Parallel Processing for Atificial Intelligence
L. Kanal, V. Kumar, H. Kitano and C. Suttner, editors
©1993 Elsevier Science Publishers B.V. All rights reserved

Chapter 1

# Speeding Up Production Systems: From Concurrent Matching to Parallel Rule Firing[*]

José Nelson Amaral and Joydeep Ghosh

*Electrical and Computer Engineering Department*
*The University of Texas at Austin, Austin, TX 78712, U.S.A.*

This chapter identifies the problems that a computer architect faces in attempting to speed up the execution of production systems. We first focus on state-saving algorithms using Rete networks because they were the major source of inspiration for a number of research efforts in the eighties'. Early attempts to speed up production systems almost exclusively concentrated on concurrently executing the match phase of the match-select-act loop. More recent studies have shown that significant speedup will not be obtained unless architectures include the capability for parallel rule firing. However, difficult problems, such as the identification of dependencies among different rules and guarantee of correctness, arise when rules are fired in parallel. The second part of this chapter identifies these problems and critiques some of the solutions proposed by different researchers.

## 1. Introduction

A production system consists of a knowledge base, a set of rules or productions, and an inference engine. The knowledge base is formed by a set of tuples containing facts about a given domain. A production consists of a collection of conditions or premises about the knowledge base and a set of actions. These actions are addition, deletion, or change of facts in the knowledge base. The inference engine is a computational mechanism that has three distinct functions: determining which rules are satisfied or

[*]This research is sponsored by Conselho Nacional de Pesquisas Cientificas e Tecnologicas - CNPq, Pontificia Universidade Catolica do Rio Grande do Sul - PUCRS, Brazil, and by a Faculty Development Award from TRW Foundation. We would like to acknowledge Jeffrey Draper and Michael McClung for their helpful comments. Please contact *nelson@pine.ece.utexas.edu* or *ghosh@pine.ece.utexas.edu* for comments or reprint requests.

enabled, selecting a rule or a set of rules to be executed or “fired”, and performing the actions specified by the rules.

A problem is specified as a set of facts in the knowledge base, a set of productions related to the facts, and the specification of a *desired* or *final* state. The knowledge base is said to be in an *initial* state before any of the actions specified by productions is performed. The inference engine *moves* the knowledge base over the knowledge space. The objective is to reach a final state in a minimum amount of time.

The quest for speeding up expert systems, which are commonly realized in the form of production systems, has led to efforts in different directions over about twenty years of research. One of the most important contributions to this research was the creation of Rete networks by Forgy in 1979 [5]. The use of Rete networks to encode the conditions of rules allowed the implementation of efficient match algorithms. Since it was published, many improvements and modifications to Rete have been proposed.

Another important contribution to the development of expert systems was the creation of OPS5 systems [6]. The OPS5 model requires the generation of a complete conflict set at each cycle and the selection of a single rule from this set, to be acted upon. The most common criterion used for rule selection is recency. The reasoning behind the recency criterion is that rules enabled more recently are more likely to move the knowledge system towards a goal state.

Early research estimated that in OPS5-like production systems, the matching of rules against facts in the knowledge base takes approximately 90% of the computing effort [10]. Consequently, a considerable amount of effort was dedicated to speeding up the matching phase of the production cycle. This observation also led to the conclusion that the speeding up of expert systems by means of parallel techniques is limited to approximately tenfold [9]. This is a direct consequence of Amdahl's law.

More recently, research using compiled versions of the TREAT algorithm at The University of Texas at Austin [24, 19] have indicated that the amount of time taken by the match phase might be around 50% of the computing effort, possibly getting as low as 30% for some programs. If these numbers are accurate, the maximum speedup that can be expected from match processing improvements alone is twofold. Therefore, there is a strong belief in the research community that it is mandatory to improve the other phases of the production system cycle to obtain significant speedup. Some researchers [16] propose to eliminate global synchronization altogether. The elimination of synchronization at each cycle and the firing of rules in parallel would increase the amount of changes in the knowledge base over time, accelerate the movement of the knowledge base over the knowledge space, and hopefully make the system reach the goal state sooner.

Problems in the maintenance of correctness and consistency of the knowledge base arise in parallel rule firing. There are a variety of definitions for the possible inconsistency of rules and a few different approaches to ensure correctness while allowing parallel rule firing. Optimization in the use of processing resources is also an issue in the parallel execution of expert systems. The way the rule system is partitioned is critical in guaranteeing high usability of processors and low communication overhead.

Kuo and Moldovan have published a detailed, well-written survey on the state of the art in parallel production systems [22]. They emphasize efforts in implementing data-driven systems [33] and initiatives in parallel productions systems like the IRIS programming methodology [29], Ishida's work [14], PARS [31], RUBIC [26], CREL [20], CUPID [17], and PARULEL [37]. This chapter partially overlaps with that survey, but also complements and updates it. In the next section we present a generic model for the architecture of an expert system. Section 3 introduces state-saving algorithms with emphasis on Rete network based algorithms. Section 4 discusses concurrent matching and parallel execution of Rete. Section 5 discusses optimization of Rete networks at compile time without parallel rule firing. Section 6 presents some key efforts towards parallel rule firing, and section 7 consists of a discussion of current trends in the research towards speeding up expert systems.

## 2. A Generic Production System Architecture

Most of the research towards speeding up expert systems via faster matching uses an architectural model similar to the one represented in fig. 1. The memory of the system is divided into a set of productions or rules stored in the production memory, and a set of facts stored in the working memory. The working memory gets its name from the fact that it is used as a "scratch" memory where the system writes and overwrites partial results. Each fact of the knowledge domain is stored in this memory as a unit called a Working Memory Element (WME).

A rule stored in the production memory consists of a set of conditions and a set of assertions. The rules are usually syntactically expressed with the conditions positioned to the left of an arrow. Therefore, the conditions are called the Left Hand Side (LHS) of the rule. Similarly, the assertions or actions, positioned to the right of the arrow, are called the Right Hand Side (RHS) of the rule. Some research groups [4] adopted a better nomenclature which labels the conditions the *antecedents* of the rule and the assertions the *consequents* of the rule.

A rule in the production memory is said to be *fireable* if all its non-

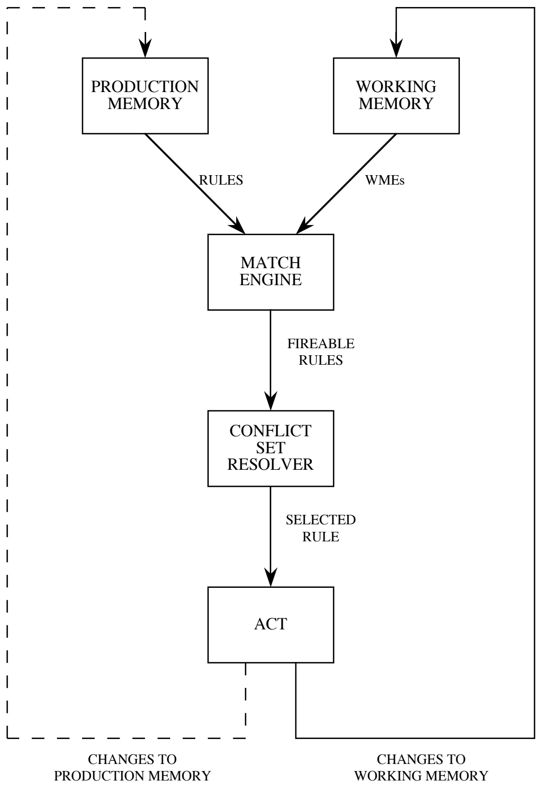


Fig. 1. Generic Production System Architecture

negated conditions are satisfied and none of its negated conditions is satisfied. Also, if variables appear in more than one condition element, all instantiations of the same variable must be bound to the same value.

In an OPS5-like system, the match engine of fig. 1 compares (or matches) all conditions of all rules in the production memory against all facts in the working memory, while keeping track of variable bindings to check which rules are fireable. The set of all fireable rules at the end of the match processing is called the *conflict set.*

The conflict set resolver decides which rule from the conflict set will be selected to fire in the current cycle. Criteria used to select the rule include: recency, specificity, priority, and context.

After a rule is selected to fire, the act phase of the system produces changes in the memory, creating or deleting WMEs. Most of the production systems produce changes only to the Working Memory, altering the facts in the knowledge base. Some systems also produce new rules or eliminate old ones. We suggest that such systems be called *adaptive expert systems* [16] or *learning expert systems*, because they have the capability of adapting to changes in the environment.

## 3. State-Saving Algorithms

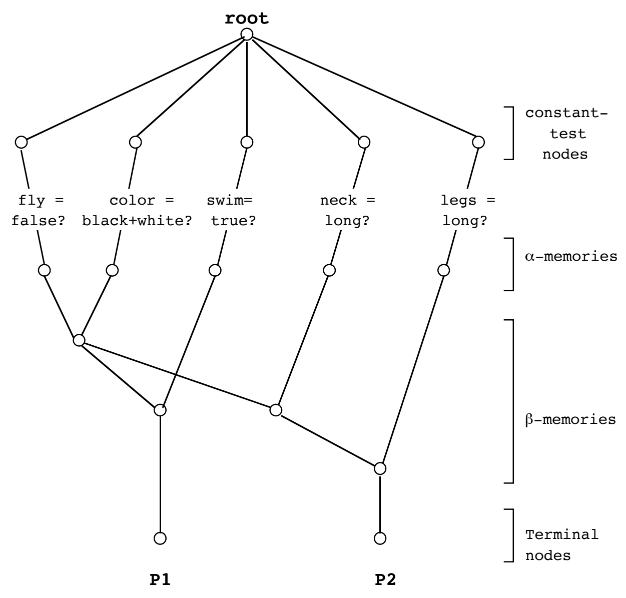
The work performed by the match engine is a combinatorially explosive problem. However, there are two characteristics of production systems that allow a good approach to this problem. The pieces of knowledge stored in the working memory of a production system change slowly over time. This implies that if the results of the matching in one cycle are saved for the next cycle, a substantial amount of work can be eliminated. The other characteristic is that there are many identical condition elements in different rules. Therefore, an algorithm that allows these conditions to be shared by distinct rules must match a condition only once, regardless of the number of rules in which the condition appears.

### *3.1. Rete Networks*

The Rete network was created by Forgy [5]. Forgy reports that Rete is inspired by the *Pandemonium* machine of Selfridge [34]. Pandemonium was one of the earliest learning machines and consisted of multiple layers of *demons*. A demon in a given level supervised an inferior level of demons. When it observed meaningful patterns, it sent messages to a superior level. The top-level demons performed more telling actions.

The Rete network is a data-flow graph that encodes the antecedents of rules. The inputs to the Rete network are changes to the working memory generated in the act phase of one cycle, and the outputs of the network are changes to the conflict set used to choose a rule to be fired in the next cycle. The following discussion of Rete networks is presented here after [8], [25], [10], and [23].

Fig. 2 presents a Rete network with the set of rule antecedents encoded in it. The network is formed by four different kind of nodes: constant-test nodes, memory nodes, two-input nodes and terminal nodes. The constant-test nodes appear in the first layer of the network. They store attributes that have constants in the value field and perform intra-condition tests to determine if a working memory element satisfies these constant fields of the condition element. In the original Rete network, the result of this test was



```
(P P1 (bird <b>)
         (fly <b> false)
         (color <b> black+white)
         (swim <b> true)
         -->
         (type <b> Penguin))
```

```
(P P2 (bird <b>)
          (fly <b> false)
          (color <b> black+whit
          (neck <b> long)
          (legs <b> long)
          -->
          (type <b> Ostriches))
```

Fig. 2. Rete Network

stored in a local $\alpha$-memory [10]. Some authors now claim that the one-input nodes can be memoryless because the constant test takes a negligible amount of time [23].

Two-input nodes, also called *and*-nodes, *join*-nodes, or $\beta$-nodes, perform the matching between distinct condition elements. All tokens arriving in a two-input node come from a memory node, and whenever a new token arrives in one of a node's inputs, it is compared with all tokens present in the other input. The results of the join operation performed in the two-input node are stored in a local $\beta$-memory. Good hashing techniques are necessary to speed up the matching in the two-input nodes. Otherwise, it might be necessary to process long lists of tokens. [8].

Terminal nodes receive a token when a production should be inserted into or removed from the conflict set. There is one terminal node for each production in the system. The data-flow discrimination network compiled by the Rete Algorithm produces nodes with a maximum fan-in of two [38].

The Rete algorithm implements the network described above and then keeps feeding it with changes to the working memory and extracting from it changes to the conflict set. The state of the system is stored in the memory nodes of the network. To enable the use of Rete networks, a problem must be compiled into primitive match tests. Rete networks are not adequate for match problems in which the objects are not constants or where the set of objects changes fast. Because it is a state-saving algorithm, it is inefficient when the data used change substantially from cycle to cycle [38].

### *3.2. TREAT Algorithm*

There is a tradeoff in the amount of information that should be saved from one cycle to the next in the Rete algorithm. Saving all the information from cycle to cycle can have a negative effect, especially if the knowledge changes substantially from cycle to cycle. On the other hand, saving too little of the state may cause some matches to be repeated frequently. Miranker pointed out that there is an overhead in memory management of the $\beta$-memories in the join nodes. This occurs because when WMEs are removed, it is necessary to perform the match to decide which memories need to be deleted. To solve this problem, Miranker created a modified algorithm called TREAT to process Rete networks [25].

The TREAT algorithm eliminates $\beta$-memories attached to join nodes. As a consequence, it is necessary to perform the matching operation each time a WME is created, but it is not necessary to perform any matching when a WME is deleted. Miranker presents empirical results that indicate that the time saved in deletions compensates for the extra time spent in addition of WMEs, and therefore TREAT is faster than Rete [25]. It is necessary to observe that TREAT is not a state-saving algorithm in the same sense as the Rete algorithm, and it is only comparable to Rete in that it makes use of a similar data-flow structure.

### *3.3. Generalized Rete Networks*

In the original Rete network, the joins in the match pattern are limited to left-associative joins. This means that a join node can have another join node only as its left predecessor. The right predecessor must be a single pattern. Later, extensions allowed arbitrary group of conditions. However, some naive extensions of Rete may introduce errors, such as duplicate or missing joins, leading to incorrect match results. Lee and Schor [23] pointed out that such a situation arises when a join node has successors that reconverge at a subsequent join node. They introduced an algorithm that handles this problem correctly without any loss of efficiency. Furthermore, this algorithm permits incremental addition of new rules and demand-driven pattern matching by means of blocking a portion of the network from being updated during normal operation and enabling it on demand.

Dynamic addition of rules in a production system is difficult to map into a Rete network because it implies augmenting the network. Since state is stored throughout the network, the contents of several $\beta$-memories in existent nodes may need to be propagated to the new nodes corresponding to the added rule. Thus, a single node (the recently inserted one) may receive many updates from multiple predecessor nodes.

Generalized Rete networks have the advantages of increasing match computation sharing, especially in join result sharing, and eliminating the "long-chain" effect. This effect occurs when a large number of patterns must be joined. Since a join node can have only its left successor as another join node in the original Rete, it is necessary to create a long chain of nodes to accomplish the task. One disadvantage of generalized Rete networks is an increased likelihood that large cross-product effects occur. The cross-product effect refers to the increase in $\beta$-memory storage and matching time when a single token flowing into a two-input node finds a large number of tokens with consistent variable bindings in the opposite memory of the node [10].

## *3.4. Rete for Real-Time Systems*

Barachini and Theuretzbacher [2] have introduced PAMELA (PAttern Matching Expert system LAnguage), which extends the Rete algorithm to allow interrupt handling necessary for real-time systems. They propose a reduction of the Rete algorithm by means of sorting intra-element conditions and increasing the sharing of nodes. Also, they propose an optimization for changing and removing tokens from the network. It consists of the use of counter memories that record the number of consistently bound tokens in the opposite memory for each incoming token. The advantage is that when the system receives an incoming token, it knows how many tokens must be looked at. PAMELA was implemented only for sequential systems and was simulated on an IBM-PC/AT, which compromises the results and comparisons presented.

## 4. Parallel Execution of Rete

### *4.1. Data-Driven Processing of Rete Network*

Gaudiot and Sohn [7, 36] have investigated the suitability of data-driven processing for the execution of parallel matching in production systems. Data-flow machines are often used exclusively for numerical processing. This happens because in symbolic processing, data structures are irregular and nondeterministic, and the basic entities manipulated are objects or sets of objects rather than numbers. To adapt to such idiosyncrasies of symbolic processing, some modifications have been proposed to the data-flow model: allowing data tokens to carry more information than a single scalar element and adding several simple functional units to each processing element to take care of the fewer primitive functions of symbolic computations (rather than encoding complex functions as numerical computation).

The machine proposed by Gaudiot and Sohn is based on the execution of the Rete algorithm using a data-flow multiprocessor. The advantages of this combination come from the natural match between the algorithm and the architecture: both are driven by data tokens. In the Rete algorithm, multiple comparisons are performed at the same time, while in the data-flow architecture, multiple actors can be fired simultaneously. In both, there are no data modifications, except for arrays, and both rely on the dependency graph obtained from the problem domain. Memorization of partial results and counters for negated nodes required by the Rete algorithm are easily handled in the data-flow architecture.

The Rete algorithm has some inefficiencies that must be overcome in the data-flow machine. The root node has to distribute all the tokens, causing a bottleneck right at the beginning of the processing. The comparisons that take place in a two-input node when a token arrives in either side are accomplished in a sequential fashion. The memory management for two-input nodes takes a substantial amount of time to delete WMEs. This last inefficiency is not addressed by the data flow approach.

Gaudiot and Sohn suggest two possible ways of allocating productions to PE's: redundant and minimum allocations. In a redundant allocation, all patterns are copied and independently allocated. This method reduces communication overhead but consumes a lot of memory. The minimum allocation technique reduces the computation time in matching and reduces the storage space necessary, but it increases the communication overhead. Gaudiot and Sohn chose to use a redundant allocation. To overcome the root distribution bottleneck, they exploited an interesting fact: a WME only matches patterns that have the same number of attribute-value pairs. Therefore it is possible to partition the WMEs and the rule conditions in groups according to the number of attribute-value pairs. The mechanism suggested is able to distribute WMEs simultaneously to different groups, reducing the impact of the bottleneck at the root. A drawback is that considerable speedup will arise only if the WMEs are evenly allocated to groups.

The authors present extensive analysis of a 12-WME, 3-rule example, as well as simulation results and analytical performance evaluation, all based on the same example. Unfortunately, there is no evaluation or comparison with commercial benchmarks to compare the data-driven model proposed with competing approaches. Nonetheless, there are indications that data-flow processing might be promising for parallel matching in production systems.

## *4.2. Multiprocessing of Rete*

Several researchers have attempted to speed up the execution of Rete networks by employing multiple processors. Given that parallel implementation of Rete at the token-passing level leads to too fine a granularity, mappings of Rete to shared data structures that can be processed using a shared-memory multiprocessor have been considered. However, this leads to overheads due to memory access conflicts and synchronization. An alternative is to use a distributed memory multicomputer with message passing. A detailed discussion of the advantages and disadvantages of these two approaches is given in chapter ??. In this section, we concentrate on four parallel implementations that shed some light on how amenable Rete is to parallelization.

### *4.2.1. DRete Algorithm and CUPID Architecture*

Kelly and Seviora [17, 18] present a multiprocessor architecture for supporting comparison level partitioning. They try to solve the problem of preserving match correctness and keeping communication overhead to a manageable level by introducing a distributed Rete algorithm called DRete. DRete implements partitioning at a token-to-token level. It is observed that constant-nodes do not need to be partitioned. Furthermore, to improve load balancing, token-node pairs are moved away from the processing elements that create them. The reasoning behind this action is that there is a high likelihood that pairs created at the same time will be activated at the same time and therefore ought to be located in distinct processor elements. The overhead of this transfer is kept low because the pairs are transferred while the host performs conflict resolution and action phases. Kelly and Seviora also claim that the use of hashing at the node level can lead to a reduced number of comparisons necessary at each node. Instead of doing the comparison with all tokens in the node, it is necessary to compare only with the tokens within a given bucket of the hashing mechanism. Although they have not used hashing in their study, they expect a combination of DRete with hashing to be promising.

The architecture proposed, denominated CUPID, consists of a matching multiprocessor attached to a host. Kelly and Seviora propose an order of hundreds of small processing elements organized as a single two-dimensional array. The communication between the host and the processing elements is realized by a pair of bidirectional trees with the processing elements as the leaves of these trees: one tree is used to broadcast match information to all processing elements, and the other is used to collect responses from them. A processing element is formed by a CPU, a local program ROM, local RAM, local CAM Block, and state machines for communication control. The content of the CAM indicates which nodes of the Rete network are simulated by the corresponding PE.

At the time of the report (1989), the design of the PE was completed and pre-fabrication simulations of the CPU were done. Simulation results of the proposed design indicate a speedup of 7.2 over a VAX 11/785 if 16 PEs are used, and significant increase in the speedup is expected if the PEs become faster.

#### *4.2.2. METE/PIPER*

Working for the Strategic Defense Initiative (SDI), Rowe *et al.* [30, 3] developed the Parallel Inferencing Performance Evaluation and Refinement project (PIPER). This inference engine is based on an extension of Rete called Merit Enhanced Transversal Engine (METE) algorithm. METE/PIPER explores intra-rule parallelism in match processing and conflict resolution, but there is no attempt to exploit multiple rule firing or any other kind of inter-rule parallelism.

PIPER is implemented in the BBN Butterfly Plus computer that consists of up to 256 processor nodes interconnected via a Butterfly Switch. BBN runs the Chrysalis operating system that allows shared-memory access and management. In this implementation, there is a processor called the Inference Manager (IM) that interfaces with the operating system. The remaining processors are used as constant test (CT) processors or as join processors (called TAND processors). A copy of all constant tests is given to each CT processor. The distribution of facts among CT processors is done dynamically by the IM processor, using a round-robin scheme.

In contrast to the CT processors, the join processors are specialized, containing only specific TAND tests. The IM processor is in charge of synchronization at the end of the match phase. Upon receiving an acknowledgement that the last CT token was processed, the IM processor generates startup messages for the TAND processors. The resolution of the conflict set (the process of given priority to fireable rules) is pipelined with the join processing. An average true speed-up of 9.29 over Gupta's results [9] was reported.

Bechtel and Rowe [3] report the development of a tool designed to realize performance analysis. The input to the tool is a rule set to be analyzed. The prediction tool analyzes four factors of interest in estimating the performance of rule sets, namely: length of inference path, interconnectedness of individual rules, rule independence, and number of distinct object types. At the time it was reported, this tool was in an early stage of development. Neither statistical confirmation of the predictions nor calibration of these factors against actual implementation had been performed yet.

### *4.3. Loosely-Coupled Implementations*

Ishida *et al.* [16] propose an organization of distributed production system (DPS) agents to improve the performance of adaptive expert systems and deliver the performance required by real-time expert systems. They point out that in contrast with parallel expert systems, DPSs have no global synchronization for conflict resolution. In a DPS all the rules are fired asynchronously and the interference among rules is avoided by local synchronization between specific agents. The structure proposed in [16] is capable of self-organization, with the agents executing decompositions when the workload is high and performing composition (fusion of agents) when the workload is light. This ability allows the release of hardware resources for future increases in the demand for processing.

Acharya *et al.* [1] explore the possibilities of using message-passing computers to implement the Rete network. They propose a system with a set of processors dedicated to constant nodes, a set of processors for the conflict set resolution, and a concurrent hash-table mechanism operating in a third set of processors to implement the match operation. The major advantage of message-passing computers is the absence of a centralized schedule. However, the static partitioning of the hash-table could cause problems because distinct tokens cannot be processed in parallel if they hash to the same processor pair. The concurrent distributed hash-table allows the activation of distinct Rete nodes to be processed in parallel. Furthermore, it allows the multiple activation of the same node to be processed in parallel.

Other noteworthy parallel implementations of Rete Network include the Encore Multimax using a shared data structure [11]; DADO, using rule level parallelism on a tree structured multicomputer [38, 39]; and PESA, a specialized, hierarchical, pipelined computer that exploits the simpler control of bus-based systems [33]. A good, brief description of these systems can be found in Kuo and Moldovan [22].

Shrobe *et al.* [35] propose a Virtual Parallel Inference Engine that is claimed to be able to isolate the issues of problem representation in a knowledge processing system, such as expert systems, from the issues of executing this system in a particular machine. They present simulation studies for executing production systems on a loosely coupled distributed system and using a Multilisp simulator. They also discuss issues of parallelizing the Rete algorithm. However, they do not include in their study the issue of ordering constraints between rules.

## 5. Compile Time Optimization of Rete

Ishida [13] proposes a compile-time optimization of the Rete network based on the evaluation of a local cost function that takes into consideration statistical measures of previous executions of the system. The measures take into account not only the size of the Working Memory (WM), but also the number of changes to WM. This method uses a cost model based on the number of inter-condition tests and the number of tokens stored in each node. The cost functions have coefficients that are adjusted according to the cost of storing and testing in the particular system being used.

Ishida points out that speeding up a particular rule often implies destroying shared joins and slowing down the overall program. Therefore, the optimization ought to be global. However, the number of possible join structures is an exponential function of the number of rules. A constructive heuristic method that deals first with the most expensive rules is used. This method optimizes the sharing of variables (connectivity constraint), avoids duplication of substructures (minimal-cost constraint), and promotes the use of more efficient structures for rules that are computationally more demanding at run time by using the statistic results of prior runs (priority constraint). The result is a Rete network that minimizes the number of intercondition tests performed over the entire network, as well as the number of tokens passed between nodes. Results indicate a significant improvement over similar man-made optimizations.

## 6. Parallel Rule Firing

Gupta [8] states that large scale parallelism is not appropriate for OPS5-like production systems because changes to WMEs do not have global effects and because large-scale architecture implies small processing elements. Therefore, it is better to map the program onto a few powerful processors. However, recent research, discussed below, has found some ways around this problem by allowing several rules to fire concurrently and thereby avoiding the serial bottleneck of a match-select-act cycle that results in the firing of only one rule.

Parallel rule firing brings some new problems to the creation of expert system machines. With many rules firing in the same cycle, it is necessary to use techniques to ensure that the outcome of the system is correct. By the same token the designer wants to maximize the amount of parallelism and improve the load balance among processor nodes, thereby increasing the hardware usability. Decreasing the internode communication is also important to increase the speed of the system and reduce its cost. Finally, the designer wants a system that is *focused.* That is, the knowledge base should move towards a desired goal state. This problem of focusing the system is also referred to as the *convergence problem.*

## *6.1. Data Dependency*

Firing rules in parallel implies the simultaneous execution of actions of more than one enabled rule. Since these actions change the contents of the Working Memory, and the rules are enabled according to the facts stored in this memory, there might be some compatibility problems between different rules. Kuo and Moldovan [21] identify three different dependencies among rules:

**Inhibiting.** A rule $R_i$ inhibits a rule $R_j$ if firing $R_i$ adds or deletes data elements such that $R_j$ is no longer satisfied.

**Output.** Two rules $R_i$ and $R_j$ are output dependent if firing $R_i$ deletes (adds) a data element which is added (deleted) by firing of $R_j$.

**Enabling.** A rule $R_i$ enables a rule $R_j$ if firing $R_i$ adds or deletes data elements such that $R_j$ becomes eligible to fire.

Inhibiting and output dependencies typically prevent concurrent execution of rules. Enabling dependencies do not prevent concurrent execution but indicate communication between rules and may have an impact on the communication overhead in a parallel implementation. Although they choose different names and notation, Schmolze [32], Kuo *et al.* [20], and Xu and Hwang [42] identify the same set of dependencies.

## *6.2. Correctness*

Correctness becomes an important issue in systems that fire rules in parallel. There are two criteria proposed to guarantee that a set of rules fired in parallel produces a correct result. Ishida and Stolfo [15] propose the *commutativity criterion*, i. e. , a set of rules is parallelizable if and only if any serial execution of the rules produces the same results. Schmolze [32] proposes the *serializability criterion*, i. e. , a set of rules is parallelizable if and only if there is a serial order of rule firing that produces the same results as the parallel one.

While the commutativity criterion requires that *all* possible sequential executions produce the same result as the parallel execution, the serializability criterion only requires that there exist *one* sequential execution that produces the same result as the parallel execution. Therefore the commutativity criterion is much stronger and its conservative nature prevents parallelization of rules. For some real-life systems the use of the commutativity criterion resulted in almost no parallelism possible [32].

The serializability criterion allows much more parallelism but makes it much more difficult for the programmer to check correctness because all possible sequential executions must be checked for correctness. If this criterion is expected to be successful in the future, good tools must be developed to aid the programmer in this work [22].

## *6.3. Rule Partitioning*

Rule partitioning is a major problem in the implementation of parallel production systems on multiprocessors. Oflazer [28] proposes a static partitioning of productions among processors to keep a low partitioning overhead per cycle. Along these lines is the work of Xu and Hwang [42] that uses simulated annealing techniques along with matrix algebra to identify the dependencies among rules and produce a good mapping onto a multiprocessor. Their goals are to maximize parallelism by distributing workload evenly and minimize communication costs in message passing among nodes. The criterion used to decide whether two rules are parallelizable is the commutativity criterion. They use a *communication matrix* to identify whether rules have enabling or output dependencies; a *parallelism matrix* to identify if the rules are compatible; a *distance matrix*, that encodes the distance (number of nodes) between processor nodes in the machine, and thus helps identify communication costs among nodes; and a *firing frequency vector* to encode how often a rule is fired. The output of the simulated annealing process consists of a *configuration matrix* that indicates in which processor node each rule shall be located to achieve the goals.

The cost function used in the annealing process by Xu and Hwang has three independent components. Each of these components represents a cost related to one of the goals: loss of parallelism, load imbalance, or internode communication. At the conclusion of the paper, they state that as expert systems have unpredictable behavior, dynamic load balance is more desirable. The firing frequency vector, which collects run time statistics, is a good instrument for predicting the run-time behavior. This technique allows the method of Xu and Hwang to have a performance close to that of a dynamic distribution of rules among nodes, without paying the cost of slowing down the system with rule distribution at run time.

### *6.4. Compiled Parallel Systems*

Kuo *et al.* [20, 19] introduce the Concurrent Rule Execution Language (CREL). CREL is syntactically equivalent to OPS5.c [6], but instead of using the recency criterion, the conflict set is resolved nondeterministically. The programmer must guarantee that any sequential execution is correct. Kuo defines a mutual exclusion set of rules as a set of rules connected by cycles of dependencies. CREL utilizes static and dynamic dependency analysis. The static analysis identifies mutual exclusion sets and groups rules into clusters. All rules belonging to the same mutual exclusion set are placed in the same cluster. Therefore, there are no dependencies among clusters that can prevent parallel firing of rules in different clusters. The dynamic analysis verifies dependencies among rules within a cluster.

It is desirable to increase the number of clusters, leaving a smaller number of simpler dependencies to be analyzed at run-time. When a static clustering is applied by itself to a CREL program, it generally partitions the program into a single, large cluster. Kuo identifies some optimizing transforms that result in better clustering. The goals of this optimization are breaking interference relations, increasing the number of clusters, and reducing the complexity of the run-time task in each cluster. Listed below are the optimizing transforms identified by Kuo.

**control variable smart.** It is not unusual for an expert system programmer, to use "control variables" to pass "secret messages" between rules with the purpose of enabling only a subset of rules at a given time. When these control variables are identified, the rules can be divided into sets that test for the same values. As rules belonging to different sets will never be enabled at the same time, any dependencies among them can be eliminated, allowing the creation of more clusters.

**propagating constants.** When rules test the same WME for different constant values, or when a rule creates a WME with a constant that differs from a constant tested by the antecedent of another rule, the dependencies between them can be statically eliminated.

**disjoint attribute tests.** In CREL, a WME has a class specification that is used in the construction of the dependency graph. A rule might test and modify a subset of the attribute-value pairs of the class. When the set of attributes modified by one rule is disjoint with the set of attributes tested by another rule, the dependency between the rules due to this class can be eliminated.

**copy and constraining.** If a given attribute has a known finite set of $n$ values, each rule that uses this attribute can be substituted by a set of $n$ equivalent independent rules, each one testing for a constant value.

The run-time checking is twofold. It is necessary to determine which pair of instantiations in the conflict set of a cluster can possibly fire in parallel. This verification is independent of any particular variable binding. After that, it is necessary to check which rules in this subset can fire in the current cycle. This last checking involves variable bindings.

Kuo concludes that because of improvements in the compilation techniques, the focus of attempts to parallelize production systems has shifted. Match is no longer a primary target for parallelization. The most significant performance improvement obtained in the CREL system is derived from run-time checking to allow multiple rule firings in one cycle. This is due to the reduction in the number of cycles that the system executes, reducing the number of synchronization points.

Highland and Iwaskiw [12] argue that a knowledge base can be compiled by restricting the inferencing techniques available, and consequently reducing the expressive power of the language. They implement the High Performance Embedded Reasoning (HiPER) System that compiles not only the match phase but the inference engine as well. HiPER generates a Rete network from the text form of the knowledge base, and this network is traversed to generate procedural code.

## *6.5. Using Meta Rules to Solve Conflicts*

Stolfo *et al.* [40] have created a parallel rule language for production systems called PARULEL. This language is tailored for a system with parallel execution semantics and for the use of redaction meta-rules to solve the conflict set. Redaction meta-rules have instantiations of the actual rules as condition elements, and its actions consist on *redact*, or eliminate, rule instances from the conflict set. The intention is to allow the programmer to dictate which instantiations need to be eliminated from the conflict set to avoid the firing of conflicting rules. Stolfo conjectures that it is inadequate to use OPS5-like languages for parallel production systems, claiming that rather than parallelizing an inherently sequential formalism, one should program using an inherently parallel formalism. This work suggests that the language's execution semantics should be based entirely on parallelism and explicit program sequentialities. The use of *meta-rules* is advocated as a proper mechanism to express control of execution of an underlying nondeterministic formalism.

There are some problems with this approach. The burden of specifying which rule instances should not be fired in parallel is on the programmer. It is unlikely that robust systems will be constructed without good tools to help in ensuring correctness. Another problem occurs when the meta-rules are complex and conflict each other. A possibility of using *meta-meta-rules* exists. However, it is obvious that this road leads to rather complex systems.

## 7. Discussion

On surveying the different initiatives to building expert systems, we have observed that there are four main types of production systems:

- A completely serial system.
- Concurrent match is allowed, but only one rule can fire at a time.
- Parallel Rule Firing with synchronization at the end of the matching phase, followed by generation and resolution of the conflict set.
- A completely distributed system, where there is no generation of a conflict set, and synchronization is strictly local.

The first approach has been abandoned since the first days of research, and it has been accepted for some years that the second approach cannot yield significant speedup. Many research efforts based on the third approach have been conducted in recent years. However, further research is not likely to result in dramatic speed improvements. The generation/solution of the conflict set is a bottleneck because the length of the match-select-act loop cannot be reduced to a period smaller than the lengthiest match operation.

Eliminating global synchronization seems to be very promising and is a current area of research. Issues of rule partitioning and memory partitioning have just started being elaborately addressed. Minimizing the communication overhead in such a system is a challenge. However, it seems to be the most promising approach to the problem of further speeding up expert systems. Similar approaches are being developed in areas not closely related with the expert system research community, such as the development of A-Teams and scale-efficient organizations [27, 41]. We anticipate that some results from these areas might be useful in improving execution of expert systems.

The issue of correctness must be addressed more carefully. There is a conflict between computer architects and system designers on this issue. The problem is that a criterion such as serializability is very convenient for the architectural construction of the machine. However, it results in a heavier burden for the programmer who has to check correctness, making the construction of a robust expert system very difficult. On the other hand, the commutativity criterion that makes programming much easier also makes it almost impossible to find parallelism among different rules. We believe that a compromise might be reached by building systems that use the serializability criterion at the architectural level and provide software tools to aid programmers in correctness verification. The use of meta-rules simplifies the hardware. However, because ensuring correctness is completely the programmer's responsibility, this approach does not seem to be very promising. Furthermore, the solution of conflicts at the meta-rules level is an unsolved problem.

Another important area of research is in Expert Systems languages and design aid tools. Murthy [27] correctly points out that for some problems, it is possible to gather a few hundred heuristics and put together an expert system in a few weeks. However, maintaining these systems during their lifetimes might cost a lot of time and money. One example is the R1 system at DEC that at one time required a few hundred people for its maintenance. Good software engineering techniques need to be adopted in the construction of expert systems to avoid such situations.

Finally we observe that so far, the research done towards speeding up production systems via parallel techniques has been short-lived. This has happened because even the most successful solutions that have been proposed deliver only a small amount of speedup — between one and two orders of magnitude. Considering the design effort and cost of the parallel techniques, this leverage over sequential implementations is too small to guarantee a reasonable useful life for the results of such research to be marketable. Also production systems are quite successfully compiled to run on sequential machines based on cheap, general-purpose microprocessors. This relative success of sequential machines is due to the advances in process technology and in compiling techniques. This observation suggests that a compiled/parallel execution of production systems might be promising.

## References

[1] A. Acharya, M. Tambe, and A. Gupta. Implementation of production systems on message-passing computers. In *IEEE Trans. on Parallel and Distributed Systems*, volume 3, pages 477–487, July 1992.

[2] F. Barachini and N. Theuretzbacher. The challenge of real-time process control for production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 705–709, August 1988.

[3] R. Bechtel and M. C. Rowe. Parallel inference performance prediction. In *Proceedings of the IEEE 1990 National Aerospace and Electronics Conference - NAECON*, pages 21–25, May 1990.

[4] J. M. Crawford and B. Kuipers. Towards a theory of access-limited logic for knowledge representation. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, 1989.

[5] C. L. Forgy. *On the Efficient Implementations of Production Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1979.

[6] C. L. Forgy. *OPS5 User's Manual.* Carnegie-Mellon University, 1981. Tech. Rept. CMU-CS 81-135.

[7] J.-L. Gaudiot and A. Sohn. Data-driven parallel production systems. *IEEE Transactions on Software Engineering*, 16:281–291, March 1990.

[8] A. Gupta. Implementing OPS5 production systems on DADO. In *Proceedings of International Conference on Parallel Processing*, pages 83–91, 1984.

[9] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, March 1986.

[10] A. Gupta, C. Forgy, and A. Newell. High-speed implementations of rule-based systems. *ACM Transactions on Computer Systems*, 7:119–146, May 1989.

[11] A. Gupta, M. Tambe, D. Kalp, C. L. Forgy, and A. Newell. Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programing*, 17, 1988.

[12] F. D. Highland and C. T. Iwaskiw. Knowledge base compilation. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 227–237, August 1989.

[13] T. Ishida. Optimizing rules in production system programs. In *Proceedings of National Conference on Artificial Intelligence*, pages 699–704, August 1988.

[14] T. Ishida. Methods and effectiveness of parallel rule firing. In *Proc. 6th IEEE Conference on Artificial Intelligence Applications*, 1990.

[15] T. Ishida and S. Stolfo. Towards the parallel execution of rules in production system programs. In *Proceedings of International Conference on Parallel Processing*, pages 568–575, 1985.

[16] T. Ishida, M. Yokoo, and L. Gasser. An organizational approach to adaptive production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 52–58, July 1990.

[17] M. A. Kelly and R. E. Seviora. An evaluation of DRete on CUPID for OPS5 matching. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 84–90, August 1989.

[18] M. A. Kelly and R. E. Seviora. A multiprocessor architecture for production system matching. In *Proceedings of National Conference on Artificial Intelligence*, pages 36–41, July 1989.

[19] C.-M. Kuo. *Parallel Execution of Production Systems.* PhD thesis, The University of Texas at Austin, Austin, Texas, 1991. Department of Computer Science.

[20] C.-M. Kuo, D. P. Miranker, and J. C. Browne. On the performance of the

CREL system. *Journal of Parallel and Distributed Computing*, 13:424–441, December 1991.

[21] S. Kuo and D. Moldovan. Performance comparison of models for multiple rule firing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 42–47, August 1991.

[22] S. Kuo and D. Moldovan. The state of the art in parallel production systems. *Journal of Parallel and Distributed Computing*, 15:1–26, June 1992.

[23] H. S. Lee and M. I. Schor. Match algorithms for generalized Rete networks. *Artificial Intelligence*, 54:249–274, April 1992.

[24] B. J. Lofaso. Join optimization in a compiled OPS5 environment. Master's thesis, The University of Texas at Austin, Austin, Texas, December 1988. Department of Computer Science.

[25] D. P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. Pittman/Morgan-Kaufman, 1990.

[26] D. I. Moldovan. Rubic: A multiprocessor for rule-based systems. *IEEE Transactions on Systems, Man and Cybernetics*, 19:699–706, July/August 1989.

[27] S. Murthy. *Synergy in Cooperating Agents: Designing Manipulators from Task Specifications*. PhD thesis, Carnegie-Mellon University, September 1992.

[28] K. Oflazer. Partitioning in parallel processing of production systems. In *Proceedings of International Conference on Parallel Processing*, pages 92–100, 1984.

[29] A. Pasik. *A methodology for programming production systems and its implications on parallelism*. PhD thesis, Columbia University, 1989. Department of Computer Science.

[30] M. C. Rowe, J. Labhart, R. Bechtel, S. Matney, and S. Carrow. Forward chaining parallel inference. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 455–462, December 1990.

[31] J. Schmolze. A parallel asynchronous distributed production system. In *Proceedings of National Conference on Artificial Intelligence*, pages 65–71, 1990.

[32] J. G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13:348–365, December 1991.

[33] F. Schreiner and G. Zimmermann. PESA I - a parallel architecture for production systems. In *Proc. 1986 International Conference of Parallel Processing*, 1986.

[34] O. G. Selfridge. Pandemonium: A paradigm for learning. In *Proceeding of a Symposium Held at the National Physical Laboratory*, pages 513–526, November 1958. Reprinted in *Neurocomputing - Foundations of Research*, edited by J. A. Anderson and R. Rosenfeld, The MIT Press, 1988.

[35] H. E. Shrobe, J. G. Aspinall, and N. L. Mayle. Towards a virtual parallel inference engine. In *Proceedings of National Conference on Artificial Intelligence*, pages 654–659, August 1988.

[36] A. Sohn and J.-L. Gaudiot. A macro actor/token implementation of pro-

duction systems on a data-flow multiprocessor. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 36–41, August 1991.

[37] S. Stolfo, H. Dewan, and O. Wolfson. The PARULEL parallel rule language. In *Proc. 1991 International Conference on Parallel Processing*, pages 36–45, 1991.

[38] S. J. Stolfo, D. Miranker, and D. E. Shaw. Architecture and applications of DADO: A large-scale parallel computer for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 850–854, August 1983.

[39] S. J. Stolfo and D. P. Miranker. DADO: A parallel processor for expert systems. In *Proceedings of International Conference on Parallel Processing*, pages 74–82, April 1984.

[40] S. J. Stolfo *et al.* PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13:366–382, December 1991.

[41] S. N. Talukdar and P. S. de Souza. Scale efficient organizations. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 1458–1463, October 1992.

[42] J. Xu and K. Hwang. Mapping rule-based systems onto multicomputers using simulated annealing. *Journal of Parallel and Distributed Computing*, 13:442–455, December 1991.