TOWARDS A HIGH PERFORMANCE PROLOG PROCESSOR

David Warren, February 1983

ABSTRACT

In this note we consider how one might design a high performance Prolog processor, by exploiting low-level parallelism in Prolog execution. We focus chiefly on potential parallelism in the implementation of unification.

N.B. The ideas expressed here are very tentative, and this account of them is very sketchy.

INTRODUCTION

Many tasks for which Prolog seems highly suited are not really practical on current machines because they can be implemented more efficiently in lower-level languages; examples include text editors, document formatters, and key parts of operating systems. This limitation of Prolog would completely disappear, however, if one logical inference (i.e. resolution) took the same order of time as a conventional machine instruction. Is it possible to design special Prolog hardware that can perform resolutions as fast as a conventional processor executes instructions, and that has a cost comparable with a conventional processor? To put it more concretely, can we build a one megalips (i.e. one million logical inferences per second) Prolog machine for the same kind of cost as a one mips VAX, for example? This note presents some initial ideas towards this goal.

We wish to exploit certain low-level parallelism in Prolog, by analogy with the way a conventional ALU exploits parallelism in arithmetic operations. By "low-level" parallelism, we mean parallelism that is invisible to the Prolog programmer. Of course logic programs seem to offer great potential for large-scale parallelism, but this kind of parallelism cannot be exploited without the programmer being concerned. It will require a fundamentally new approach to logic programming. Prolog's control mechanisms (including cut) and use of side effects will have to be replaced by something new. Since exploiting large-scale parallelism requires major advances in both programming methodology and computer architectures, it is clearly a subject for longer term research. What we propose here is less radical, and hopefully can be realised sooner.

Our goal is to take a conventional Prolog system on a von Neumann machine, and speed up its innermost mechanisms. Briefly, this means replacing the central processor, but leaving the organisation of main memory essentially unchanged.

If we look at a typical resolution, such as the 'concatenate' loop, we see that it requires of the order of 100 basic operations. On the DEC-10, for instance, the 'concatenate' loop compiles into 50 machine instructions (see Appendix II), and runs at around 30,000 lips on a KL-10 processor. For the Psi machine being designed at ICOT, the performance is expected to be similar, in the region of

30,000 lips (or perhaps a bit more). This machine is esssentially a state-of-the-art Prolog interpreter, implemented in firmware rather than software. The microinstructions themselves are relatively primitive operations, not particularly specialised towards Prolog execution. The micoinstruction cycle time is estimated at 200 nanoseconds, from which one can infer that one resolution will take about 150 microinstructions.

All work to date on improving the performance of Prolog, including the design of Psi, has been concerned with making the basic operations run faster. The operations themselves are performed in a strictly sequentiel manner. The processor does only one thing at a time, and that thing is painfully primitive. So long as the basic operations are performed strictly sequentially, there is a limit to the performance that can be achieved with current hardware technology; perhaps 100,000 lips is the best that can be hoped for in a machine of moderate cost. So the goal of one megalips performance seems to necessarily imply some degree of parallelism in the performance of the basic operations of [illegible]og.

DESIGN CONSIDERATIONS

In thinking about possible designs, we will tend to first assume a non-structure-sharing implementation. The reason is that non-structure-sharing is conceptually simpler, and amongst other things simplifies garbage collection. Structure-sharing may well not be worthwhile if the alternative, copying, can be implemented efficiently by hardware. However, there are counter-arguments, and we will certainly keep structure-sharing as an option to be considered.

The main problem in designing a higher performance Prolog processor seems to be the bottleneck between the processor and main memory. To achieve one resolution per micosecond will certainly require a very high traffic between the processor and main memory. For example, at first sight, one resolution of 'concatenate' requires the processor to read 22 data items and write 8 data items:

```
                                   --------------
[......clause......] 14 ---> !              !
[....goal....] 4 -----------> !    Prolog    ! ------> [....goal....] 4
[var] 1 ---------------------> !   processor  ! ------> [var] 1
[list record] 3 ------------> !______________! ------> [list record] 3
```

However with "tail recursion optimisation" and a cache of sufficient size, the number of main memory data accesses can hopefully be reduced to 3 items read and 3 items written:

```
                Reads                                         Writes
                                   ------------------
                                  !                  !
0 musec   ------------------->    !      Prolog      !
1 musec   [cons!a1!l1!..]         !     processor    !
2 musec   [cons!a2!l2!..]         !                  !    ------------------->
3 musec   ...                     !__________________!    [cons!a1!l1'!..]
...                                                       [cons!a2!l2'!..]
                                                          ...
```

The functions to be performed by the Prolog processor are, basically, unification and backtracking. (Proceure invocation can be viewed as a special case of unification). Generally speaking, the bulk of the processing is unification. One can think of a Prolog processor as a machine which can move, test, and compare symbolic data very quickly. There is little or no need for the usual arithmetic/logical operations on binary data.

PARALLEL UNIFICATION

Our first proposal for speeding up Prolog is a processor which executes unification steps in parallel. The key observation here is that it is immaterial what order the steps of a unification are performed. Although different orderings may result in different variable bindings, the results are equivalent according to the semantics of logic. This is a very nice property. Contrast the effect of assignment in traditional languages.

An example of how a unification might be performed in parallel follows. For the time being we will ignore the copying (renaming) of input terms which must take place prior to or during unification; we consider only the unification of constructed terms. The effect of the unification will be to compute the effect of one resolution of 'concatenate'; 'resolve(G,G')' holds if G is a goal stack and G' is the new goal stack which results from doing one resolution.

```
 resolve(conc(cons(a,b),c,L,g),G') =
 resolve(conc(cons(X,L1),L2,cons(X,L3),G),conc(L1,L2,L3,G))
                  ___________!__________________________
                !                                      !
 conc(cons(a,b),c,L,g) =                           G' = conc(L1,L2,L3,G)
 conc(cons(X,L1),L2,cons(X,L3),G)                      !
    ____________!_________________________             !
   !             !        !               !            !
 cons(a,b) =    c = L2   L = cons(X,L3)  g = G     conc(L1,L2,L3,G) -> G'
 cons(X,L1)      !        !               !
   !______       !        !               !
   !      !      !        !               !
a = X   b = L1  c -> L2 cons(X,L3) -> L  g -> G
   !      !
   !      !
a -> X  b -> L1
```

In this example, there is only one attempt to bind each variable. In general, if unification steps are to proceed in parallel, there may be simultaneous attempts to bind the same variable, and equally there may be simultanous attempts to read the same variable or piece of structure. The hardware must allow for this. The processor we envisage will allow simultaneous reads and writes of the same storage location. All reads will proceed without conflict and will obtain the same value. Only one of the writes will succeed; the others will fail and it will be up to the firmware or software to take appropriate action on write failures. In the case of variable bindings in unification, the action required is simple--merely redo the unification step with the new value just written to the variable by another unification step. A sketch of possible hardware to realise parallel unification follows.

3 SKETCH OF A PROCESSOR FOR PARALLEL UNIFICATION

- Each Prolog symbol occupies one 32-bit word: an address plus a few type bits. For example:

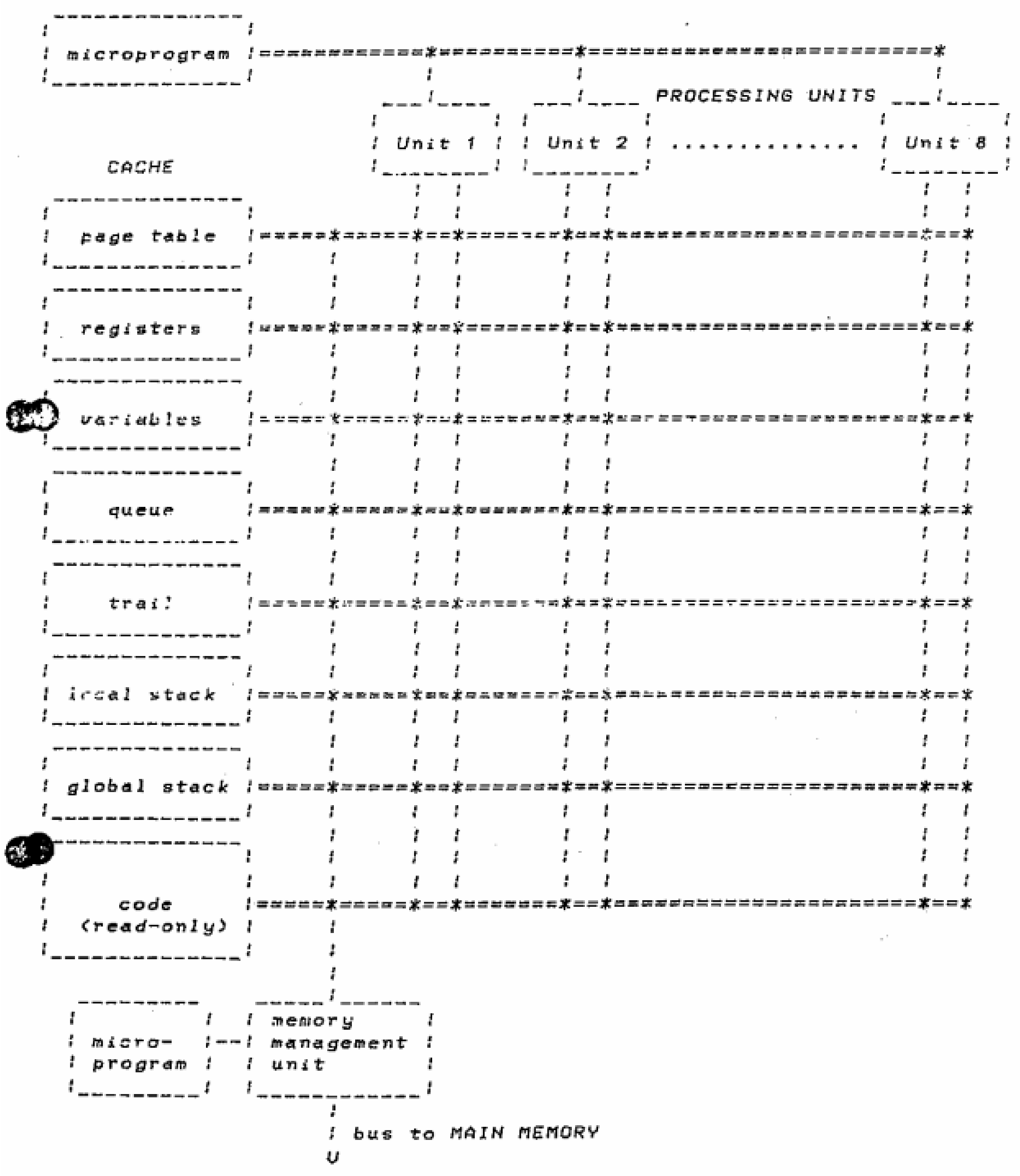
```
 ________________________________________________________
!                                           !        !   0 = blank
!          address / value                  ! type   !   1 = reference
!___________________________________________!________!   2 = atomic
0                                           30      32   3 = structure
```

- Processor A (see Figure 1) comprises:
  - (a) 8 (or fewer) (or more) processing units, each with a few internal registers a simple arithmetic unit, and two 32-bit i/o buses to a shared cache.
    Each processing unit is controlled by the same microprogram.
    Each processing unit has a cycle time ("clock tick") of 100 nanoseconds or less.
  - (b) Cache of up to 64 pages of 4 (or possibly more) words each; total 256 words.
    Associative memory maps 32-bit addresses into cache locations.
  - (c) Memory management unit responsible for handling cache page faults, and doing page i/o to main memory.
  - (d) Interleaved main memory capable of reading and writing pages of 4 32-bit words in one cycle. Every microsecond, the processor can read one page and write another.

- Each i/o bus of each processing unit can be simultaneously performing any one of the following operations in one cycle:
  - (a) Read any word from the cache. NB. Several buses may simultaneously read the same word.
  - (b) Attempt to write to any word in the cache.
    If several buses attempt to write the same word, only one will succeed; failure will be signalled to the others.
  - (c) Blank out a cache page. (Intended for use in creating new structures).
  - (d) Remap a cache page to a new page address.
  - (e) Discard a cache page.

The feasibilty of building a machine such as this is supported by the existence of the QA-1 computer designed and built at Kyoto University by Shinji Tomita and colleagues. The QA-1 hardware includes four separate ALUs which can simultaneously access a set of shared registers under control of a single 160-bit horizontal-type microinstruction. Several ALUs may simultaneously read or write the same register. The action in the case of simultaneous write is different from what we propose; on the QA-1, the values to be written are simply OR-ed together. Apart from that, the basic concept is very similar. The Kyoto team are currently designing a successor machine, called QA-2, with a similar architecture.

Figure 1 - PROLOG PROCESSOR A



MICROPROGRAMMING THE PROLOG PROCESSOR A

A non-structure-sharing unification algorithm for a single (sequential) processor is shown in Apperdix I. The intention is that the operations of each line can be performed concurrrently, as a single micro-instruction. It will be seen that a single unification step takes 6-10 clock cycles (+ 1 clock cycle for each extra dereferencing operation needed).

The hope is that, with minor modifications, this algorithm will serve as a basis for the microprogram for parallel unification, which will control each of the 8 processing units of the Prolog processor. Lower case names are intended to denote shared locations, while upper case names denote registers that are private to each processing unit. In a parallel processing situation, writes to shared locations, such as 'Q -> q' in line 'dequeue+1', may fail. In the parallel version of the algorithm, a failure action must be provided, which in this case would probably be to go back to line 'dequeue'. Similarly, a failure of the variable binding in line 'bindx' might cause a jump back to line 'switch-1'.

It will be seen that only one unification step can be initiated each clock tick (because the processing units are sharing the counter 'i'), but since unification steps take on average about 8 clock ticks, it appears possible to keep 8 processing units busy, thereby achieving a speed-up of up to 8 times.

The maximum possible throughput of the processor is therefore one unification step per clock tick. If we consider the 'concatenate' loop to consist of 14 unification steps, we might hope that this would take not much more than 14 clock ticks. To achieve 1 megalips, the clock tick would therefore have to be 70 nanoseconds. Of course, we are making lots of optimistic assumptions in this simplistic analysis. In particular, we are ignoring the question of cache page faults and the associated main memory accesses.

Also, of course, the algorithm of Appendix I needs to be extended to provide for creation of new structures, and to cover the rest of Prolog's basic machinery, namely procedure invocation and backtracking. Whether this architecture is cost effective very much depends on whether one can keep multiple processing elements busy at all stages of Prolog execution.

An interesting possibility for future consideration is to extend the basic idea to cover more general AND-parallelism in logic programs. One can view unification as a special case of AND-parallelism, where there is no nondeterminacy. It frequently happens in logic programs that there is more than one goal that is available for execution without any backtrack points needing to be created. A good example is "quicksort", where both the "split" and "sort" procedures are ready to start executing as soon as their main arguments are instantiated. It appears that the architecture already described could equally well be used to simultaneously execute more than one goal, so long as none of the goals needs to create a backtrack point.

CRITIQUE OF PROLOG PROCESSOR A

As we've seen, Prolog processor A achieves at best one unification step per clock tick. It would be possible to improve on this by a more elaborate queuing mechanism, which would allow more than one unification step to be initiated each clock tick. However, if such an improvement is not made, it can be argued that processor A is unnecessarily complex for the performance it achieves. Similar performance can probably be achieved by a more conventional architecture which pipelines unification steps. Such a machine (let's call it Prolog processor B) might well be able to perform one unification step in a single micro-instruction in most cases. A factor favoring this approach is that sticking to an essentially sequential unification algorithm allows one to optimise some of the operations. In particular, one can mark the "first" occurrence of a variable and be sure that the variable will be unbound at that point in the unification.

PIPELINING UNIFICATION - PROLOG PROCESSOR B

Consider a purely sequential unification algorithm, where a Prolog clause is represented as a sequence of Prolog instructions (one for each Prolog symbol). The commonest step will be unifying the first occurrence of a variable against the next argument of a structure. The following is the relevant routine in a program which interprets the Prolog instructions (where T is the current argument, pointed to by S, and X is the current instruction, pointed to by the program counter P):

```
unify(var,any): T -> [X]          Bind term to variable.
                S <- S+1          Advance to next argument.
                T <- [S]          Fetch the next argument.
                P <- P+1          Advance to next Prolog instruction.
                X <- [P]          Fetch the next Prolog instruction.
                unify(opcode(X),type(T))  Decode the next step.
```

This is very similar to the instruction execution cycle in a conventional machine, with the difference that there are two items which must be fetched to determine the next operation. One can think of unification as having two instruction streams, where the opcode is the concatenation of tag bits from a pair of instructions.

If each "instruction pair" is prefetched, in the same way that a single instruction is prefetched in a conventional pipelined machine, it seems possible for all the operations of a simple unification step to proceed in parallel, controlled by a single micro-instruction, say. This would mean that the commoner unification steps could be performed in a single clock tick (cf. processor A). To see this, consider a rewrite of the previous example, where all the operations are performed in parallel:

```
unify(var,any): T -> [X]          Bind term to the variable.
                T <- T'           The next term becomes the current term.
                T' <- [S]         Fetch the term after the next.
                S <- S+1          Advance the term pointer.
                X <- X'           The next instr becomes the current one.
                X' <- [P]         Fetch the instruction after the next.
                P <- P+1          Advance the program counter.
                unify(opcode(X'),type(T')) Decode the next step.
```

Here X and T are the current instruction and term, X' and T' are the next instruction and term, and P and S point to the instruction and term following that.

In general the following steps will be overlapped:

> Execute instuction N against term N.
> Decode (ie. fetch micro-instruction for executing)
> instruction N+1 against term N+1.
> Fetch instruction N+2 and term N+2.

Of course, the more complex unification steps will take more than one microinstruction to complete. If on average a unification step takes 1.5 cycles, and concatenation takes 14 steps, then the cycle time will need to be about 50 nanoseconds to achieve 1 megalips.

Prolog processor B will need separate buses (and caches?) for:

> Reading mico-instructions from the control store.
> Reading Prolog instructions from a code memory.
> Reading/writing terms etc. from/to data memory.

plus several buses for transferring data between internal registers, plus at least two adders/incrementers. It looks feasible to build such a machine from off-the-shelf bit-slice devices.

CONCLUSION

We have sketched two possible designs for high performance Prolog processors.

The first design is more radical, and aims to take maximum advantage of parallelism in unification. A major question that needs to be investigated is whether there is enough parallelism in the other aspects of Prolog execution to make the machine worthwhile. Whatever the outcome, such an architecture could be a useful first step towards realising more general AND-parallelism in logic programs.

The second design is more conventional, and perhaps more immediately practical. It involves pipelining (i.e. overlapping) the execution of unification steps with the fetching and decoding of subsequent steps. The next step in investigating this design would be to produce a more detailed outline of the microcode and analyse its performance.

ACKNOWLEDGEMENTS

In forming the ideas presented here, I have benefitted from discussions with Klaus Berkling, Bruce Hunt, Fernando Pereira, Ehud Shapiro, Bill Zaumen, and members of ICOT, especially Takeshi Chikayama and Minoru Yokota. I heartily thank Kazuhiro Fuchi and Koichi Furukawa for inviting me to visit ICOT and giving me the opportunity to develop these ideas.

APPENDIX I - Non-structure-sharing Unification Algorithm

```
 2  dequeue:     Q <- q+3;    Q < q' else succeed;
 3               Q -> q;    I <- [Q];    unify1;

 1  unify:       Q <- q;    I <- i;    I > 0 else dequeue;
 2  unify1:      I-1 -> i;
 3               X' <- [Q+1]-I;    Y' <- [Q+2]-I;
 4               X <- [X'];    Y <- [Y'];
 5  switch:      case(type(X),type(Y)) of    'BLANK'  'REF'     'ATOM'   'STRUCT'
                                  'BLANK'  [bindxy,  derefy,   bindx,   bindx,
                                  'REF'     derefx,  derefxy,  derefx,  derefx,
                                  'ATOM'    bindy,   derefy,   compare,fail,
                                  'STRUCT'  bindy,   derefy,   fail,    match];

 6  derefx:      X' <- X;    X <- [X];    switch;

 6  derefy:      Y' <- Y;    Y <- [Y];    switch;

[illegible]  derefxy:     X' <- X;    Y' <- Y;    X <- [X];    Y <- [Y];    switch;

 6  compare:     X = Y then unify else fail;

 5  match:       X = Y then unify;
 7               I <- arity(X);    I = arity(Y) else fail;
 8               Q <- q';
 9               Q+3 -> q';    I -> [Q];
10               X -> [Q+1];    Y -> [Q+2];    unify;

 6  bindx:       Y -> [X'];    X' < bg else unify;
 7               TR <- tr;
 8               TR+1 -> tr;    X' -> [TR];    unify;

 6  bindy:       X -> [Y'];    Y' < bg else unify;
 7               TR <- tr;
 8               TR+1 -> tr;    Y' -> [TR];

 6  bindxy:      X' = Y' then unify
                 X <- X';    Y <- Y';    X' < Y' then bindy else bindx
```

APPENDIX II - The Concatenate Loop in Compiled DEC-10 Prolog

```
 L Symbol          Code                    Explanation

10 conc(           CAIL VV,(X)             Is the parent determinate?
                   MOVEI V,(X)             Discard parent's frame.
                   MOVEM A3,3(V)           Save argument 1 as local 1.
                   MOVEM A4,4(V)           Save argument 2 as local 2.
                   MOVEM A5,5(V)           Save argument 3 as local 3.
                   JSP C,%HSS1             Enter "head" routine.
                   HLRZM TR,R1             Take the current trail pointer,
                   HRLI R1,(V1)            pair it with the global stack pointer,
                   MOVEM R1,2(V)           and save the pair in the current frame.
                   MOVEM TR,$TR0           Remember the initial trail state.
 6 .(              HLRZM B,V               Extract frame pointer from arg 1.
                   CAIGE V,MOLS            Is arg 1 a molecule?
                   MOVE R2,0(B)            Get its functor.
                   JRST 1(C)               Exit "head" routine.
                   CAMN R2,list            Is the functor a list?
                   JRST clause             Goto the appropriate clause.
 3 X,              MOVE T,@1(B)            Get the first field of arg 1.
                   TLNN T,MSKMA            Is it a molecule or atomic?
                   MOVEM T,1(V1)           Store it in global variable 1.
 3 L1),            MOVE T,@2(B)            Get the second field of arg 1.
                   TLNN T,MSKMA            Is it a molecule or atomic?
                   MOVEM T,6(V)            Store it in local variable 4.
12 .(X,L3)         SETZM 2(V1)             Initialise global var 2 to unbound.
                   MOVE B,5(V)             Get arg 3.
                   JSP C,%USKV             Enter "output molecule" routine.
                   CAILE B,MAXREF          Is arg 3 a reference?
                   SKIPN R1,0(B)           Is the reference unbound?
                   JRST DEREF1             Proceed.
                   MOVE R1,0(C)            Get the skeleton address,
                   HRLI R1,(V1)            pair it with the global stack pointer,
                   MOVEM R1,0(B)           and store the resulting molecule.
                   CAIL R,GMARG(V1)        Is the reference a global?
                   CAIGE B,(VV1)           Is it after the last choice point?
                   JRST 1(C)               Exit "output molecule" routine.
[illegible]):-     JSP C,$NECK             Enter "neck" routine.
                   HRRM FL,0(V)            Save the addr of alternative clauses.
                   MOVEI R1,^0400000       Mark the presence of a global frame
                   ANDCAM R1,2(V)             in the current environment.
                   MOVEI X,(V)             Set parent's local frame.
                   MOVEI X1,(V1)           Set parent's global frame.
                   MOVEI V,@0(C)           Allocate local frame.
                   MOVEI V1,@1(C)          Allocate global frame.
                   SETOM -1(V1)            Mark end of global frame for GC.
                   CAMLE V,$VMAX           Is the local stack not full?
                   CAMLE V1,$V1MAX         Is the global stack not full?
                   JRST 2(C)               Exit "neck" routine.
 1           L3)   MOVEI A5,2(X1)          Arg 3 is a ref to global 2.
 1         L2,     MOVE A4,4(X)            Arg 2 is the value of local 2.
 1       L1,       MOVE A3,6(X)            Arg 1 is the value of local 4.
 1 conc(           JRST @conc+1            Goto the concatenate procedure.
50 .
```

COMMENTS ON THE PERSONAL SEQUENTIAL INFERENCE MACHINE, PSI

David Warren, February 1983

My overall impression of Psi is that it is a well thought-out design. I think it was the right to decision to stick to well-proven techniques, in order to design a machine which can be realised quickly. The planned performance of 20-30 K lips, and memory of 1 M words or more, should be more than adequate for a personal machine. It will be good to have a personal Prolog machine with a power somewhat in excess of a DEC 2060. I'd like to have one!

The absence of virtual memory should not be too much of a drawback, provided one can afford an adequate amount of physical memory. The DEC-10 address space of 256 K words is ample for most Prolog programs today, and in general rather more information will be packed into the Psi word than the DEC-10 word. Of course a lot depends on how big the Psi operating sytem will be.

The main area where I think there is room for improvement in the current Psi design concerns tail recursion optimisation (TRO). The form of TRO currently envisaged is more limited than that implemented in DEC-10 Prolog, and will miss important cases where TRO is really needed.

In DEC-10 Prolog, TRO applies to the last call of every clause, provided only that the calling procedure contains no backtrack points (which is simple to check at run-time). It is NOT necessary that the callee be determinate, i.e., TRO is effective even if the callee has more than one potentially matching clause. To implement this more general form of TRO requires that the arguments of every call be copied into registers and then deposited into the callee's stack frame. In cases where TRO is applicable, the callee's stack frame will overwrite the caller's stack frame.

The DEC-10 form of TRO would seem to be particularly advantageous for a machine like Psi. More items can be kept in hardware registers without needing to be saved in memory. In particular, it should be possible to avoid saving the contents of Psi's frame buffer in many cases. Also the cache will be more effective, since accesses to the local and control stacks will be kept more localised in memory. Also, for certain styles of programming (such as is used in concurrent Prolog), general TRO will be essential to conserve space on the local and control stacks, and to permit garbage collection to properly reclaim space on the global stack.

As presently designed, the Psi machine code does not directly support indexing of clauses. I think some indexing mechanism will be essential for many applications, and for much of the basic Psi software (where clause indexing will play the role of a case statement). In general, the machine design assumes that code is static and does not need to be frequently updated. There will clearly be a need to provide for more dynamic asserting and retracting of clauses, which implies some other form of clause storage. A distiction like that between DEC-10 Prolog's compiled and interpreted code will emerge, and this is probably a reasonable thing.

The KL0 machine language provides ample hooks and low-level primitives to cover all eventualities. It should be easy to fall back on conventional techniques if the high-level language framework proves too constrictive for implementing basic software. I wonder, though, whether quite so many primitives are needed. From the ordinary user's point of view, the 'bind_hook' predicate (which seems to be identical to Colmerauer's "freeze") will be a particularly welcome addition. I fear, though, that it will complicate the basic machinery of Psi.

To conclude, the design of the basic Psi machine looks very solid. The big challenge is going to be implementing all the software that will be needed to make it a really usable personal machine. I have not yet got a very clear picture of what the final user environment will be like.

PERFORMANCE EVALUATION OF PROLOG SYSTEMS

David Warren, February 1983

How can one fairly compare the performance of different Prolog systems? To some extent it is an impossible goal, since it all depends on what Prolog task one is interested in. The relative speeds of two systems will vary according to the task chosen for comparison. It is not really meaningful to say one system is X times faster than another, without further qualification. That having been said, it is nevertheless useful to agree on terminology and standards for measuring Prolog performance.

A term which has captured the popular imagination is "lips", standing for "logical inferences per second". This term obviously parallels "mips" ("millions of instructions per second"), which is a rough and ready, but useful, unit for measuring conventional machine performance. There is no [illegible]ndard machine instruction" that is being measured here. I[illegible] practice, "mips" is just a measure of the fastest rate at which a machine can execute instructions, and the instructions timed are simply whatever that machine can execute fastest! It seems reasonable to follow the same line with "lips". All that needs to be made precise is what is meant by "one logical inference". I take this to mean "one (successful) resolution". I would only include resolutions against user-defined clauses; calls to built-in predicates should be discounted. So the Prolog system builder should feel free to pick whatever example of Prolog his system performs resolutions fastest on. However, maybe resolutions over nullary predicates should be disqualified! The example which I personally use for measuring lips is list concatenation.

Apart from crude measurements of lips, there is a need for recognised benchmarks for Prolog. The benchmarks should exercise, to differing extents, the main facets of Prolog execution, namely:

- procedure call and return,
- unification,
- backtracking.

Testing of the commoner standard built-in procedures, especially arithmetic, should also be covered, in separate benchmarks. The benchmarks used in my PhD thesis (and included in the "Implementing Prolog" report) could form a starting point, although I think it would be a good thing to make a more deliberate and systematic attempt to cover a range of different types of Prolog program. For instance, the example that was intended to exercise backtracking (the geographical database query) is probably more a test of arithmetic capability.

For some Prolog systems, the precise ordering of clauses and/or placement of cuts can make a significant difference to performance. It is difficult to know how to properly smooth out these effects. In general, more reliable comparisons are obtained if all solutions are enumerated and Many testers of Prolog performance prefer to time examples of "real" programs; for example my own programs Warplan and Chat have been used for this purpose. This view has a lot to recommend it, but, with large programs, it is often difficult to be certain that exactly the same code is being tested on different systems.

Whatever benchmarks are chosen, extreme care has to be taken to exclude irrelevant factors when making the timings, and to make sure that what one is trying to measure is not swamped by other phenomena. Particular care needs to be taken when "scaling up" a short benchmark to make it last a measurable length of time. Factors which, if not properly allowed for, can easily distort performance measurements of Prolog include:

- time to read in a program or command,
- time to output results,
- time for garbage collection, stack shifts, etc.

Failure to allow for these factors has resulted in at least [illegible] sets of bogus Prolog performance measurements appearing in print (papers by Moss, and by Gutierrez).

DAVID H. D. WARREN

Computer Scientist
Artificial Intelligence Center
Computer Science and Technology Division
SRI International, Menlo Park, CA 94025   tel: (415) 859-2128

SPECIALIZED PROFESSIONAL COMPETENCE

Logic programming--the use and implementation of Prolog; natural language question answering; plan generation and program synthesis

PROFESSIONAL EXPERIENCE

Research fellow, Department of Artificial Intelligence, University of Edinburgh, 1975-81: research on planning, natural language and logic programming; author of the DEC-10 Prolog compiler/interpreter

Programmer, International Computers Limited, 1971-72: compiler implementation

Programmer, IBM United Kingdom Laboratories, 1969-70: PL/1 language definition

ACADEMIC BACKGROUND

B.A. (1969), Mathematics, Cambridge University

Ph.D. (1977), Artificial Intelligence, University of Edinburgh

PUBLICATIONS

"Issues in natural language access to databases from a logic programming perspective," Proc. of the 20th Meeting of the Assoc. for Computational Linguistics, Toronto, Canada (June 1982)

"Efficient processing of interactive relational database queries expressed in logic," Proc. of the Seventh International Conference on Very Large Data Bases, Cannes, France (September 1981)

Coauthor, "Definite clause grammars for language analysis--a survey of the formalism and a comparison with augmented transition networks," Artificial Intelligence 13, pp. 231-278 (1980)

"Logic programming and compiler writing," Software Practice & Experience 10, pp. 97-125 (1980)

"Prolog on the DECsystem-10," in Expert Systems in the Micro-Electronic Age, ed. D. Michie, Edinburgh University Press (1979)

Coauthor, "Prolog--the language and its implementation compared with LISP," Proc. of the ACM Symposium on AI and Programming Languages, Rochester, New York (August 1977)

Applied Logic, Ph.D. thesis, University of Edinburgh (1977)

"Generating conditional plans and programs," Proc. of the AISB Summer Conference, Edinburgh, Scotland (July 1976).

PROFESSIONAL ASSOCIATIONS AND HONORS

Association for Computational Linguistics

Open scholarship to Churchill College, Cambridge