Initial Assessment of Architectures for Production Systems

Charles Forgy[1]
Anoop Gupta[1]
Allen Newell[1]
Robert Wedig[2]
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Although production systems are appropriate for many applications in the artificial intelligence and expert systems areas, there are applications for which they are not fast enough to be used. If they are to be used for very large problems with severe time constraints, speed increases are essential. Recognizing that substantial further increases are not likely to be achieved through software techniques, the PSM project has begun investigating the use of hardware support for production system interpreters. The first task undertaken in the project was to attempt to understand the space of architectural possibilities and the trade-offs involved. This article presents the initial findings of the project. Briefly, the preliminary results indicate that the most attractive architecture for production systems is a machine containing a small number of very simple and very fast processors.

1. Introduction

Forward-chaining production systems are used extensively in artificial intelligence today. They are especially popular for use in the construction of knowledge-based expert systems [9, 11, 13, 14, 17]. Unfortunately, production systems are rather slow compared to more conventional programming languages. Consequently some computationally intensive tasks that are otherwise suitable for these systems cannot be implemented as production systems. The Production System Machine (PSM) project was created to develop hardware solutions to this problem. The first goal of the project is to understand the space of architectural possibilities for the PSM and the trade-offs involved. This article describes the initial results of the studies performed by the PSM project.

The rest of the paper consists of the following sections. Section 2 provides a brief description of the OPS production systems considered by the PSM project and includes a description of the Rete algorithm that is used to implement them. The Rete algorithm forms the basis for much of the later work. Section 3 elaborates on the need for hardware for production systems. It explains why we do not expect substantial further speed-ups from software techniques. Section 4 presents the results of measurements of some existing production system programs. The measurements enable us to explore the possibility of using parallelism in executing production system programs. Sections 5, 6, and 7 discuss three methods for speeding up the execution of production systems. Section 5 considers the role of parallelism, Section 6 considers processor architectures, and Section 7 considers hardware technology issues. The conclusions are presented in Section 8.

[1]With the Department of Computer Science.

[2]With the Department of Electrical and Computer Engineering.

2. Background

The PSM project is concerned with the OPS family of production systems [2, 4, 6]. These languages are for writing pure forward-chaining systems. An OPS program consists of a collection of *production rules* (or just "productions") and a global data base called *working memory*. Each production has a left-hand side which is a logical expression and a right-hand side consisting of zero or more executable statements. The logical expression in the left-hand side is composed of one or more *conditions*. A condition is a pattern; the left-hand side of a production is considered satisfied when every condition matches an element in working memory. The OPS interpreter executes a program by performing the following cycle of operations:

1. **Match:** The left-hand sides of all the productions are matched against the contents of working memory. The set of satisfied productions is called the conflict set.
2. **Conflict Resolution:** One of the satisfied productions is selected from the conflict set. If the conflict set is empty, the execution halts.
3. **Act:** The statements in the selected production's right-hand side are executed. The execution of these statements usually results in changes to the working memory. At the end of this step, the match step is executed again.

In this paper we are primarily concerned with speeding up the match operation. This is because the match operation takes most of the run time of interpreters that are implemented in software on uniprocessors. Moreover, when OPS is run on a parallel machine (which the PSM will be) the three operations can be pipelined, and much of the time required for conflict resolution and act can be overlapped with the time taken for the match. The total run time will consist of the time for the match plus a small amount of start-up time for the other two operations.

The algorithm that will be used in the production system machine is the Rete match algorithm [1, 3]. This algorithm has been used with variations in all the software implementations of OPS. It exploits two basic properties of OPS production systems to reduce the amount of processing required in the match:

- **The slow rate of change of working memory.** It is common for working memory to contain from a few hundred to over a thousand elements. Typically, executing a production results in two to four of the elements being changed. Thus on each cycle of the system, the vast majority of the information that the matcher needs is identical to the information it used on the previous cycle. Rete matchers take advantage of this by saving state between cycles.
- **The similarities among the left-hand sides.** The left-hand sides of productions in a program always contain many common subexpressions. Rete attempts to locate the

common subexpressions, so that at run-time the matcher can evaluate each of these expressions only once.

The Rete interpreter processes the left-hand sides of the productions prior to executing the system. It compiles the left-hand sides into a network that specifies the computations that the matcher has to perform in order to effect the mapping from changes in working memory to changes in the conflict set. The network is a dataflow graph. The input to the network consists of changes to working memory encoded in data structures called tokens. Other tokens output from the network specify the changes that must be made to the conflict set. As the tokens flow through the network, they activate the nodes, causing them to perform the necessary operations, creating new tokens that pass on to subsequent nodes in the network. The network contains essentially four kinds of nodes:

- **Constant-test nodes**: These nodes test constant features of working memory elements. They effectively implement a sorting network and process each element added to or deleted from working memory to determine which conditions the element matches.
- **Memory nodes**: These nodes maintain the matcher's state. They store lists of tokens that match individual conditions or groups of conditions.
- **Two-input nodes**: These nodes access the information stored by the memory nodes to determine whether groups of conditions are satisfied. For example, a two-input node might access the lists of tokens that have been determined to match two conditions of some production individually and determine whether there are any pairs of tokens that match the two conditions together. In general, not all pairs will match because the left-hand side may specify constraints such as consistency of variable bindings that have to hold between the two conditions. When a two-input node finds two tokens that match simultaneously, it builds a larger token indicating that fact and passes it to subsequent nodes in the network.
- **Terminal nodes**: Terminal nodes are concerned with changes to the conflict set. When one of these nodes is activated, it adds a production to or removes a production from the conflict set. The processing performed by the other nodes insures that these nodes are activated only when conflict set changes are required.

3. The Need for Hardware

The previous work on the efficiency of OPS systems has concentrated on software techniques. Over the past several years, improvements in the software have brought about substantial speed increases. The first LISP-based version of OPS was OPS2, which was implemented in 1978 [5]. The widely-used LISP version OPS5 was implemented about 1980 [2]. The improvements in software technology during that time made OPS5 at least five to ten times faster than OPS2. OPS5/LISP has been followed by two major reimplementations: an interpreter for OPS5 written in BLISS (a systems programming language) and the OPS83 interpreter [6]. OPS5/BLISS is at least six times faster than OPS5/LISP, and OPS83 is at least four times faster than OPS5/BLISS.[3] The speed-up from OPS2 to OPS5/BLISS resulted from a number of factors, including changing the representations of the important data structures and putting in special code to handle common cases efficiently. The additional speed-up of OPS83 resulted primarily from a new method of compiling left-hand sides. In all earlier versions of OPS, the left-hand sides were compiled into an intermediate representation that had to be interpreted at run time; in OPS83, the left-hand sides are compiled into native machine code.

It appears that with the advent of OPS83, further substantial improvements in software techniques have become difficult to achieve. Some amount of optimization of the compiled code is certainly possible, but this is expected to result in rather small increases in speed compared to what has occurred in recent years. The code that the OPS83 compiler produces is fairly good already. A factor of two speed-up due to compiler optimizations might be achieved; a factor of five seems unlikely at this time. Since the importance of achieving further speed increases for OPS is so clearly indicated, we feel that it is essential to investigate hardware support for OPS interpreters.[4]

4. Measurements of Production Systems

One of the first tasks undertaken by the PSM group was to perform extensive measurements of production systems running in OPS5. These measurements were necessary to evaluate the possibilities for speeding up Rete interpreters. Six systems were measured: R1 [13], a program for configuring VAX computer systems; XSEL [14], a program which acts as a sales assistant for VAX computer systems; PTRANS [9], a program for factory management; HAUNT, an adventure-game program developed by John Laird; DAA [11], a program for VLSI design; and SOAR [12], an experimental problem-solving architecture implemented as a production system. The R1, XSEL, and PTRANS programs were chosen because they are three of the largest production systems ever written, and because they are actually being used as expert systems in industry. The DAA program was chosen because it represents a computation-intensive task compared to the knowledge-intensive tasks performed by the previous programs. The SOAR program was chosen because it embodies a new paradigm for the use of production systems. Altogether, the six programs represent a wide spectrum of applications and programming styles. The systems contain from 100 to 2000 productions and from 50 to 1000 working memory elements. A few of the more important results are presented here; more detailed results can be found in [7].

The first set of measurements concern the surface characteristics of production system programs—that is, the characteristics of the programs that can be described without reference to the implementation techniques used in the interpreter. Table 1 shows the results. The first line gives the number of productions in each of the measured programs.[5] The second line gives the average number of conditions per production. The number of conditions in a production affects the complexity of the match for that production. The third line gives the average number of actions per production. The number of actions determines how much working memory is changed when a typical production fires. Together these numbers give an indication of the size and complexity of the productions in the systems. They show that productions are typically simple, containing neither large numbers of conditions nor large numbers of actions.

Feature	R1	XSEL	PTRANS	HAUNT	DAA	SOAR
1. Productions	1932	1443	1016	834	131	103
2. Conds/Prod	5.6	3.8	3.1	2.4	3.9	5.8
3. Actions/Prod	2.9	2.4	3.6	2.5	2.9	1.8

Table 1: Summary of Surface Measurements

[3] In absolute terms, a large production system with a large working memory and moderately complex left-hand sides (e.g., R1 [13]) might be expected to run at a rate of one to two production firings per second with OPS5/LISP running on a VAX 11/780; at a rate of six to twelve firings per second with OPS5/BLISS; and a rate of twenty-five to fifty firings per second with OPS83.

[4] The DADO project at Columbia University is also investigating hardware support for production systems [8, 18].

[5] In some cases only a subset of the complete production system program was measured because of problems with the LISP garbage collector. The numbers given in the table indicate the number of productions in the subset of the program that was measured.

The second set of measurements relate to the run-time activity of the OPS5 interpreter. Table 2 shows how many nodes are activated on average after each change to working memory. Line 1 shows the number of constant-test nodes activated. Although constant-test node activations constitute a large fraction (65%) of the total node activations, only a small fraction (10% to 30%) of the total match time is spent in processing them. This is because the processing associated with constant-test nodes is very simple compared to the memory nodes and the two-input nodes. Line 2 shows the number of memory nodes activated, and Line 3 the number of two-input nodes. Most of the matcher's time is spent evaluating these two kinds of nodes. Line 4 shows the number of terminal nodes activated. Since these numbers are small, updating the conflict set is a comparatively inexpensive operation. There are two major conclusions that can be drawn from this table. First, except for the constant-test nodes, the number of nodes activated is quite small. Second—and perhaps more significantly—except for the constant-test nodes, the numbers are essentially independent of the number of productions in the system.[6] This is important in the design of parallel production system interpreters (see the discussion of parallelism below).

Node Type	R1	XSEL	PTRANS	HAUNT	DAA	SOAR
1. Constant-test	136.3	105.3	122.1	88.5	35.9	26.5
2. Memory	12.3	8.7	10.7	12.5	4.0	11.1
3. Two-input	47.1	32.4	35.0	36.8	22.2	39.5
4. Terminal	1.0	1.7	1.7	1.5	2.0	4.0

Table 2: Node Activations per Working Memory Change

5. Parallelism

On the surface, the production system model of computation appears to admit a large amount of parallelism. This is because it is possible to perform match for all productions in parallel. Even after the left-hand sides have been compiled into a Rete network, the task still appears to admit a large amount of parallelism, because different paths through the network can be processed in parallel. It is our current assessment, however, that the speed-up available from parallelism in production systems is much smaller than it initially appears.

We are exploring three sources of parallelism for the match step in production system programs: production-level, condition-level, and action-level parallelism. In the following paragraphs we briefly describe each of these three sources, and where possible give the speed-up that we expect from that source.

5.1. Production-level Parallelism

In production-level parallelism, the productions in the system are divided into several groups and a separate process is constructed to perform match for each group. All the processes can execute in parallel. The extreme case for production-level parallelism is when the match for each production is performed in parallel. The major advantage of production-level parallelism is that no communication is required between the processes performing the match, although the changes to working memory must be communicated to all processes. Since the communications requirements are very limited, both shared memory and non-shared memory multiprocessor architectures can exploit production-level parallelism.

The measurements described in Section 4 are useful in determining the amount of speed-up that is potentially available from production-level parallelism. Line 3 of Table 2 shows that on average, each change to working memory causes about thirty-five two-input nodes to be activated. Since the sharing of nodes at this level of the network is limited, the number of two-input nodes activated is approximately equal to the number of productions containing conditions that match the working memory element. Thus, on average, when an element is added to or deleted from working memory, the stored state for thirty-five productions must be updated.[7] The number of affected productions is significant because most of the match time is devoted to these productions. Thus the immediately apparent upper bound to the amount of speed-up from production-level parallelism is around thirty-five. However, it is easy to see that this is a very optimistic upper bound. Measurements show that it is common for a few of the affected productions to require five or more times as much processing as the average production. Thus in a machine that uses substantial amounts of production-level parallelism, the match would be characterized by a brief flurry of parallel activity followed by a long period when only a few processors are busy. The average concurrency would be much lower than the peak concurrency.

5.2. Condition-level Parallelism

In condition-level parallelism, the match for each condition in the left-hand side of a production is handled by a separate process. Condition-level parallelism involves more communication overhead than production-level parallelism. It is now necessary to communicate tokens matching one condition to processes that combine tokens, thus forming new tokens matching several conditions in the left-hand side. This increased communication makes shared-memory multiprocessors preferable to non-shared memory multicomputers. The speed-up expected from condition-level parallelism is quite limited. This is because productions tend to be simple, as Table 1 shows. Since the typical production contains only three to six conditions, even when all the conditions in an left-hand side have to be processed (a rare occurrence) only three to six parallel processes can be run.

5.3. Action-level Parallelism

In action-level parallelism, all the changes to working memory that occur when a production fires are processed in parallel. Action-level parallelism does not require any more data communication overhead than the previous two sources of parallelism, but it does involve a substantial amount of extra synchronization overhead. The speed-up possible from action-level parallelism is also quite limited. A typical production makes two to four working memory changes, so the amount of action-level parallelism available is at most two to four.

5.4. Simulation Results

To gain a more detailed evaluation of the potential for parallelism in the interpreter, a simulator has been constructed, and simulations of the execution of the XSEL, PTRANS, and DAA expert systems have been run. The cost model assumed for the simulation is based on the costs that have been computed for the OPS83 matcher. Since the OPS83 matcher would have to be modified somewhat in order to run in parallel, the costs have been adjusted to take these modifications into account.

The graph in Figure 1 indicates the speed-up that is achieved through the use of production-level, condition-level, and action-level parallelism. As the graph shows, the speed-up obtained is quite limited. This is a combined effect of the facts that (1) the processors must wait for all affected productions to finish match before proceeding to the next cycle, and (2) there is a large variance in the computational requirements of the affected productions. The graphs show that a speed-up of four to six times can be obtained with relatively good processor utilization, but to obtain a larger factor requires much more hardware.

[6] There are known methods of reducing the effect of production system size on the number of constant-test node activations (see [1]).

[7] Note that the number thirty-five is independent of the number of productions in the program. An intuitive explanation for this is that programmers divide problems into subproblems, and at any given time the program execution corresponds to solving only one of these subproblems. The size of the subproblems is independent of the size of the overall problem and primarily depends on the complexity that an individual can deal with at the same time.

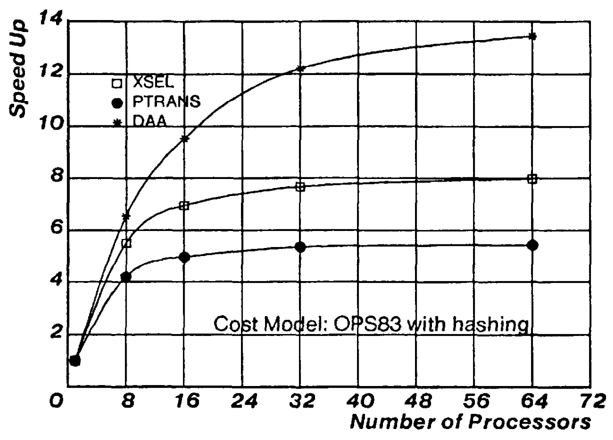

Figure 1: Parallelism in Production Systems

6. Processor Architecture

Because of our experience with the Rete network, we have a good idea of how a machine executing OPS will behave. In the Rete network, there are only a few different types of code sequences to deal with. By calculating the time that a given processor requires to execute these sequences, we can accurately determine how effective the processor is for this task. Typical code sequences from the Rete network are shown in Figures 2 and 3. Figure 2 shows the computation performed by a constant-test node. Figure 3 shows a loop from a two-input node. The loop is executed when the two-input node compares a token from one memory with the tokens in another memory.

```
load  R1,"active"          ;load the constant
cmp   R1,1(R.CurWme)       ;compare the value
jne   L1                   ;if not equal, fail
```

Figure 2: Assembly Code for a Constant Test

```
     move  R0,R.MPtr1                 ;test memory pointer
     jeq   L2                         ;exit if nil
10$: load  R.Wme1,WME(R.MPtr1)        ;get the wme
     jsb   L3                         ;goto tests
     load  R.MPtr1,NEXT(R.MPtr1)      ;get next token
     jne   10$                        ;continue if not nil
     jmp   L2                         ;exit
```

Figure 3: Loop from a Two-input Node

As these code sequences illustrate, the computations performed by the matcher are primarily memory bound and highly sequential. Each instruction's execution depends on the previous one's, leaving little room for concurrent execution of the instructions. Consequently, it is not advantageous to develop a processor with multiple functional units able to extract concurrency and simultaneously execute multiple instructions. It is also not worthwhile to design a computer with a large range of complex instructions and addressing modes since the majority of time is spent executing simple operations. We conclude that a machine for executing production systems should have a simple instruction set and should execute the instructions in as few clock cycles as possible. The processor designs that best satisfy these requirements are the reduced instruction set (RISC) machines such as the Berkeley RISC [15], the Stanford MIPS [10], or the IBM 801 [16]. Such a machine could execute most instructions in two machine cycles. We estimate that a complex instruction set machine requires four to eight cycles per instruction, making the simple machine two to four times faster.

7. Device Technology

Since the correct choice for the machine appears to be a RISC-like processor and rather modest levels of parallelism, we are exploring the use of high-speed logic families in its implementation, such as ECL or GaAs. The difficulties inherent in the use of these technologies are offset to a large degree by the fact that the machine will use relatively little hardware. Certainly designing each component will be more difficult than designing a similar component in TTL or MOS; however the machine will be fairly simple so the total design time will not be excessive. In addition, while the processors will be more expensive than processors implemented in slower technologies, the machine will not contain large numbers of them, and the total cost will not be excessive. We estimate an ECL implementation of the machine would be about four times faster than a TTL implementation, provided the processor did not spend too much time waiting on memory.

8. Conclusions

The PSM project is investigating the use of hardware support for production system interpreters. We expect to obtain speed increases from three sources: parallelism, processor architecture, and device technology. Our studies are not complete, but some initial results are available:

- Parallelism: The task admits a modest amount of parallelism. We expect parallelism to contribute a 5 to 10 fold increase in speed.
- Processor architecture: The most attractive architectures for this task are the simple (or so-called RISC) processors. We estimate that a RISC machine would be 2 to 4 times faster than a complex instruction set machine.
- Device technology: Since speed is of paramount importance in this task, and since very simple processors are appropriate, it will be advantageous to use high-speed device technologies. We estimate that using ECL would provide a factor of 4 increase in speed.

In summary then, a machine built along the lines we suggest would be between 5 * 2 * 4 = 40 and 10 * 4 * 4 = 160 times faster than a complex uniprocessor implemented in a slower speed technology. It should be emphasized that these are preliminary results, and are subject to change as the work proceeds.

9. Acknowledgments

H. T. Kung, John McDermott, and Kemal Oflazer contributed substantially to this research.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-00450.

References

1. Forgy, C. L. *On the Efficient Implementations of Production Systems.* Ph.D. Th., Carnegie-Mellon University, 1979.
2. Forgy, C. L. OPS5 User's Manual. Tech. Rept. CMU-CS-81-135, Carnegie-Mellon University, 1981.
3. Forgy, C. L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem." *Artificial Intelligence 19* (September 1982).
4. Forgy, C. L. and McDermott, J. OPS, A Domain-Independent Production System. International Joint Conference on Artificial Intelligence, IJCAI-77.
5. Forgy, C. L. and McDermott, J. The OPS2 Reference Manual. Department of Computer Science, Carnegie-Mellon University, 1978.
6. Forgy, C. L. The OPS83 Report. Department of Computer Science, Carnegie-Mellon University, May 1984.

7. Gupta, A. and Forgy, C. L. Measurements on Production Systems. Carnegie-Mellon University, 1983.
8. Gupta, A. Implementing OPS5 Production Systems on DADO. International Conference on Parallel Processing, August, 1984.
9. Haley, P., Kowalski, J., McDermott, J., and McWhorter, R. PTRANS: A Rule-Based Management Assistant. In preparation, Carnegie-Mellon University
10. Hennessy, J. L., et al. The MIPS Machine. Digest of Papers from the Computer Conference, Spring 82, February, 1982, pp. 2-7.
11. Kowalski, T. and Thomas, D. The VLSI Design Automation Assistant: Prototype System. Proceedings of the 20th Design Automation Conference, ACM and IEEE, June, 1983.
12. Laird, J. and Newell, A. A Universal Weak Method: Summary of Results. International Joint Conference on Artificial Intelligence, IJCAI-83.
13. McDermott, J. R1: A Rule-based Configurer of Computer Systems. Tech. Rept. CMU-CS-80-119, Carnegie-Mellon University, April, 1980.
14. McDermott, J. XSEL: A Computer Salesperson's Assistant. In *Machine Intelligence*, J.E. Hayes, D. Michie, and Y.H. Pao, Ed.,Horwood, 1982.
15. Patterson, D. A. and Sequin, C. H. "A VLSI RISC." *Computer 9* (1982).
16. Radin, G. "The 801 Minicomputer." *IBM Journal of Research and Development 27* (May 1983).
17. Stolfo, S. J. and Vesonder, G. T. ACE: An Expert System Supporting Analysis and Management Decision Making. Department of Computer Science, Columbia University, 1982.
18. Stolfo, S. J. and Shaw, D. E. DADO: A Tree-Structured Machine Architecture for Production Systems. National Conference on Artificial Intelligence, AAAI-1982.