be used, in which the address is supplied as part of the microinstruction. Next, the address of the following code in the I-stream is loaded into abstract register CP. The address of "proc", computed by the microassembler, is loaded into scratch register 0, and then a delayed branch to that address is executed. The next two instructions clear scratch register 1, and use the resulting zero value to clear the cut flag, stored in scratchpad. The delayed branch then takes effect, and control proceeds to the address of "proc".

Implementing Warren's machine on the NCR/32-000 involved four phases: 1) deciding upon the format for the data types, 2) allocation of address space and registers, 3) construction of microroutines for basic Prolog operations, and 4) construction of microroutines for the W-instructions. Steps 1 and 2 were explained in a previous section; here we consider 3 and 4.

Two Prolog functions are not directly accessible to the user, but are instead called during execution by several W-code instructions. These "basic" functions are unification and failure. Since unification and backtracking on failure are perhaps the two most important features of Prolog and consequently lie at the heart of any Prolog implementation, these routines were constructed first. We believed that once implementing unification and failure was fully understood, the rest of the Warren Machine would follow easily.

The fourth and longest phase of the project was the construction and testing of the microroutines for each of Warren's instructions. Fortunately, the semantics of each instruction are well defined, and have been explained in detail in previous work [6] , [9]. Constructing the microroutines consisted of converting the C routines emulating each instruction in [9] into NCR/32-000 microinstructions, using the allocation of address space and register resources decided upon in phase 2. We parenthetically observe that this translation was sufficiently straightforward to suggest investigation of the feasibility of automatic construction of the microroutines for other target architectures.

## 6. Performance Results

### 6.1. Measurements

We used the two sets of benchmarks in table 2 and table 3 to measure performance. These programs were compiled, downloaded, and executed on the NCR/32-000 using the software described in section 2. Tables 5 and 6 summarize our measurements and compare them to those obtained for two other systems: the Berkeley PLM and (where results were available) Warren's compiled Prolog running on a DEC-10. Recall that the Berkeley PLM is a special purpose Prolog processor that interprets Warren's instructions directly, while Warren's DEC-10 Prolog executes according to the scheme of figure 1.

#LIPS on Warren Benchmark Set

| Name | NCR/32 | Berkeley PLM | DEC-10 |
|---|---|---|---|
| nrev | 25K | 115K | 9.3K |
| qs4 | 35K | 174K | 11.2K |
| palin25 | 21K | 134K | 10.5K |
| times10 | 13K | 63K | 7.7K |
| div10 | 11K | 55K | 7.8K |
| log10 | 15K | 79K | 7.8K |
| ops8 | 21K | 106K | 11.2K |
| query | 89K | 367K | 31.9K |

Table 5

#LIPS on Berkeley Benchmark Set

| Name | NCR/32 | Berkeley PLM |
|---|---|---|
| con1 | 53K | 305K |
| con6 | 110K | 465K |
| hanoi | 59K | 310K |
| queens | 50K | 148K |
| pri2 | 7K | 191K |
| mutest | 20K | 89K |
| ckt2 | 17K | n.a. |

Table 6

Since performance on the deterministic concatenate benchmark seems to be accepted as a standard measure of Prolog performance, we have included tables 7 and 8, showing the performance on this benchmark for several known and planned systems.

Performance Figures for Deterministic Concatenate, Existing Systems

| Machine | System | #LIPS |
|---|---|---|
| **NCR/32** | Warren, Compiled | 53K |
| DEC 2060 | Warren, Compiled | 43K |
| SUN-2 | Quintus Compiler | 40K |
| IBM 3033 | Waterloo | 27K |
| VAX-780 | Macrocoded | 15K |
| LMI/Lambda | Uppsala | 8K |
| VAX-780 | POPLOG | 2K |
| VAX-780 | M-ROLOG | 2K |
| VAX-780 | C-PROLOG | 15K |
| SYMBLOICS 3600 | Interpreted | 1.5K |
| PDP 11/70 | Interpreted | 1K |
| Z-80 | MicroProlog | .12K |
| Apple-II | Interpreted | .008K |

Table 7

Performance Figures for Deterministic Concatenate, Planned Systems

| Machine | System | #LIPS |
|---|---|---|
| Berkeley PLM | TTL/Compiled | 425K |
| TICK & WARREN | VLSI | 415K |
| Aquarius I | TTL/Compiled | 305K |
| J 5th Gen HPM | Microcoded | 280K |
| SYMBOLICS 3600 | Microcoded | 110K |

Table 8

The figures in table 8 warrant some explanation. The Berkeley PLM measurement is based on simulation, assuming a memory as fast as the machine itself. The Tick and Warren figure is estimated, and the Aquarius I figure reflects the current system with a memory three times slower than that of the processor. The remaining results are all estimated. Note that the NCR/32-000 implementation compares quite favorably with other existing Prolog systems. In fact, this implementation is the fastest fully functional Prolog known to the authors. Even after the various research efforts with faster estimates become viable, it will still remain the fastest Prolog available on a non-symbolic (i.e. general purpose) processor.

### 6.2. Analysis

In this section we discuss where the existing performance of the system comes from, and where we feel potential performance was lost. We also discuss barriers to improved performance.

We believe the performance of this system is due to three factors: 1) compiling Prolog instead of interpreting, 2) compiling directly into microcode instead of interpreting (saving fetch and decode cycles), and 3) the pipelining of the instructions by the NCR/32-000.

With compiled Prolog, Prolog clauses and structured data types are represented as sequences of W-code instructions. This is much more efficient than representing the entire program as a data structure and traversing it interactively. Looking at the figures for deterministic concatenate, we see that the best results have been reported for compiled systems, while the poorest results have been reported for interpreted systems. Thus it is not surprising that compiling Prolog helped improve performance on the NCR/32-000.

Performance was also achieved by compiling directly into microcode, as opposed to having the Prolog microroutines merely resident in microstore and invoked by macroinstructions. By executing microinstructions directly out of microstore, the overhead associated with opcode cracking and reaching the appropriate microcode flow is eliminated.

One obvious disadvantage of this approach is that it uses large amounts of microstore. Compiled programs require an inordinate amount of memory, as shown below:

Compiled Benchmark Size

| Name | approx. size in bytes |
|---|---|
| nrev | 8K |
| qs4 | 8K |
| palin25 | 16K |
| diff | 16K |
| query | 8K |
| ckt2 | 32K |
| con1 | 2K |
| con6 | 2K |
| hanoi | 2K |
| mumath | 8K |
| pri2 | 4K |
| queens | 8K |

Table 9

Since the NCR/32-000 addresses at most 64K 16-bit microinstructions, and since the original Prolog sources for the above benchmarks are relatively small, large Prolog programs will have to be broken up and swapped in to WCS from main memory. If, however, we can swap microcode concurrently, then the steadily decreasing cost of memory leads the authors to conclude that the space/time tradeoff is a good one.

The final source of improved performance is the pipelining of the NCR/32-000 and the use of delayed branches. The NCR instruction set provides eleven delayed branching instructions. These instructions were used whenever possible in the Prolog microroutines. Judicious use of delayed branching kept pipeline flushes to a minimum.

What limits further performance gains? To answer this question, we must consider the process of implementing an abstract machine architecture. To implement an abstract architecture efficiently, the host machine should be as closely mapped to it as possible. The better the mapping, the higher the performance. When the mapping is less than ideal, performance is lost as the implementer attempts to make up for what he does not have, or tries to have one scarce resource serve two different functions. This problem came up over and over again in the course of this project. Thus it is believed that imperfections in the mapping from Warren's abstract machine to the NCR/32-000 are the principal barriers to improved performance. We now consider some of these.

Warren's machine has four basic data types, indicated by a tag. Thus architectural support for tag extraction and processing is crucial to a successful implementation of Warren's machine. Unfortunately, the NCR/32-000 is a general-purpose microprocessor, and provides relatively little support of this kind. Only four of its sixteen registers are byte-addressable, and none are addressable at any smaller level of granularity. Thus, to determine the data type of a value, it is necessary to first move it into a byte-addressable register. One must then either use a conditional "jump-on-register-byte" instruction, or load a literal value into another byte addressable register and use a "compare-byte" microinstruction, branching on the result of the comparison. Both of these operations take extra microcycles, due to the difficulty of tag extraction from data values. Ideally, one would want a new microinstruction that does a four-way branch based on a two-bit field of a data value.

Another performance limitation comes from the scarcity of registers on the NCR/32-000. In addition to performing at least some of the functions of the special purpose registers on Warren's machine, NCR/32-000 registers must also be used for storing temporary values, passing parameters to microroutines, and holding addresses and data for memory accesses. Thus some of Warren's abstract registers cannot be mapped onto real NCR/32-000 registers, and must instead be located in scratchpad. (Recall figures 7 and 8). This in turn leads to wasted microcycles when these registers must be accessed.

One of the more curious features of the NCR/32-000 that hinders Prolog performance (and, perhaps, performance in general), is the grouping of registers in address/data pairs. When storing to memory, only the register containing the data is specified in the instruction: the address where the data is to be written must be stored in the corresponding data register. With the Warren machine, however, any of the special registers could serve as address registers; associating a separate data register with each one of them would have forced even more resources out of the register file and onto scratchpad. Thus the implementer spends a great deal of time moving addresses and data into paired registers. Currently, the "store" microinstruction of the NCR/32-000 devotes eight bits to the opcode, four bits to write-enable byte tags, and four bits to indicate the data register of the register pair. For systems where writing individual bytes of memory is not of interest, the byte-enable tags could be replaced by a four bit field encoding the data register. The extra complexity necessary to implement this feature would be negligible, and would afford a great deal of flexibility to the assembly language programmer. For systems where the ability of writing to individual bytes of memory is important, the write-enable tags could be moved to an external register, which could be loaded on the previous cycle with a "transfer-out-external" (TOE) microinstruction (see [7]).

In general, further performance improvements are limited by the general-purpose nature of the NCR/32-000. Prolog is a symbolic language; achieving high Prolog performance requires support for symbolic processing. However, it is not necessary that microprocessors give up their general purpose nature in order to execute symbolic languages efficiently. By making modest changes involving support for tagging and a more flexible instruction set, the NCR/32-000 (and microprocessors in general) can efficiently execute symbolic languages without losing performance on other classes of problems.

## 7. Conclusions

This paper has presented the results of an attempt to implement Prolog on the NCR/32-000 microprocessor. Using the results of simulating the execution of Prolog programs, we mapped the Warren abstract architecture onto the resources of our NCR/32-000 host, and wrote the required software to bridge the levels of translation. In so doing, we have been able to achieve the fastest Prolog currently available. We were also able to gain insight into how processors might be modified to support symbolic languages, as well as insight into the architectural issues involved in supporting Prolog.

## Acknowledgements

The authors gratefully acknowledge the work of Tep Dobry, whose simulator of Warren's machine made this investigation possible. This work was partially sponsored by DARPA, and monitored by Naval Electronic System Command under Contract No. N00039-84-C-0089 and Contract No. N0039-83-C-0107. Part of this work was also sponsored by the California MICRO program. Finally, thanks are due to NCR Corporation, for their generous donations of equipment and software.

## References

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

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

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

4. Wayne Citrin and Peter Van Roy, "Compiling Prolog for the Berkeley PLM," *Proceedings of the 19th Hawaii International Conference on System Sciences*, (1986). To appear.

5. Tep Dobry, Yale Patt, and A. M. Despain, "Design Decisions Influencing the Microarchitecture For A Prolog Machine," *MICRO 17 Proceedings*, (October 1984).

6. 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).

7. NCR Corporation, *NCR/32 General Information*. 1983.

8. David H. D. Warren, *Applied Logic -- Its Use and Implementation as a Programming Tool*, University of Edinburgh, Scotland (1977). Ph.D. Thesis.

9. Tep Dobry, *PLM Simulator Reference Manual*, Computer Science Division, University of California, Berkeley, CA (July 1984). Technical Note.