CUCS-340-88

An Algorithmic Taxonomy of Production System Machines

Russell C. Mills

Columbia University
Computer Science Department

29 April 1988

Abstract

This paper presents a survey of computer architectures designed to execute production systems. After a brief description of production systems and production system languages, the paper summarizes match algorithms, particularly the Rete algorithm, and outlines suggested parallelizations. Most parallel production system algorithms have as their unit of sequential computation a single production's left-hand side, activations of a single Rete node, a single activation of a Rete node, or a single comparison in a Rete node. The paper discusses a number of proposed production system machine architectures in terms of the parallel and sequential computations performed in the algorithms suggested for each machine. A taxonomy of parallel production system algorithms, describing in detail the distribution and replication of data and computations, concludes the paper.

1 Introduction

The production system paradigm is a data-directed formalism widely used in artificial intelligence research and in the building of expert systems. A number of commercially successful expert systems such as XCON at Digital Equipment Corporation [29], and ACE at AT&T [66] have been implemented in production system languages. The slow execution speeds of production systems (XCON requires five minutes of CPU time on a fairly powerful computer to configure a single VAX system, while ACE implemented in OPS4 requires many hours of CPU time to process a single day's data on telephone cable failure reports for a small city) stand in the way of constructing larger or real-time production systems. Consequently, many researchers have attempted to accelerate production system execution through hardware, software, and algorithmic techniques. This paper discusses some proposed production system machine architectures and proposes a taxonomy based on the algorithm(s) proposed for each.

1.1 Production Systems

A *production system*, or PS [37, 5] is a pattern- or data-directed program expressed as a set of *production rules*, known collectively as the *production memory*, or PM, which operates on a global database, the *working memory*, or WM, under the direction of a *control strategy*. Each production consists of an if-then rule whose left-hand side (LHS) consists of a precondition on the database for application of the rule, and whose right-hand side (RHS) consists of a set of actions, any of which may affect the database. The control strategy dictates the order of application, or *firing*, of rules whose preconditions are satisfied. A rule whose LHS is satisfied, together with the set of WM elements (WME's) that satisfies it, is known as an *instantiation* of the rule.

A typical production system control strategy dictates that the system repeatedly execute execution steps consisting of three parts:

1. *Match*: Determine the set of all rule instantiations, which is known as the *conflict set*.
2. *Select*: Choose a subset of the instantiations to fire. In the most widely-used production system languages, this process, known as *conflict resolution*, selects exactly one instantiation. If there are no rule instantiations, the system halts.
3. *Act*: Perform the actions indicated by the RHS's of the selected instantiated production(s).

Production systems are typically but not always forward-chaining inference systems.

1.2 Production Systems as a Basic Computational Paradigm for AI

Production systems are widely used in AI research because they are data-driven and separate program operations from program control [39]. Many problems in AI can be reduced to production systems.

Nils Nilsson [39] describes the reduction of theorem-proving and state-space search to generalized production systems. His reduction makes PM the set of state-generating operators and WM the set of states being explored.

Michael Rychener [53] shows how semantic nets [46] can be expressed in production system terms, and constructs an expert system for computer-aided design. In his system, PM consists of two classes of

rules: those that create and traverse the semantic net, and those that encode the net itself, while WM consists of goals and temporary network structures.

Mark Perlin [45] has proved the equivalence of frame systems [30] and production systems. First he demonstrates that a frame system can simulate a production system at a cost that is at most linear in the size of the rule system by constructing, for a given production system, a frame system that performs the matching operations for the productions' LHS's. Next he shows that a production system can simulate a frame system, again at a cost that is at most linear. For each *active memory* operation such as *read* and *write*, and for each *local propagation* schedule, he constructs a production that implements the operation.

1.3 Production System Languages

In a typical production system, rules are of the form

P: C_1 & ... & C_n --> A_1 ... A_m,

where each C_j is a condition (known as a *condition element*, or CE), and each A_k is an action, which may add or delete WM elements, or perhaps interact with a user or perform input or output on a file system. Each condition element is a representation of a class of WM elements that match the condition-- each production system language defines the form of the representation and what it means to match a condition element. Condition elements can contain variables, and different CE's can refer to the same variable; when a CE matches a WME, variables in the CE are bound to the corresponding features of the WME. In most production system languages, condition elements can also be negated. If the above production P contains p positive (not negated) CE's, an instantiation of P consists of P together with a p-tuple of WME's (w_1,...,w_p) such that

1. each w_i satisfies the i-th postive condition element,

2. all variables in the positive CE's are bound consistently,

3. and for each negated CE, there is no WME satisfying it with variable bindings consistent with those in the positive CE's.

The OPS family of production system languages [43], developed at Carnegie-Mellon University, display a variety of representations of working memory. In each OPS language, condition elements are abstractions of working memory elements. Each language provides a variety of predicates for matching CE's to WME's.

One of the earlier members of the OPS family, OPS4 [7], represents WME's as arbitrary list structures. OPS4 provides predicates for matching list structures, but also allows programmers to write arbitrary LISP functions to match CE's and WME's.

In contrast, OPS5 [3, 10], probably the most widely used OPS language, represents a WM element as a tuple of (named attribute, constant value) pairs. An OPS5 condition element is an abstraction of a WM element; it contains a number of attribute fields, each of which can contain variables, constants, and predicate symbols testing equality, ordering, and equality of type. Each OPS5 WME or CE also belongs to exactly one *class*, which is just a name-valued attribute that itself has no name, and occupies the first position in that WME or CE. OPS5 variable names are enclosed in angle brackets ("<name>"), while attribute names begin with a caret ('^'). Figure 1 displays a contrived OPS5 production and two WME's that together constitute an instantiation of the rule.

Figure 1: An OPS5 Production

```
(p example
   (class1 ^attr1 <x> > 1  ^attr2 <x>)
   (class2 ^attr1 <z> < <x>)
   -->
   (remove 1)
   (make (class3 ^attr1 <z>)))

(class1 ^attr1 5 attr2 5)
(class2 ^attr1 4)
```

OPS5's set of predicates is limited, and the language does not allow the programmer to define new ones. OPS83 [12] extends OPS5's pattern-matching capabilities by allowing the programmer to call functions in the LHS of productions. In fact, OPS83 is a procedural language with an embedded **element** data type (much like a standard record type), and **make**, **modify**, and **remove** statements to manipulate the contents of WM. Instead of providing a conflict-resolution strategy, OPS83 provides the **fire** statement, which fires a programmer-specified rule instantiation.

In spite of the limited expressiveness of OPS5, with its small number of pattern-matching operations and its lack of programmer-defined predicates, most proposed production system machines are designed specifically to execute OPS5. This situation may stem from the general availability of an OPS5 interpreter and the consequent widespread use of the language. But OPS5 may not be an ideal production system language, and not all production systems are written in OPS5, so OPS5-specific machine designs are open to criticism as being too inflexible.

1.4 Execution Speed of Uniprocessor PS Implementations

In the worst case, the problem of deciding whether a production's LHS can be satisfied is at least NP-complete, since a known NP-complete problem, conjunctive boolean query [4], can be recast in production system terms. Average-case execution speed is therefore of much more interest than worst-case speed. Since there is no model of an average production system, most measurements of production system execution speed are empirical studies of large working systems.

Charles Forgy has reported [8] that most production systems spend 90% of their execution time in the match phase, even using an efficient match algorithm. Most research on accelerating production system execution has therefore centered on the match phase. However, one should remember that if the match, select, and act phases are not overlapped, completely eliminating the time spent in the match phase can speed up execution only by a factor of 10.

Because different productions make different numbers of changes to WM, and because the number of WM changes determines the amount of match computation required, the execution speed of production system interpreters is usually measured in WM changes per second, rather than rule firings [60]. Measurements are usually made on a moderately powerful computer, typically a VAX-11/780. Execution speed varies widely and depends on the language, the implementation language of the interpreter, and the degree of compilation of the rules. For example, Anoop Gupta reports [17] that the standard OPS5 interpreter written in LISP executes 8 WM changes/second, while the Bliss-based interpreter executes about 40. OPS83, which is compiled into machine code rather than interpreted, runs at about 200 WM

changes/second. The order-of-magnitude difference in speed between interpreted code and compiled code complicates the comparison of performance projections for a number of production system machines discussed later in this paper.

1.5 Algorithms for Sequential Production System Execution

The problem of finding all rule instantiations can be recast in relational database terms [64]. As before, let the production

P: C_1 & ... & C_n --> A_1 ... A_m,

have p positive CE's C_{i_1},...,C_{i_p}. For each C_i, let R_i be the relation defined as the subset of WM satisfying C_i. Then the set of rule instantiations I(P) is that subset of $R_{i_1} \otimes ... \otimes R_{i_p}$, the *join* of the relations R_i on the variables common to two or more CE's in P, not excluded by a WME matching a negated CE with consistent bindings.

The most naive match algorithm for production systems constructs the conflict set during each match phase by comparing every production with every tuple of WM elements. However, since the number of changes to WM each cycle is typically smaller than WM itself (in fact, in OPS5 programs, much smaller [15]), more efficient match algorithms process the changes to WM to produce changes to the conflict set. These algorithms save some of the state of the match algorithm between production system cycles. Researchers have designed a spectrum of state-saving algorithms, described below, distinguished by the amount of state they save.

Daniel Miranker's TREAT algorithm [32, 33] saves only the relations R_i and the conflict set I(P). In each match cycle, it uses new WME's as seeds to construct new instantiations, and it limits the search for new instantiations to those productions all of whose positive CE's have a nonempty corresponding relation.

Charles Forgy's Rete algorithm [8, 11] saves all initial subsequence relations of P. For each i in 1,...,n, let P_i be the partial LHS

P_i : C_1 & ... & C_i

The Rete algorithm saves all the I(P_i), the set of all instantiations of P_i, in addition to the R_i. A change to WM, whether addition or deletion, that affects R_j can cause changes in all the P_i for i >= j. The Rete algorithm underlies the commercial OPS5 and OPS83 interpreters and compilers.

A variant of the Rete algorithm, the subject of some experiments by Anoop Gupta [18], recursively splits the LHS of each production in two, computes the set of instantiations of each half, and computes the join of the two sets of partial instantiations. Gupta's results indicated that on the OPS5 programs he studied, computing the partial instantiations of the right-hand half of each LHS wasted time and space.

A final variant of the Rete algorithm, proposed by Kemal Oflazer and reported in [Gupta86a], saves all nonredundant instantiations of all subsequences of the LHS of each production. Oflazer's algorithm underlies the design of a proposed machine for production systems, described later in this paper.

1.6 The Rete Match Algorithm In More Detail

Since many of the production system architectures discussed in this survey attempt to accelerate matching by parallelizing the Rete match algorithm, a more detailed discussion of the sequential Rete algorithm is warranted. The PS language compiler translates the LHS of each production into a dataflow network, which the runtime system interprets or executes. For each production P and each change to WM, the network computes the changes to $I(P_i)$ from the changes to R_i and to $I(P_{i-1})$. Each node in the network stores part of the saved match state as a set of *tokens*, each of which represents an element of R_i or $I(P_i)$, or performs part of the computation to update the state. Rete network nodes are of several types:

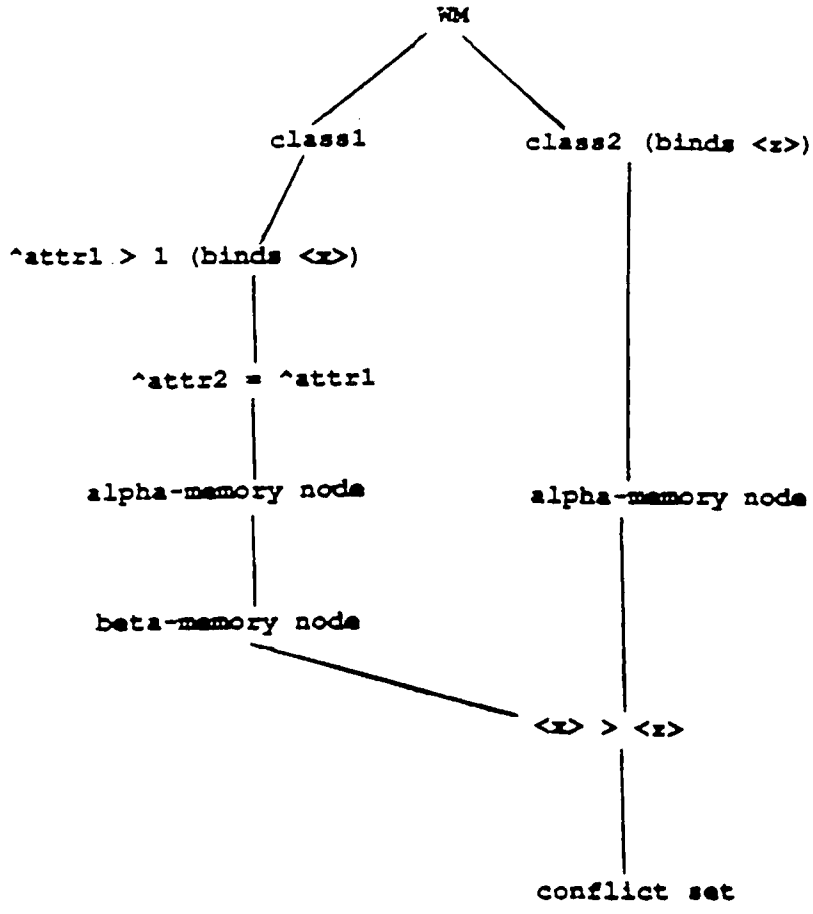
- *alpha-memory* nodes, each of which stores a relation R_i.
- *beta-memory* nodes, each of which stores a relation $I(P_i)$. The beta-memory node storing $I(P_n)$ is also known as an *output* node, and it stores part of the conflict set.
- *single-input-test* nodes, which perform tests on WME's for membership in a relation R_i. These tests include comparing a WME attribute against a constant and checking the consistency of variable bindings within a single CE.
- *two-input-test* nodes, which construct $I(P_i)$ from $I(P_{i-1})$ (the *right input*) and R_i (the *left input*). If the CE corresponding to the right input of a node is negated, the node is known as a not-node; otherwise it is called an and-node. These tests are also known as *inter-condition* tests.

Figure 2 shows the network constructed from the production given in figure 1. In practice, the compiler merges the alpha-memory node storing R_1 and the beta-memory node storing $I(P_1)$.

Rete alpha- and beta-memory nodes can also be shared. If two productions P and P' share a substring of CE's P_i and P'_i, the compiler can generate a network in which the tests involved in constructing $I(P_i)$ and $I(P'_i)$ are performed once, and a single beta-memory node storing $I(P_i)$ has two outputs. Notice that if the compiler does not generate shared nodes, it can merge each two-input-test node with its two memory-node inputs.

During production system execution, adding an element to WM initiates a sequence of one-input tests and the possible creation of an alpha-memory token, which *activates* the and- and not-nodes connected to the alpha-memory node. And-nodes receiving the new token compare it with the tokens stored in their other memory-node input, and create a new token for each new partial instantiation. Not-nodes receiving a new token from the right also compare it with the tokens stored in their left input. If this new token is the first token from the right input matching a particular token from the left, some previously-created partial instantiations must be retracted, so the not-node creates a *negated* token, which flows through the network annihilating matching positive tokens and creating new negated tokens. Deleting an element from WM causes the removal of all alpha-memory tokens corresponding to it, and the creation of negated tokens. A negated token arriving at the right input of a not-node causes the creation of positive tokens if the positive alpha-memory token corresponding to the newly-arrived negated token is the only one blocking the creation of partial instantiations. The sequential Rete match algorithm guarantees that a negated token arriving at a memory node finds a corresponding positive token.

Figure 2: A Rete network



1.7 Some Statistics on Production Systems

Anoop Gupta [15, 18] has collected extensive statistics on two sets of six production systems each. He measured the following properties of production systems:

- Textual characteristics such as the number of CE's per production, the number of negated CE's per production, and the number of WM changes per RHS. In his sample, 27% of all productions contained at least one negated CE, so negated CE's are important.
- Rete network characteristics such as the number of nodes per CE, for compilation with node sharing and without.
- Run-time characteristics such as the average and maximum number of Rete match tokens in beta-memory nodes, the percentage of and- and not-nodes that perform no test for variable equality, the number of node activations per WM change, and the number of productions whose saved state is changed per WM change. He found that on the average, each WM change affects about 26 productions in this manner.

Many of the designs discussed in this paper are based on these statistics.

1.8 Suggested Parallelizations of Production Systems

This paper divides proposed production system architectures into groups based on characteristics of the algorithms proposed for them. This division highlights important common features of the machines in each group, and the characteristics that distinguish the groups. The paper first discusses uniprocessor designs, because these designs are often used as building blocks in multiprocessor architectures. It then divides the parallel production system algorithms into classes based on which operations are parallelized.

The simplest parallelization technique performs the match operation for different rules in parallel. Several variants on this technique attempt to provide more speedup. Non-state-saving algorithms can be parallelized; one example is the distributed version of the TREAT algorithm described later in this paper. The Rete match can also be parallelized in a number of ways. A parallel Rete match algorithm can process sequentially-activated Rete network nodes in parallel, it can process sequentially nodes activated in parallel, or it can process in parallel nodes activated in parallel. Finally, the productions' RHS can be processed in parallel if the production system interpreter allows multiple rules to fire simultaneously.

2 Specialized Production System Uniprocessors

Several researchers have explored uniprocessor architectures specialized for production systems. Uniprocessor architectures are important in this context because fast uniprocessors set upper bounds on communication speed in parallel systems. These designs are also used as processing elements (PE's) in some of the parallel machines described in this paper.

2.1 RISC architectures for production systems

Theodore Lehr has proposed [26] a Reduced Instruction Set Computer (RISC) [44] architecture for production system execution. His RISCF processor implements the complete Rete match algorithm. He bases its design on several characteristics of OPS5 program execution. First, a processor executing the Rete match algorithm makes many references to memory, since inter-condition tests typically involve large numbers of tokens. Second, the processor executes arithmetic operations consisting mostly of integer comparisons. Finally, the processor's branching behavior is complex and erratic, as the assembly code generated from OPS5 programs contains a large number of conditional jumps and subroutine calls, which can slow down pipelined architectures and cause cache misses.

The proposed RISCF processor addresses the problems of heavy memory traffic and branching and has a simple ALU. The compare instructions, for example, compare a register with the contents of memory and set condition codes and branch prediction bits, and must precede a conditional branch. Statistics derived from Anoop Gupta's measurements of production systems show that the results of many of these tests can be predicted statically with 90% accuracy. The branch prediction bits then allow the processor to keep its instruction fetch/decode/execute pipeline full most of the time. Lehr projects an overall speedup of 1.15 from the branch prediction strategy. The RISCF processor, like other RISC's, also has a large register file.

Lehr has also designed a gallium arsenide realization of the RISCF processor [27]. This chip set comprises 9 chips, and has a projected execution rate of one instruction every 30-nanosecond machine cycle.

James Quinlan has studied the effectiveness of a number of uniprocessor architectures, including

Lehr's, in executing production systems [47]. Using Anoop Gupta's measurements on six production systems, Quinlan estimated the number of instruction fetches, data reads and writes, and computation instructions performed by six architectures, including a hypothesized microcoded OPS matcher, the RISCF processor, and a VAX-11/780. From these statistics he derived total number of machine cycles and execution times for the various architectures. He concluded that the microcoded machine should run three to six times as fast as a VAX and twice as fast as an NMOS realization of the RISCF processor, but that a gallium arsenide realization of the RISCF would be as fast as the microcoded machine provided it had an effective cache.

At the time of Quinlan's study, the VAX-11/780 represented technology that was almost ten years old. While the use of the VAX is legitimate for baseline comparisons, the age difference among the technologies he studied vitiates his conclusions about the effectiveness of special processor designs relative to conventional architectures.

3 Production-level parallelism

Production-level parallelism entails distributing entire productions among different processors. Each processor receives a different subset of the rules in a production system, and stores the initial working memory elements potentially matching its rules. During the match phase, all processors, using a suitable match algorithm, (perhaps naive match, TREAT, or Rete), match their rules against their subset of working memory. During the select phase, the processors cooperate to determine which rule gets fired in the act phase. Production-level parallelism can be combined with other levels of parallelism, as is seen in machines described in later sections.

The copy-and-constrain method [61] applied to a rule produces constrained copies of the rule, each of which matches a subset of the instantiations of the original rule. If the set of possible working memory elements is finite, then the set of possible instantiations of a rule not containing any negated condition elements is finite as well, and it is at least theoretically possible to create a constrained copy of the rule for each instantiation. Such a specialized rule contains no variables, and an instantiation can be found with just one constant test per condition element. At least two working production system interpreters work on this principle: the Aspro and the Concurrent Inference System.

3.1 Illiac-IV

Charles Forgy has studied the possiblity of interpreting production systems on the Illiac-IV [9]. The Illiac-IV is an 8-by-8 mesh-connected array of SIMD[1] PE's. Each PE consists of a powerful processor, a small local memory (2K bytes), connections to its four neighbors in the array, connections to a broadcast bus for instructions and data, and access to its own section of a disk. Forgy's algorithm is just an SIMD version of the Rete match algorithm in which the rules are distributed among the PE's. Each PE receives a number of productions; during the match phase, the host computer broadcasts changes to working memory and instructions to control the Rete network match in each PE. The algorithm does not use the mesh inter-PE connections; it simply uses the Illiac-IV as a set of processors on a broadcast bus.

In order to simplify the implementation, Forgy adopted a simple production system language, SPS, that

[1] Single Instruction, Multiple Data

represents data as triples, rather than as tuples of attribute-value or as lists. SPS's match strategy is not guaranteed to find all satisfied productions: starting with the most recently-added working memory elements, it attempts to build instantiations CE by CE from left to right in each production, binding variables as it proceeds. If at some point in the process, it cannot satisfy a CE, it abandons the production rather than backtracking. SPS has no conflict resolution strategy; the interpreter simply fires all satisfied productions on each cycle.

The SPS compiler divides the set of Rete network nodes into a number of different classes such as not-, and-, and single-input-test nodes, and allocates space for the input and output memories of each node. The no-backtracking strategy allows the compiler to determine the maximum size of the alpha- and beta-memories required. During each production system cycle, all PE's evaluate together all nodes of each type.

Forgy presents no performance projections or data for his SPS implementation. His parallel match algorithm differs from that used by other researchers on SIMD machines such as NON-VON and the CAP (see below) in that it performs the entire match operation in lockstep. In doing so, the algorithm abandons backtracking and sacrifices complete search.

3.2 Finding Optimal Partitions

In each PS execution cycle, the amount of match computation required varies tremendously among the rules in the system. Kemal Oflazer of CMU has studied the problem of assigning productions to processors at compile time to achieve run-time load balancing among the partitions [40]. His studies used both analyses of program texts and statistics derived from previous program executions to derive production system partitionings; simulations showed, however, that they were only slightly better than random ones.

Oflazer compared three partitioning methods to random partitioning. The first method assigned productions to processors in a round-robin fashion based on their textual order in a program. This assignment strategy is not random, since programmers are likely to put similar productions, or productions that work on similar patterns in working memory, close together. Round-robin assignment should therefore place similar productions, which are likely to require processing during the same interpreter cycles, into different processors, and contribute to load balancing. The second method used syntactic information. Many OPS5 productions have *goal* or *context* condition elements, whose purpose is to condition their activation to a phase of program execution. Syntactic assignment places productions with the same goal element into different partitions. The third technique used the processing time required by the Rete algorithm to maintain the state of each production and the frequency with which the algorithm processed productions together during the same match cycle to predict good partitionings. Finding optimal partitionings is an NP-complete bin-packing problem, so Oflazer used simulated annealing to search for approximations to the optimal partition.

Simulated executions of the partitioned production systems showed that no static partitioning scheme worked very well. The method based on execution time gave somewhat better partitionings than the two textual methods, but all three techniques gave speedups of 1.15 to 1.25 over random partitioning. One explanation for this poor performance is that production systems have irregular and very data-dependent execution paths, and that these paths are difficult to detect statically.

3.3 The Aspro System

The Aspro Parallel Inference Engine [49], developed by Goodyear Aerospace, is a specialized pattern-matching machine attached to a sequential processor. The Aspro consists of a 2K-element array of bit-serial PE's, each with 4K bits of memory. The Aspro represents the complete state of working memory as a single 2K-bit vector; it uses the same representation for the LHS and RHS of a production. During each match phase, the interpreter compares each production bitwise in turn with the current state of working memory. A production is satisfied if every bit set in its LHS (i.e. every condition) is also set in working memory. During the act phase, the interpreter fires every satisfied production; firing a production may include calling functions in the host.

Since each bitwise comparison of working memory with the LHS of a production is a single instruction, match time is linear in the number of productions. But since the Aspro memory can hold at most 2K productions, the system is guaranteed to execute at least 500 match cycles/second.

3.4 The Concurrent Inference System

The Concurrent Inference System (CIS) [1], developed at the MIT AI lab, is a forward- and backward-chaining inference engine. It has been implemented on the Connection Machine [20], a massively parallel highly-connected SIMD array of very small processors.

The CIS represents rules in a standard if-then form, but each conclusion has associated with it a real-valued certainty factor. Variables in rules are allowed, provided that the set of possible values is finite and known at compile time; the compiler creates an equivalent set of constrained rules containing no variables. Since the compiler knows each variable's set of possible values, it can compile the set of rules into a fixed graph.

At run time, each rule is associated with an activity factor. Inference proceeds synchronously as an activity network. During each inference step, each rule computes the minimum or maximum activity, for conjunctions and disjunctions respectively, of its LHS. It then adjusts the activity of each of its conclusions. All inferences proceed simultaneously, but the system stops after a user-specifiable number of cycles so that the user can interact with it.

The performance claimed for the CIS is very good: Blelloch states that the system can contain 100000 rules and still provide interactive responses. The system obtains high concurrency by taking advantage of concurrent matching and concurrent forward propagation. It is difficult to compare the performance of the CIS with that of any OPS implementation, since the model of matching used in the CIS is very different from that in OPS systems.

Citing the Prospector [6] and Mycin [56] expert systems as examples, Blelloch claims that restricting variables to a finite set of values is not very limiting. As he notes, the requirement that the set of values be known at compile time can be relaxed, provided that the activity network be allowed to change dynamically, something the Connection Machine architecture allows. But no large rule bases have been implemented in the CIS, so the flexibility of the system has yet to be proved.

Another limitation of both the Aspro and the CIS is the absence of negated conditions. Since the Connection Machine supports global reduction operations in parallel, it is possible that negations could be added to the CIS.

4 Non-State-Saving Machines and Algorithms

The two machines proposed for non-state-saving algorithms, DADO and the Delta-Drive Computer, execute different algorithms and are quite different architecturally.

4.1 DADO

The DADO machine [57, 62] represents an attempt to accelerate production system execution through massive parallelism. As originally conceived, the DADO machine consists of a very large number (perhaps 10000 to 100000) of fairly small PE's connected in a complete binary tree. The PE's do not share memory; all inter-PE communication is through I/O circuitry. Each PE has a modest amount of local memory, enough to store a small matching program and some data. The DADO machine functions as a rule coprocessor attached to a conventional host machine.

The current operational prototype, DADO2, has 1023 PE's. Each PE consists of an eight-bit processor (an Intel 8751), 16K bytes of memory, and a semicustom I/O processor. The I/O processor provides rapid bidirectional global communication. Its broadcast circuit allows the host to broadcast data to all PE's in the tree; its *resolve/report* circuit calculates the minimum of a set of 8-bit values submitted by the various PE's, and sets a flag in the PE having the smallest value. All PE's have to participate in each communication through the I/O processor. Each PE can also communicate with its parent and left and right children through a channel separate from the I/O processor. One interesting architectural feature of the I/O processor allows a PE to disconnect itself, under software control, from its parent: a PE that does so becomes the root of its own DADO subtree, and can broadcast data to its descendants, as well as receive data from them through its resolve/report circuit. Several algorithms proposed for DADO (see below) exploit this capability.

The DADO machine combines aspects of SIMD and MIMD computers. Since all PE's must execute the same sequence of communication instructions, communication is SIMD. But between commmunications, the DADO machine is computationally an MIMD computer, since each PE has its own local memory and stored program, and can execute arbitrary non-communicating code.

Stolfo [58] has proposed a number of algorithms for production system execution on DADO. All these algorithms use DADO to accelerate the match and select phases of production system execution, and make the host computer responsible for actions in the RHS of rules.

The *full distribution* algorithm is the simplest, and exploits only production-level parallelism, as described in the previous section. The algorithm also uses DADO I/O hardware during the select and act phases. During the select phase, PE's communicate encoded priorities for their best instantiations to the host. The PE with the highest priority instantiation, as determined by the I/O processor, then communicates its instantiation to the host. During the act phase, the host broadcasts changes to working memory to the DADO PE's; each PE retains those changes that are potentially relevant to its rules.

The *original DADO* algorithm divides the DADO into three components: the PM-level (one level of the tree), the upper tree (those PE's above the PM-level), and the set of WM-subtrees, one for each PE in the PM-level. Each PE in the PM-level receives a number of productions; PE's in the WM-subtree below a PM-level PE store working-memory elements relevant to a rule in that PM-level PE. During the match phase, each PM-level PE constructs all instantiations of its satisfied rules. For each production P with LHS C_1 ... C_n, and for each j in 1,...,n, the algorithm constructs $R_1 \otimes ... \otimes R_{j+1}$ from $R_1 \otimes ... \otimes R_j$. For each

element Q_j of $R_1 \otimes ... \otimes R_j$, the PM-level PE substitutes into C_{j+1} all variables bound in Q_j, broadcasts the resulting constrained condition element, and reads back sequentially all WM elements in its WM-subtree satisfying the broadcast condition element. The PM-level PE combines each such WM element with Q_j to form another element of $R_1 \otimes ... \otimes R_{j+1}$. Thus the PM-level PE's use their WM-level subtrees as generalized content-addressable memory.

Daniel Miranker's *distributed TREAT* algorithm [32] is a state-saving version of the original DADO algorithm. Each PM-level PE stores (either in itself or in its WM-level subtree) the current instantiations of its rules. Each WM-level PE stores a number of Rete alpha-memory tokens; the tokens for a single alpha-memory node are distributed throughout the WM-level PE's. During the match phase of the TREAT algorithm, each PM-level PE first determines which rules are affected by the current changes to its working memory and have non-empty alpha-memories for each positive condition element. For each such rule, the PE then constructs all new instantiations of that rule.

The *fine-grain parallel Rete* algorithm maps the Rete network directly onto the DADO machine. The Rete network without node sharing is in fact a binary tree, so the mapping is a trivial one: leaf PE's store linear chains of one-input tests, while interior PE's receive and- and not-test nodes. During the match phase, the host broadcasts changes to PE's containing one-input tests; these PE's construct tokens for WM elements matching their CEs and pass the tokens to their parent PE's. Two-input test PE's in a pipelined fashion read tokens from their children, construct new tokens from them, and pass the new tokens to their parent PE's. This algorithm processes activations of different nodes in parallel, but because each node is mapped to a single PE, it cannot process multiple activations of a single node in parallel.

Anoop Gupta [16] has suggested another match algorithm for DADO. His algorithm distributes productions to the PM-level and distributes the tokens stored in each Rete match alpha- and beta-memory node throughout the WM-level subtree of the associated production. Each Rete network token for this algorithm consists of the node ID, a list of WME IDs, and a list of the values of the variables used in the following two-input test. During the match phase, the host broadcasts WM changes to the PM-level PE's, which perform all intra-condition tests on them; each PM-level PE then stores the resulting alpha-memory tokens throughout its WM-subtree. For each new token, the WM-level PE broadcasts it and the ID of the opposite memory node to all WM-level PE's; each WM-level PE executes the consistency checks required for the new token. For each two-input test passed, the PM-level PE then creates a new token, which it stores in some WM-level PE. Conflict resolution proceeds as in the original DADO algorithm. The algorithm exploits production-level parallelism and parallelism in the evaluation of each node activation, but it serializes node activations during a single match phase. Gupta's algorithm requires that each PM-level PE store all working memory elements (together with all its attribute values) relevant to any of its rules, as well as a list of the attributes used by each two-input test node. Thus this algorithm requires that the PM-level PE's store more data than their WM-level PE's, and is more suitable to a heterogeneous architecture than to the homogeneous DADO architecture.

Miranker [31] has projected the performance of the DADO2 machine by estimating the number of instructions required to perform each of a number of primitive operations and the number of times the match phase performs each such operation. He concluded that the DADO2 machine running the distributed TREAT algorithm should execute about 212 WM changes/second. Based on the same instruction counts, Gupta [16] estimated an execution speed of 167 WM changes/second for the fine-

grain Rete algorithm. As the instruction counts are wildly inaccurate, neither performance figure can be trusted.

4.2 The Delta-Driven Computer

As part of the European Esprit project, a group at BULL SA Research Center in France has designed a parallel computer architecture, the Delta Driven Computer [50], to execute relational, logic, and functional programs. The Delta Driven Computer is unique in having an intermediate language based on production rules with a forward-chaining inference strategy. Compilers for relational, logic, and function programs translate the programs into production rules.

The architecture of the Delta-Drive Computer is a cluster of bus-based message-passing PE's with no shared memory. Each PE consists of a powerful microprocessor, local memory, and an attached symbolic coprocessor for performing joins, unification, or pattern matching. The authors do not specify the scale of their machine, but from their hopes for speedup on the order of 1000, one can deduce that the machine will contain many PE's.

The authors propose a forward-chaining production rule paradigm as the machine's intermediate language. Their paradigm encompasses rules of the form P_1 & ... & P_n -> A_1 ... A_m, where the P_i and A_j are predicates of atoms and variables. Thus, although their paradigm seems to lack negated condition elements (predicates), it permits more general patterns than OPS5. Execution proceeds by cycles. At each cycle, the interpreter feeds the changes made to the global database back into the rules--hence the name "Delta Driven." All satisfied rules fire on each cycle; the system provides no conflict resolution strategy.

Rather than distributing productions among the processors and duplicating parts of the database, the system distributes individual relations (corresponding to alpha-memories), and duplicates productions. The interpreter further distributes each relation to a number of processors by hashing on attributes used in subsequent joins.

The algorithms suggested for the Delta Driven Computer do not seem suitable as accelerators for OPS-style production systems, since they do not provide negated condition elements or conflict resolution. Further, distributing the elements of individual alpha-memories makes negated conditions difficult to implement without a fast global communication mechanism. On the other hand, it would be very interesting to see how well a machine designed for OPS-style production system execution could execute the intermediate language generated by the Delta-Driven Computer's higher-level language translators.

5 Parallel Processing of Sequentially-Activated Nodes

All the machines described in this section parallelize the Rete algorithm by distributing the contents of Rete alpha- and beta-memory nodes. During the match phase, they process each node activation by using the distributed memory nodes as associative memory. Because this processing requires high communication bandwidth and low latency, the machines are all SIMD.

5.1 NON-VON

The NON-VON machine (proposed by [55]) is a fine-grain-parallel architecture designed to accelerate a wide range of artificial intelligence tasks, including database operations, expert systems, and image understanding. Like DADO, NON-VON is organized as a binary tree, but its granularity is much finer, its architecture is not homogeneous, and its operating modes, and hence the algorithms proposed for it, are different.

NON-VON is actually organized as two very different subsystems connected to a conventional host computer. The *primary processing subsystem* consists of a very large number (perhaps a million, in a full-scale version of the machine) of *small processing elements* (SPE's) connected in a binary tree. Each SPE consists of an 8-bit ALU, a very small amount of memory (perhaps 64 bytes), and connections to its parent and left and right children. Each SPE also has a connection to its predecessor and successor in an inorder traversal of the entire tree; thus algorithms can treat any subtree of the primary processing subsystem as a linearly ordered array. Leaf SPE's are also interconnected in a mesh, but the production system algorithm proposed for NON-VON does not use these connections. The SPE's are SIMD processors: they read and execute instructions broadcast by a controlling processor. An *enabled bit* controls conditional instruction execution; Disabled PE's ignore all instructions except the instruction that enables the PE. A special instruction, the *resolve* instruction, sets a bit in the first enabled PE in an inorder traversal of the SPE tree. Algorithms that process sequentially a set of data stored in the SPE's use the resolve instruction to enumerate the set. Global instructions–the resolve operation and communication between linear neighbors--take ten times as long as computation instructions.

The *secondary processing subsystem* is a highly-connected network of a smaller number (perhaps 31 to 1023) of powerful processors known as *large processing elements (LPE's*. Each LPE has a 32-bit processor, a fairly large memory (at least 256K bytes), and a connection to one of the SPE's in the upper levels of the primary processing subtree. The LPE's operate in MIMD mode, but each LPE can also broadcast instructions and data to the subtree of SPE's to which it is attached. Thus NON-VON supports multiple-SIMD processing, where some subset of the LPE's, typically those attached to SPE's at one level of the primary processing subsytem, control SIMD computation in their respective subtrees.

Bruce Hillyer and David Shaw propose in [21] an algorithm for production system execution inspired by Gupta's algorithm for DADO [16] and by an unpublished algorithm of Daniel Miranker. Their algorithm partitions the rules in a production system into 32 groups; they assume that the partitioning scheme places similar rules in different groups, so that the processing time for each production system cycle is roughly the same for all partitions. Each LPE-SPE subtree rooted at the sixth level (the level containing 32 PE's) receives one of the groups of rules.

Their match algorithm, like Gupta's for DADO, processes node activations sequentially, but uses the tree of PE's as associative memory to speed up each activation. Their algorithm differs from Gupta's in its representation of condition element and Rete network tokens, and in its handling of intra-condition tests. Since each NON-VON SPE has very little memory, it cannot store an entire condition element or a complete Rete network token. Thus Hillyer and Shaw propose that each condition element be stored in a linear array of SPE's, with each SPE storing a single term of the condition, and that each token likewise be stored in a linear array, with each SPE storing a single working memeory element ID. An SPE can store both a term of a condition element and a piece of a token. Each LPE performs intra-condition tests by broadcasting a working memory change and having each SPE test for satisfaction of its term of a

condition element. If the longest condition element in its productions has n terms, the SPE's then compute in (n-1) cycles the AND of the tests for each condition element; this computation exploits the linear connection of the SPE's. LPE's allocate Rete network tokens and process node activations very much as the WM-level PE's do in Gupta's DADO algorithm.

Hillyer and Shaw have analyzed their algorithm by writing NON-VON assembly code for each of the primitive operations in the match cycle and counting machine cycles. Basing their analysis on Gupta's statistics on the execution of six large OPS5 programs, they then compute the average match-phase processing time per working memory change. Assuming an LPE processing speed of 3 MIPS, and an SPE instruction cycle time of 300 microseconds, they conclude that NON-VON should be able to execute about 2000 working memory elements/second. These performance projections depend on a good rule-partitioning strategy, but they do not depend on reducing the variance in the processing time for node activations, since their algorithm processes node activations sequentially.

5.2 The Cellular Array Processor

Ruven Brooks and Rosalyn Lum propose [2] using the ITT Cellular Array Processor (CAP) for production systems. The CAP is an array of perhaps 32 to 128 processors, each with a fairly large local memory, connected to a broadcast bus. The CAP has been realized in a CMOS chip set comprising five chips for a 16-bit CAP processor.

Brooks and Lum suggest a parallelized Rete match algorithm much like the algorithm executed by each NON-VON LPE. Their algorithm distributes constant tests and input memories for two-input nodes among the CAP PE's. During the match phase, the controlling computer broadcasts changes to working memory. Their match algorithm processes node activations sequentially, but uses the CAP PE's to speed up each activation. For each change to working memory, the CAP PE's evaluate the associated constant tests and create or delete Rete network tokens. For each two-input node activation, the CAP PE's search the opposite memory in parallel, creating or deleting new tokens.

The authors suggest performance evaluations of the CAP based on comparisons of aggregate memory-processor bandwidths. For example, they contrast DADO2, with the capacity to process 1023 bytes/instruction, with the 512 bytes/instruction processed by a CAP consisting of 256 16-bit processors operating with a faster clock (each CAP PE can perform a 16-bit addition in 100 nanoseconds), and conclude that the CAP offers comparable processing capacity with many fewer components. They further assert that since two-input node activations consume most of the processing in the Rete match, and since most memory nodes are either large or empty, CAP processor utilization should be high.

The performance claims for the CAP can be questioned on two grounds. First, it is not clear that comparing the speeds of processors of different generations has any relevance to performance evaluation. As Stolfo has pointed out [59], DADO2 was a prototype machine, and a DADO built today would be constructed of fast 32-bit processors. Furthermore, a machine's aggregate processing power is only an upper bound on its performance. Claims for the CAP's performance can be verified only by a detailed study of processor utilization.

The CAP, with its fast communication (broadcasts occur at instruction speed) and lack of contention for shared memory, may accelerate production system execution. However, the CAP, and all parallel processors that use a large number of processors as associative memory, must compete with

uniprocessor indexing schemes that reduce the cost of searching for a pattern in a set of data.

5.3 The OOPS-MOP

A group at the University of Tennessee has designed and fabricated their OOPS-MOP (Our OPS Matching-Only Processor) [38] as part of an accelerator for OPS production systems. They envision a system built of a number of OOPS-MOP slave chips and a controlling processor.

Each OOPS-MOP chip consists of eight identical "chunks", each of which consists of six "slivers" whose outputs are ANDed together; each sliver consists of a programmable arithmetic comparator. The slivers in a chunk compare the value of a single binding variable to five constants or bound variables. Binding a variable and ANDing together a number of comparisons with constants or variables takes one instruction cycle.

The group only hints at a system architecture and an algorithm to exploit the OOPS-MOP. Presumably, each one- and two-input test in an OPS program would be assigned to a different chunk at compile time. During program execution, the controlling processor would execute an algorithm much like Daniel Miranker's TREAT algorithm. Starting with all variables unbound, the controlling processor would try to construct a complete instantiation from working memory elements by binding variables in succession and backtracking when it could not complete an instantiation. Since the OOPS-MOP chips have only enough memory to bind a variable, they cannot store the tokens involved in the Rete match.

The authors suggest that OOPS-MOP's may speed up OPS programs by a factor of 10 or 20 over brute force sequential implementations, but they offer no substantiation for this claim, and they do not specify what they mean by a brute force method; the Rete match algorithm alone speeds up matching by at least a factor of 10.

Lack of on-chip memory and an inflexible instruction set hobble the OOPS-MOP architecture. An OOPS-MOP system must contain a dedicated "chunk" for each one- and two-input test. Since many OPS5 programs contain thousands of tests, a usable OOPS-MOP system would have to contain thousands of matching chips, almost all of which would be idle during each OPS5 match cycle. The OOPS-MOP processor can handle only the simplest matching tasks; the spareness of its instruction set makes it unsuitable even for OPS83.

6 Sequential Processing of Nodes Activated in Parallel

The machines described in this section parallelize the Rete match by assigning individual Rete network nodes to processors, either at compile time or at run time. They require fast access to shared memory containing WM elements, and are based on fast buses.

6.1 The Carnegie-Mellon Production System Machine

Anoop Gupta and others at Carnegie-Mellon University [13, 18] have proposed and analyzed a specialized *Production System Machine* (PSM) to accelerate production system execution. The PSM is a multiprocessor realization of the parallelized Rete match algorithm; processors are assigned dynamically to individual alpha- and beta-node activations. Successful node activations create new tokens; a task scheduler, when passed these tokens, creates new node activations. A few architectural features define

the PSM.

First, the PSM is a shared-memory multiprocessor with a fairly small number (32 to 64) of processors. Gupta justifies the small number of processors by citing the relatively low degree of parallelism he found in the execution traces of the OPS5 programs he studied [15]. The granularity of communication and computation in the PSM dictates a shared-memory architecture. individual node activations, which consume only 50-100 instruction cycles [18], create large tokens that must be communicated to other nodes in the Rete network. Since pointers to tokens are much smaller than the tokens themselves, it is faster to pass pointers to the newly-created tokens, and since processors must dereference pointers to structures created by other processors, the tokens must be in shared memory.

Second, each processor in the PSM is a powerful processor with some private memory and a cache. Since the PSM has relatively few processors, each one can be fairly powerful. But since each processor runs code that is memory-reference-intensive rather than computation-intensive [47], caches are required if processors are not to wait on memory. Further, the caches must be able to store shared data objects such as Rete network tokens, since many of the references to shared memory are to these tokens. Gupta proposes the specialized RISCF processor for the PSM.

Because shared data objects must be cacheable, the PSM must have a cache-coherency scheme. Therefore, processors communicate with shared memory over a shared bus, rather than through an interconnection network such as a crossbar switch or a log(n) stage interconnection network, since cache-coherency schemes for shared buses are much easier to construct ([52], cited by Gupta). The shared bus limits the number of processors in the PSM, since contention for the shared bus seriously degrades performance if the number of processors is larger than 64. Gupta proposes a multiple-bus system for larger numbers of processors.

Finally, a *hardware task scheduler* assigns processes (node activations) to waiting processors. The scheduler sits on the shared bus. A processor passes the scheduler a pointer to a new token by writing to memory-mapped registers in the scheduler; the scheduler assigns a node activation to an idle processor by writing to memory-mapped registers in the processor. Both operations take one bus cycle. The scheduler is responsible for ensuring that multiple activations of a single node that cannot be processed simultaneously are assigned serially. It handles this task by maintaining a *task queue* in associative memory of all active and pending node activations. The task queue must be quite large, since Gupta's simulations show that for the programs he simulated, the maximum number of nodes in the queue was 2000, while the average number was 90.

The hardware task scheduler transforms a fairly conventional general-purpose shared-memory bus-based architecture into a specialized dataflow machine (for a description of dataflow machines, see [22]). Each computation (node activation) in the Rete network can create new tokens, which actively (through the scheduler) create new node activations. The mapping of computations to processors can be completely dynamic, since the shared bus imposes no topological restraints on the pattern of interprocessor communication.

Gupta's performance projections for the PSM are based on extensive event-driven simulations of the execution of six OPS5 programs with a parametrized cost model. Traces of actual OPS5 program execution drive the simulator; they show changes to working memory elements, node activations, and the creation of Rete network tokens. The cost model includes the costs of the operations involved in

scheduling and processing node activations. Gupta estimated these costs by writing hypothetical assembly code to perform various primitive operations and counting processor cycle times. The cost model also includes the effects of contention for shared memory, based on user-specified cache-hit ratios.

Gupta measured both concurrency--the mean number of processors busy--and speedup for parallelization of the Rete match at the production level, at the node level (where the machine processes node activations in parallel, but only one activation of a given node at a time), and at the intra-node level (multiple activations of all nodes). Speedup differs from concurrency in the PSM because of the effects of memory contention, loss of node sharing in the Rete match network, and scheduling overhead. The intra-node parallelism figures are the most significant, since the PSM is designed to take advantage of parallelism at this fine grain. Only one of the six programs Gupta studied showed significant performance gains with more than 32 processors. With 32 processors, concurrency ranged from 4 to 22; the true speedup was less, ranging from 2 to 15. These figures corresponded to execution speeds of 2000 to 14000 WME changes/second. The corresponding figures for node-level parallelism are somewhat worse: concurrency ranged from 4 to 13, speedup from 1.6 to 6.6, and execution speed from 2000 to 9000 WME changes/second. The figures for production-level parallelism are worse still: concurrency ranged from 2 to 12, speedup from 1.2 to 4.8, and execution speed from 800 to 7200 WME changes/second.

Gupta's simulations also showed that the hardware task scheduler is a necessary part of the PSM. Using multiple software task queues rather than a hardware scheduler resulted in a halving of the performance of the system.

According to the simulations, a single-processor PSM should be able to execute about 1000 WME changes/second, which is several times faster than the best OPS83 implementations. Much of the improvement seems to come from a change in the data structures used in the two-input nodes. Current OPS implementations keep the tokens for the left and right inputs of the two-input nodes in linked lists, so that the average cost of deleting a token from a node or of searching the node for a token with consistent bindings is proportional to the length of the list. Gupta proposes keeping all tokens in two global hash tables, one for left inputs of two-input nodes, and one for right inputs. Tokens would be hashed on an identifying number for the node and on the values of the attributes used in the consistency tests in the node. With this scheme, the average cost of deleting a token or of searching a node for tokens satisfying equality tests should be independent of the number of tokens stored in the node. One architectural consequence of maintaining a global hash table for tokens is that processors inserting or deleting tokens must be able to lock an individual hash bucket.

6.2 The Encore Multiprocessor

A group at Carnegie-Mellon University has recently implemented OPS5 on the Encore multiprocessor [19] and collected statistics on execution times of several programs. The Encore multiprocessor resembles the proposed PSM in many ways, so the group's experiments also represent tests of the PSM.

The Encore multiprocessor is a shared-memory bus-based machine with 2 to 20 processors. Each pair of processors shares a 32K byte cache; the caches monitor bus activity to maintain cache coherency, just as in the PSM. Each Encore processor is a 32-bit microprocessor, an NS32032. Since the Encore is a general-purpose machine, it lacks the PSM's specialized hardware task scheduler; therefore, Gupta's experimental results must be compared with his simulation results for a PSM with multiple software task

queues.

The group varied the number of task queues, locking strategies for global hash table buckets, and the number of processors used, and recorded actual execution times for three fairly large OPS5 programs (not the same programs studied in [18]). The results agreed quite well with the predictions of Gupta's simulator: after calibrating the simulator for the NS32032 instruction set, it predicted many execution times that were within 50% of the observed. Speedups over a single-processor execution ranged from 2 to 11 with 13 processors performing matching tasks. One unanticipated phenomenon was considerable contention for shared hash buckets in a program that constructed large cross-products of working memory elements; each hash bucket grew quite large, and each processor spent hundreds of cycles waiting for each access to the hash table.

The experiments on the Encore multiprocessor are a partial, preliminary validation of the effectiveness of the PSM. Gupta's experiments indicated that one major bottleneck in the Encore implementation was slow task scheduling, and his simulator predicted much greater speedup for a machine with a fast scheduler. One the other hand, for some programs, contention for shared hash buckets was the major difficulty, and for these programs, software techniques such as copying and constraining rules promise greater speedup than the hardware task scheduler.

6.3 MANJI

A group of researchers at Keio University in Japan has proposed and evaluated MANJI, a shared-memory multiprocessor for production systems [34]. The algorithm proposed for MANJI differs somewhat from the PSM algorithm, and the differences dictate architectural choices made by the researchers. No performance projections are available, so comparisons with the PSM and the Encore must be tentative.

The MANJI group proposes a Rete match algorithm with distributed node activations, but with nodes assigned statically to processors, several nodes per processor. The arrival of a token at a processor activates the processing of a node. Thus task assignment in MANJI is a distributed operation, while it is a centralized one in the PSM. The PSM requires centralized task assignment because the architecture supports multiple simultaneous activations of a node; since MANJI processes activations of a single node sequentially, it does not require a centralized arbiter.

MANJI is a shared-memory machine with two buses, one for access to working memory, and one for broadcasting Rete match tokens. The match processors use the working memory bus for access to global working memory; in order to reduce bus traffic, each processor has a cache whose block size and replacement strategy are tied to the structure of OPS5 working memory elements. Match processors use the token bus for broadcasting and receiving Rete match tokens. The broadcast mechanism is a receiver-selectable *multicast* (any processor can broadcast). A processor generating a token writes it to an address in global memory determined by the node ID; each destination processor detects the arrival of the token, and reads and processes it. There is no global queue for the activations of each node, so a processor generating a node activation must wait to write the token to global memory until all destination processors have read the previous token at that address. The receiver-selectable multicast has a complex paging mechanism so that processors can avoid mapping the entire global address space into their local address space. The active role of the token arrival queue at each node gives the MANJI machine a distinct dataflow character.

The MANJI group studied the steady-state characteristics of a simple Markov model of bus congestion in the machine. They concluded that most of the time, in the absence of page faults in the receiver-selectable multicast mechanism, bus congestion is minor, and most processors can be kept busy. Presumably, large OPS5 programs would generate page faults, which the model does not take into account.

Several potential problems loom over MANJI. Since nodes are assigned statically to productions, one problem is load balancing, but this problem should not be as severe for node-level parallelism as for production-level parallelism, since there are many more nodes than productions. Another problem is serialization of activations of a single node; Gupta's studies seem to indicate that this serialization limits the available parallelism.

7 Parallel Processing of Nodes Activated in Parallel

In the Carnegie-Mellon Production System Machine [13], the unparallelizable unit of computation is the node activation. Parallelizing single node activations may offer more parallelism than distributing complete node evaluations. Several researchers [14, 51] have suggested dataflow machines as possible architectures for fine-grained execution of production systems. Two proposed dataflow machines try to exploit this parallelism by distributing each Rete memory node among a number of processors. Another machine described below, a distributed-memory message-passing machine, executes a state-saving variant of the Rete match algorithm, which distributes tokens among the PE's at the leaves of a complete binary tree.

7.1 The Waterloo Dataflow machine

Michael Kelly and Rudolph Seviora of the University of Waterloo suggest a distributed Rete match algorithm and a specialized dataflow machine for accelerating production system execution [25]. The architecture they propose is a highly parallel distributed-memory machine with fast global communication. Since their algorithm motivates the architectural specifics of the machine, it is described first.

Their algorithm distributes Rete match alpha- and beta- memory tokens throughout the machine. Each logical Rete node contains a single token, though a processor may hold several such nodes. This scheme lumps memory nodes together with the two-input-test nodes that follow them in the network. Each logical node in the Waterloo machine consists of the tests and ID of a two-input-test node, together with one token from its left or right input.

The arrival of a token (by a mechanism to be specified later) triggers one of several actions. If a matching positive token arrives at the empty side of a logical and-node, the node creates a new composite token and passes it on. If a positive token arrives at the full side of a logical and-node, and that node is the single *generative* copy of the node, the node creates a new copy of itself, gives the new node the token it received, and makes the new node the generative copy. The arrival of a negative token at the full side of a logical and-node destroys the token and the node. Processing not-nodes is similar, but more complicated, since no processor stores global information. The arrival of a token at the negated input of a not-node generates tokens of the opposite polarity. The authors present a scheme for handling the tokens generated at not-nodes that depends on a serializing communication channel.

The proposed architecture consists of a number of PE's interconnected with a bidirectional tree-

structured bus with higher throughput near the root of the tree. (Such an architecture was proposed and analyzed by Leiserson [28], who called it a *fat-tree*.) Each token generated by a logical node flows up and down throughout the entire tree, since all logical nodes must see it. In addition, the PE's are connected locally, perhaps in a grid, to facilitate load balancing. At the end of each match phase, the entire machine executes a load balancing algorithm: each PE passes some of its newly generated logical nodes to its neighbors, choosing the destination PE's by some criterion unspecified by the researchers.

Kelly and Seviora simulated the performance of their machine by simulating the steady-state execution of a single repetitively-fired production. They obtained timings from a register-transfer-level simulator and found a speedup of 2.5 with 4 PE's and 4.5 with 16 PE's. They claim that their machine has the potential to speed up production systems by a factor of 350, the average number of comparisons performed while processing the two-input nodes in the average execution cycle in Gupta's study [15].

Several questions about the performance of Kelly and Seviora's machine remain unanswered. The tree-structured bus, though it is asynchronous, must depend on buffers at each PE for its asynchrony. Since buffers are finite, will the variance in processing time at each PE (processing time must vary in spite of the load-balancing procedure) have a synchronizing effect? Parallelizing the search of a list by distributing its elements and using many processors does speed up the search, but so can hashing techniques, which use only one processor. How much speedup will distributing the contents of memory nodes contribute over hashing them? Also, logical node migration implies code migration as well, since the tests for a logical node migrate with it. How much does code migration increase the communication overhead of the machine? Finally, the potential speedup by a factor of 350 is only an upper bound. How many of the 350 comparisons are inherently sequential?

7.2 PESA-1

A group of researchers at Honeywell Computer Sciences Center has proposed and analyzed a different approach to building a dataflow production system machine. The PESA-1 machine [48, 54] uses buses and random-number generators to distribute Rete network tokens evenly throughout the machine, rather than relying on token replication and a load-balancing algorithm to effect distribution.

Conceptually, PESA-1 is just a bus-based distributed-memory machine built of custom processors. Each PE stores in its local memory all the code for evaluating the production system. PESA-1 executes a distributed Rete match algorithm; the algorithm's execution consists of a series of token creations and node activations. A node is activated when a PE storing part of that node receives a token tagged with the node's ID. If the new token matches the tokens stored in the PE, the PE generates new combined tokens and broadcasts them on the bus. For each new token, the creating PE also generates a destination PE number, using some random number generator. Thus in this distribution scheme, each PE has a random selection of Rete match tokens from all Rete network nodes. The authors do not discuss the processing of not-nodes, but presumably they can be handled as in the Waterloo dataflow machine.

PESA-1's actual architecture and algorithm are somewhat more complicated. The PESA-1 PE's are arranged in levels; at each level i, two shared buses connect it with level i+1 and level i-1. The levels correspond to levels in the Rete match network, and the number of PE's at each level decreases with increasing level number, as Gupta's statistics indicate that the Rete match creates fewer tokens in each successive matching level. If the compiler generates a Rete network with more levels than there are

levels in PESA-1, the compiler folds the network to make it fit in the machine. When a PE creates a token, it tags it not just with its destination PE, but also with a level number.

Simulations with an instruction-level simulator predict an execution rate of 25K working memory element changes/second on a small program (10 productions) and a configuration of 4 PE's in the first level, 32 in the second, 4 in the third, 2 in the fourth, and 1 in the fifth. Analyses of bus contention predict that a bus with a 100-nanosecond cycle time (the same cycle time assumed in the Production System Machine) should be able to handle 160K WME changes/second, so bus contention should not be a limiting factor for PESA-1.

7.3 Oflazer's machine

Kemal Oflazer [41] proposes a tree-structured machine to execute a variant of the Rete match algorithm that saves all nonredundant state information, and provides some performance projections derived from simulations.

Recall that for each production, the Rete match saves the relations (i.e. ordered k-tuples of WME's) satisfying each prefix C_1 & ... & C_k of the production's LHS. Oflazer proposes an algorithm that, for each production, saves the relations satisfying all subsequences C_{i_1} & ... & C_{i_k} of the LHS. Oflazer terms a member of one of these relations an *instance element.* Although a change to working memory can cause changes to many of these relations, most of the processing of these changes can be done in parallel. Oflazer calls a subsequence of an instance element a *redundant* instance element, since the state information it contains is duplicated in the larger one. His algorithm does not save redundant instance elements. Unfortunately, changes to WM can create redundant instance elements, and eliminating them must be done sequentially.

Oflazer proposes a tree-structured distributed-memory multicomputer to execute his algorithm. The proposed machine has several hundred fast processors at the leaves and specialized switches at the interior nodes. The algorithm distributes the stored state for a production among the leaf processors of some subtree, and each processor stores part of the state of a number of productions. During the match phase, processors make changes to their stored state in response to each working memory change, and communicate through the I/O switches at the interior nodes in order to eliminate redundant instance elements.

Results from simulations of three of the OPS5 programs Gupta studied indicated that 512 5- to 10-MIPS processors should execute the systems at about 2200 to 7000 WME changes per second.

The most serious difficulty with Oflazer's algorithm is the potential for an exponential explosion of the size of the saved state. As a remedy for this problem, he suggests splitting productions with many CE's into several productions with fewer CE's each, and connnecting them with message CE's. Unfortunately, this remedy serializes some of the processing, and increases the number of productions fired in a program run.

8 Parallel Rule Firings

A final source of parallelism in production systems is sometimes called *application* parallelism [18], which may consist of multiple threads of control or of firing many non-interfering rule instantiations simultaneously. Several groups have considered multiple rule firings. While their results have generally been inconclusive, their methods promise to have some influence on production system architectures.

D. I. Moldovan and F. M. Tenorio [63, 35, 36] have studied the problem of partitioning production systems in order to minimize communication among partitions. They identify four types of dependencies between productions, which they term output dependence, interface dependence, input-output dependence, and input dependence. Only two types of dependence prevent simultaneous rule firings: input dependence, where one production's actions destroy the preconditions for another production's firing, and input-output dependence, where one production creates preconditions for another. When two rules cannot fire simultaneously, they must be in the same partition, or there must be communication between their respective partitions. The authors built a simulator that estimated total execution time, taking into account synchronization constraints and communication costs for various networks, and claim that a 16-processor system should execute 4000 rule cycles per second.

One problematic aspect of Moldovan and Tenorio's research is that it derives partitionings that are very different from Oflazer's. Oflazer's methods, which addressed load-balancing problems, placed similar productions into different partitions. But similar productions are likely to have similar pre- and post-conditions, and hence to have dependencies. Oflazer's partitioning trades communication (broadcasting working memory changes to all partitions in the hope that each change affects all partitions) and storage (replicated working memory elements) for load balancing, while Moldovan and Tenorio's partitioning seems to do the opposite.

Toru Ishida and Salvatore Stolfo have analyzed dependencies and synchronization requirements in production systems in [23, 24]. They also propose a framework for multiple rule firings on tree-structured machines.

They construct a graph with one vertex for each production and working memory class and plus- and minus-labeled directed edges between productions and working memory classes. The graph contains an edge from a production to a working memory class if that production can create (+) or destroy (-) a working memory element of that class. Likewise, the graph contains an edge from a working memory class to a production if that class appears in a positive (+) or negative (-) condition element on the LHS of the production. They distinguish three classes of interactions between productions. Suppose P and Q are two productions. If, for every working memory class W such that P has a + edge to W, there is no edge, or a there is a + edge from W to Q, then P and Q can be fired in parallel. If there is a class W such that P has a + edge to W, but W has a - edge to Q, then P and Q may not be fired in parallel, since firing P may destroy Q's execution environment. Finally, if there is a class W such that P has a + edge to W, while Q has a - edge to W, then P and Q cannot be fired in parallel, since the result may not be the same as any serial firing of P and Q.

Ishida and Stolfo next designed a hierarchical partitioning algorithm intended to maximize parallel execution. Applying this algorithm to one production system whose synchronization analysis had been improved manually, they found that they could fire an average of seven productions in parallel. Based on this splitting algorithm, they propose a scheme for executing multiple rule firings in parallel on a tree-

structured machine such as DADO.

An unresolved problem in the work of both groups is the consistency of their multiple-firing schemes with standard conflict resolution strategies. For instance, suppose that instantiations of rules P and Q both exist, but that the conflict resolution strategy favors P's instantiation. Suppose further that firing P creates a working memory element that creates another instantiation of Q, and that conflict resolution selects Q's second instantiation for firing. Then the work of Ishida and Stolfo allows P's instantiation and Q's first instantiation to fire simultaneously, while sequential execution with conflict resolution fires P's instantiation, followed by Q's second instantiation.

A. O. Oshisanwo and P. P. Dasiewicz propose in [42] a heterogeneous production system machine, MAPPS, and a scheme for allowing multiple rule firings. They propose a run-time check for conflicting instantiations, and present high performance predictions for a distributed Rete algorithm running on the machine.

The MAPPS machine comprises three sets of PE's. The first set executes constant tests, which are distributed statically among it. The second set consists of clusters of PE's sharing common memory within a cluster but communicating between clusters on a bus; each cluster executes the two-input tests for a set of productions, which constitutes a subset of a partition. The third set is a tree of PE's for conflict resolution and execution of multiple rule firings.

Oshisanwo and Dasiewicz state that an (unspecified) statistical model of production systems based on Gupta's statistics gave simulated execution times of 10K to 19K working memory element changes per second. They are now working on a register-transfer-level simulator of the machine.

Unfortunately, the authors do not present the details of their run-time strategy for avoiding the firing of conflicting rules. They seem to imply that the cost of the run-time checks involved is linear in the number of partitions, the number of condition elements in each production, and the number of working memory elements created or destroyed by each rule. Since each pair of instantiations must be checked for conflicts, the obvious algorithms have costs quadratic in the number of partitions. If the number of partitions is large, this check is likely to be expensive, and a major bottleneck in the machine. rtitions.

9 Conclusion

This paper has presented descriptions of proposed production system architectures classified according to properties of the algorithm(s) each is designed to execute. Almost no consensus can be discerned about any aspect of production system architecture or method of parallelizing the match computation. The proposed architectures include shared- and distributed-memory machines. Among the distributed-memory computers, PE's communicate through buses, interconnection networks of various topologies, or a combination of the two. Researchers have proposed homogeneous and heterogeneous machines, both SIMD and MIMD, with processors ranging from single-bit ALU's to powerful custom 32-bit chips. The proposed algorithms occupy several points in the state-saving spectrum, although most are parallelizations of the Rete match. Parallel versions of the Rete match algorithm parallelize different parts of the computation. The algorithms display tradeoffs between the number of parallel tasks (inversely, the granularity of each task) and delays due to communication, synchronization, and scheduling constraints and overheads. Each parallelization technique distributes some of data structures and replicates some data, as displayed below.

Figure 3: Taxonomy of Parallel PS Algorithms

Scheme	Distributed	Replicated
Rete Match		
Rule-level parallelism	PM	Alpha-memories
Sequential node activations, parallel processing of each	Contents of alpha- and beta-memories	Code (possibly at runtime)
Parallel node activations, sequential processing of each	One- and two-input tests (statically or dynamically distributed)	Alpha- and beta-memories (if cached)
Parallel node activations, parallel processing of each	Contents of alpha- and beta-memories	Code
Other		
Distributed TREAT	PM, alpha-memories, conflict set	WM
State-saving	All partial instantiations	Code, WM
Parallel rule firings	set of RHS	Code

The wide range of parallel algorithms for PS execution stems partly from ignorance about the run-time behavior of production systems. In particular, questions such as how much state a production system interpreter should save, and how parallelizable production systems are, still lack definitive answers. Miranker's comparisons of TREAT and Rete [31, 33] show that, for at least some OPS5 production systems, maintaining the state saved by Rete is computationally more expensive than throwing away partial instantiations that do not form part of a complete instantiation, and recomputing them as necessary. Although Gupta's statistics indicate that for the systems he studied, most changes to working memory caused changes to the stored state of some 26 productions, the size of his sample was quite small--he published statistics on only twelve production systems, all written in OPS5. Although he tried to select a representative sample of OPS5 programs, the phenomena he noted may be artifacts of a programming style dictated by the language (*cf.* [32, 65]). Also, the number of affected rules in each PS cycle is closely related to the number of changes to WM in each cycle. Firing several rules in each cycle can increase the number of changes to WM, but semantic problems involved in multiple rule firing must still be resolved.

The performance projections offered by various research groups do not go far toward establishing sound bases for comparing different machines. The projections are based almost entirely on simulation results, in some cases at a register-transfer level, in others at the level of events such as node activations. Some working prototype machines (such as DADO) exist, but few performance measurements for them have been published. Device technology and processor cycle time are important determinants of performance, but each research group assumes different values of these parameters. To make matters worse, performance measurements on prototypes cannot be compared directly with performance projections, because the prototypes may not use current technology. Finally, the different degrees of rule compilation assumed in different machines, and the confounding role of the interpreter's

base language, make comparisons even more difficult.

Even if the various performance projections were normalized and directly comparable, most of them would still be inadequate for evaluating the proposed machines. Simulations at the register-transfer level, for example, give very accurate timing information, but they must be run on very small data sets, and can simulate only very small systems. Conclusions based on these small systems cannot be generalized to very large systems in the absence of statistical and worst-case knowledge about their behavior. On the other hand, simulations driven by events such as Rete node activations can give information about much larger systems, but the results are only as general as the data driving the simulations are representative. Statistical models of production system behavior offer generality, but only the MANJI group have constructed such models, and only of bus contention in their machine (which is probably not its performance-limiting factor anyway).

Most published analyses of production system machines omit two important evaluation criteria: generality and efficiency. Production system machines should be general--capable of efficient execution of more than OPS5 programs--because not all production systems are OPS5 programs. Performance evaluations of these machines should consider not just throughput, but should also include studies of utilization and efficiency.

In summary, although many interesting algorithms and machines for acccelerating production system execution have been proposed, it is impossible to use predict how well they will perform on real systems. Furthermore, even though most of the proposed machines are designed to execute one language (OPS5), it is impossible to conclude which is fastest. Given the large number of proposed machines, and the importance of speeding up production systems, a more systematic and general evaluation is required.

References

[1] Blelloch, G. E.
CIS: A Massively Concurrent Rule-Based System.
In *Fifth National Conference on Artificial Intelligence*, pages 735-741. AAAI, 1986.

[2] Brooks. R., and Lum, R.
Yes, An SIMD Machine Can Be Used For AI.
In *Ninth International Joint Conference on Artificial Intelligence*, pages 73-79. ACM, 1985.

[3] Brownston, L., Farrell, R., Kant, E., and Martin, N.
Programming Expert Systems in OPS5.
Addison-Wesley, Reading, Massachusetts, 1985.

[4] Chandra, A. K., and Merlin, P. M.
Optimal Implementation of Conjunctive Queries in Relational Data Bases.
In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77-90. 1977.

[5] Davis, R., and King, J.
An Overview of Production Systems.
Machine intelligence 8: Machine Representations of Knowledge.
Ellis Horwood, Chichester, 1977, pages 300-322.
Also published as Technical Report AIM-271, Computer Science Department, Stanford University, 1975.

[6] Duda, R. O. Hart, P. E., Konolige, K., and Reboh, R.,
A Computer-based Consultant for Mineral Exploration.
Technical Report, SRI International, September, 1979.

[7] Forgy, C. L.
OPS4 User's Manual.
Technical Report CMU-CS-79-132, Computer Science Department, Carnegie-Mellon University, July, 1979.

[8] Forgy, C. L.
On the Efficient Implementation of Production Systems.
PhD thesis, Carnegie-Mellon University, 1979.

[9] Forgy, C. L.
Note on Production Systems and Illiac IV.
Technical Report CMU-CS-80-130, Computer Science Department, Carnegie-Mellon University, 1980.

[10] Forgy, C. L.
OPS5 User's Manual.
Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, July, 1981.

[11] Forgy, C. L.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence (19):17-37, 1982.

[12] Forgy, C. L.
The OPS83 Report.
Technical Report CMU-CS-84-133, Computer Science Department, Carnegie-Mellon University, May, 1984.

[13] Forgy, C. L., and Gupta, A.
Preliminary Architecture of the CMU Production System Machine.
In *Nineteenth Hawaii International Conference on System Sciences*, pages 194-200. ACM, Kona, Hawaii, January, 1986.

[14] Graham, P. C. J.
Providing Architectural Support for Expert Systems.
Computer Architecture News 12(5):12-18, 1984.

[15] Gupta, A., and Forgy, C. L.
Measurements on Production Systems.
Technical Report CMU-CS-83-167, Computer Science Department, Carnegie-Mellon University, 1983.

[16] Gupta, A.
Implementing OPS5 Production Systems on DADO.
In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 83-91. IEEE, 1984.

[17] Gupta, A., Forgy, C. L., Newell, A., and Wedig, R.
Parallel Algorithms and Architectures for Rule-Based Systems.
In *The 13th Annual International Symposium on Computer Architecture*, pages 28-37. IEEE, Tokyo, Japan, 1986.

[18] Gupta, A.
Parallelism in Production Systems.
PhD thesis, Carnegie-Mellon University, March, 1986.

[19] Gupta, A., Forgy, C. L., Kalp, A., Newell, A., and Tambe, M.
Results of Parallel Implementation of OPS5 on the Encore Multiprocessor.
Technical Report CMU-CS-87-146, Computer Science Department, Carnegie-Mellon University, 1987.

[20] Hillis, W. D.
The Connection Machine (Computer Architecture for the New Wave).
Technical Report 646, M.I.T. Artificial Intelligence Laboratory, September, 1981.

[21] Hillyer, B. K., and Shaw, D. E.
Execution of OPS5 Production Systems on a Massively Parallel Machine.
Journal of Parallel and Distributed Computing 3(2):236-268, 1986.

[22] Hwang, K., and Briggs, F. A.
Computer Architecture and Parallel Processing.
McGraw-Hill, New York, New York, 1984.
Pages 732-768.

[23] Ishida, T., and Stolfo, S. J.
Simultaneous Firing of Production Rules on Tree Structured Machines.
Technical Report CUCS-109-84, Computer Science Department, Columbia University, 1984.

[24] Ishida, T., and Stolfo, S. J.
Towards the Parallel Execution of Rules in Production System Programs.
In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 568-575. IEEE, 1985.

[25] Kelly, M. A., and Seviora, R. E.
A Multiprocessor Architecture for Production System Matching.
In *Sixth National Conference on Artificial Intelligence*, pages 36-41. AAAI, 1987.

[26] Lehr, T. F.
The Implementation of a Production System Machine.
In *Nineteenth Hawaii International Conference on System Sciences*, pages 177-186. ACM, Kona, Hawaii, January, 1986.

[27] Lehr, T. F.
The GaAs Realization of a Production System Machine.
In *Nineteenth Hawaii International Conference on System Sciences*, pages 246-252. ACM, Kona, Hawaii, January, 1986.

[28] Leiserson, C. E.
Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing.
In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 393-402. IEEE, 1985.

[29] McDermott, J.
R1: A Rule-based Configurer of Computer Systems.
Artificial Intelligence (19):39-88, 1982.

[30] Minsky, M.
A Framework for Representing Knowledge.
The Psychology of Computer Vision.
McGraw-Hill, New York, 1975.

[31] Miranker, D. P.
Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE.
Technical Report CUCS-118-84, Computer Science Department, Columbia University, 1984.

[32] Miranker, D. P.
Treat: A New and Efficient Match Algorithm for AI Production Systems.
PhD thesis, Columbia University, October, 1986.

[33] Miranker, D. P.
TREAT: A Better Match Algorithm for AI Production Systems.
In *Sixth National Conference on Artificial Intelligence*, pages 42-47. AAAI, 1987.

[34] Miyazaki, J., Amano, H., and Aiso, H.
MANJI: An Architecture for Production Systems.
In *Twentieth Hawaii International Conference on System Sciences*, pages 236-245. ACM, Kona, Hawaii, January, 1987.

[35] Moldovan, D. I.
A Model for Parallel Processing of Production Systems.
In *Proceedings of the 1986 IEEE International Conference on Systems, Man, and Cybernetics*, pages 568-573. IEEE, 1986.

[36] Moldovan, D. I.
A Multiprocessor for Rule-Based Systems.
In *Proceedings Supercomputing '87*, pages 482-490. International Supercomputing Institute, Inc., 1987.

[37] Newell, A.
Production Systems: Models of Control Structures.
Visual Information Processing.
Academic Press, New York, 1973, pages 463-526.

[38] Newport, D. F., Alley, G. T., Bryan, W. L., Eason, R. O., and Bouldin, D. W.
A Parallel Symbol-Matching Co-processor for Rule Processing Systems.
In *Proceedings of the 1986 IEEE International Conference on Systems, Man, and Cybernetics*, pages 578-81. IEEE, 1986.

[39] Nilsson, N. J.
Principles of Artificial Intelligence.
Tioga Publishing Company, Palo Alto, California, 1980.

[40] Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 92-100. IEEE, 1984.

[41] Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
PhD thesis, Carnegie-Mellon University, March, 1987.

[42] Oshisanwo, A. O., and Dasiewicz, P. P.
A Parallel Model and Architecture for Production Systems.
In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 147-153. IEEE, 1987.

[43] Pasik, A.
The OPS Family of Production System Languages.
Technical Report CUCS-232-86, Computer Science Department, Columbia University, 1986.

[44] Patterson, D. A., and Sequin, C. H.
A VLSI RISC.
IEEE Computer 15(9):8-21, 1982.

[45] Perlin, M.
On the Computational Equivalence of Frame Systems and Rule Systems.
In *U.S.-Japan AI Symposium 87.* Tokyo, November, 1987.

[46] Quillian, R.
Semantic Memory.
Semantic Information Processing.
MIT Press, Cambridge, Massachusetts, 1968.

[47] Quinlan, J.
A Comparative Analysis of Computer Architectures for Production System Machines.
In *Nineteenth Hawaii International Conference on System Sciences*, pages 187-193. ACM, Kona, Hawaii, January, 1986.

[48] Ramnarayan, R., Zimmermann, G., and Krolikoski, S.
PESA-1: A Parallel Architecture for OPS5 Production Systems.
In *Nineteenth Hawaii International Conference on System Sciences*, pages 201-205. ACM, Kona, Hawaii, January, 1986.

[49] Reed Jr., B.
The Aspro Parallel Inference Engine (P.I.E.): A Real Time Production Rule System.
Technical Report 85-6048, Goodyear Aerospace, 1985.
Pages 459-464.

[50] Rohmer, J., and Gonzalez-Rubio, R.
Delta Drive Computer: A Parallel Machine for Symbolic Processing.
In *Convention Informatique*, pages 150-155. SICOB, Paris, France, 1986.

[51] Rokey, M.
The Dataflow Architecture: A Suitable Base for the Implementation of Expert Systems.
Computer Architecture News 13(4):8-14, 1985.

[52] Rudoph, L., and Segall, Z.
Dynamic Decentralized Cache Schemes for MIMD Parallel Processors.
In *Eleventh Annual International Symposium on Computer Architecture*, pages 340-7. Ann Arbor, Michigan, 1984.

[53] Rycher, M. D.
A Semantic Network of Production Rules in a System for Describing Computer Structures.
In *Sixth International Joint Conference on Artificial Intelligence*, pages 738-743. ACM, 1979.

[54] Schreiner, F., and Zimmermann, G.
PESA 1--A Parallel Architecture for Production Systems.
In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 166-169. IEEE, 1987.

[55] Shaw, D. E.
The NON-VON Supercomputer.
Technical Report CUCS-29-82, Computer Science Department, Columbia University, August, 1982.

[56] Shortliffe, E. H.
Computer-based Medical Consultations: MYCIN.
Elsevier, New York, 1976.

[57] Stolfo, S. J., and Miranker, D. P.
DADO: A Parallel Processor for Expert Systems.
In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 74-82. IEEE, 1984.

[58] Stolfo, S. J.
Five Parallel Algorithms for Production System Execution on the DADO Machine.
In *Proceedings of the National Conference on Artificial Intelligence*, pages 300-307. AAAI, 1984.

[59] Stolfo, S. J.
A Note on Implementing OPS5 Production Systems on DADO.
Technical Report CUCS-130-84, Computer Science Department, Columbia University, 1984.

[60] Stolfo, S. J.
On the Design of Parallel Production System Machines: What's in a LIP?
In *Eighteenth Hawaii International Conference on System Sciences*, pages 232-237. ACM, Kona, Hawaii, January, 1985.

[61] Stolfo, S. J., Miranker, D. P., and Mills, R. C.
More Rules May Mean Faster Parallel Execution.
In *Proceedings of the Workshop on AI and Distributed Problem Solving.* Washington, D. C., May, 1985.

[62] Stolfo, S. J., and Miranker, D. P.
The DADO Production System Machine.
Journal of Parallel and Distributed Computing 3(2):269-296, 1986.

[63] Tenorio, M. F. M., and Moldovan, D. I.
Mapping Production Systems into Multiprocessors.
In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 56-62. IEEE, 1985.

[64] Ullman, J. D.
Principles of Database Systems.
Computer Science Press, Rockville, Maryland, 1982.

[65] van Biema, M., Miranker, D. P., and Stolfo, S. J.
The Do-Loop Considered Harmful in Production System Programming.
In *First International Conference on Expert Database Systems*, pages 88-97. ACM, Charleston, South Carolina, April, 1986.

[66] Vesonder, G., Stolfo, S. J., Zielinsky, J. E., Miller, F. D., and Copp, D. H.
ACE: An Expert System for Telephone Cable Maintenance.
In *Eighth International Joint Conference on Artificial Intelligence*, pages 116-121. ACM, 1983.