Implementation of Production Systems on Message-Passing Computers

Anurag Acharya, Milind Tambe, and Anoop Gupta

Abstract— **In the past, researchers working on parallel implementations of production systems have focused on shared-memory multiprocessors and special-purpose architectures. Message-passing computers have not been given as much attention. The main reasons for this have been the large message-passing latency (as large as a few milliseconds) and high message-handling overheads (several hundred microseconds) associated with the first generation message-passing computers. These overheads were too large for parallel implementations of production systems, which require a fine-grain decomposition to obtain a significant speedup. Recent advances in interconnection network technology and processing element design, however, promise to reduce the network latency and message-handling overhead by 2–3 orders of magnitude, making these computers much more interesting for implementation of production systems.**

In this paper, we examine the suitability of message-passing computers for parallel implementations of production systems. We present two mappings for production systems on these computers, one targeted toward fine-grained message-passing machines and the other targeted toward medium-grained machines. We also present simulation results for the medium-grained mapping and show that it is possible to exploit the available parallelism and to obtain reasonable speedups. Finally, we perform a detailed analysis of the results and suggest solutions for some of the problems.

Index Terms— **Coarse-grain mapping, concurrent distributed hash table, fine-grain mapping, medium-grain mapping, message-passing computers, OPS5, parallel production systems, Rete network, simulation results.**

I. Introduction

PRODUCTION systems (or rule-based systems) occupy a prominent place in Artificial Intelligence. They have been extensively used to develop expert systems spanning a wide variety of applications (e.g., R1/XCON [26], SPAM [27], Weaver [21]), as well as to understand and model the nature of intelligence (e.g., ACT* [2], Soar [23]). Production system programs, however, are highly computation intensive and slow. This limits the utility of these systems both in research and application environments. Further research and development in production systems will require enlarging the production-memory (knowledge bases) in these systems [5], [30], which will exacerbate the problem of long execution times.

Sophisticated compilation techniques and parallelization have been the two main approaches taken by researchers in their efforts to solve the problem. To obtain significant speedup by parallelization of production systems, it is necessary to exploit parallelism at a very fine granularity. For example, the average duration of a task in the implementation described in [17] is a few hundred instructions. Since the first-generation message-passing computers, such as the Cosmic Cube [35], had high network latencies (as large as several milliseconds) and high message-handling overheads (several hundred microseconds), it was not feasible to to use them for exploiting such fine-grained parallelism. As a consequence, researchers focused on special-purpose architectures and shared memory multiprocessors for high-performance implementations of production systems [8], [15], [17], [28], [31], [33]. Recent developments such as *virtual cut-through* [22] and *worm-hole routing* [34], however, have reduced the network latencies for message-passing machines by two to three orders of magnitude to a few microseconds. Similarly, specially designed processors, such as the *Message-Driven Processor* [10], are expected to reduce the message-handling overhead by two orders of magnitude to several microseconds. The emergence of this new generation of message-passing computers makes it interesting to consider them for implementing production systems.

An additional motivation for studying message-passing machines is their easy scalability to a large number of processors. While current production system programs have shown only limited benefit from parallelism [15], we expect the situation to change in the future. For example, production system programs are currently being developed for perceptual and sensory tasks [24], [30] and for consistency maintenance in databases [9]. These systems are expected to show significantly higher concurrency due to a higher rate of change to data memory. Furthermore, researchers are exploring new production system formalisms, some of which admit greater internal concurrency [20] and others which allow explicit expression of parallelism [18]. The systems under development and the systems that will be based on the new formalisms have the potential of benefiting significantly from the scalability of message-passing machines. In fact, since the early results of this work were first reported [1], [16], there have been several other investigations focusing on mapping production systems onto message-passing machines [29], [37].

Manuscript received May 18, 1990; revised August 30, 1991.This work was supported by the Defense Advanced Research Projects Agency (DOD), ARPA Order 4976, Amendment 20, under Contract F33615-87-C-1499, monitored by the Air Force Avionics Laboratory and by the Encore Computer Corporation. A. Gupta is supported by DARPA Contract N00014-87-K-0828 and a NSF Presidential Young Investigator Award.

A. Acharya and M. Tambe are with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

A. Gupta is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305

IEEE Log Number 9200950.

Message passing computers can be classified into two broad categories: *medium-grained* and *fine-grained* [4]. Medium-grained machines typically have between a few and several hundred powerful processors each with several megabytes of main memory. Examples of medium-grained machines are Nectar [3] and Intel iPSC/2 [19]. On the other hand, fine-grained machines may have tens of thousands of processors, each with only a few tens or hundreds of kilobytes of main memory. Examples of fine-grained machines are Mosaic [4] and J-machine [12]. In this paper, we present two mappings of production systems on message-passing computers, one for medium-grained machines and the other for fine-grained machines. Both these mappings assume low network latency and low message-handling overheads. We also present simulation results for the medium-grained mapping showing that it can achieve reasonable speedups. The simulator used was based on Nectar [3], a crossbar based message-passing computer developed at the School of Computer Science, Carnegie Mellon University. We were not able to evaluate the fine-grain mapping since a corresponding simulator for a fine-grain machine was not available to us.

The paper is organized as follows. Section II provides background information on production systems and the Rete algorithm used to implement them. Section III discusses the issues that arise in mapping Rete onto message-passing computers and present mappings for both medium-grained and fine-grained machines. Section IV describes the simulation methodology, the execution traces used to drive the simulation, and the architectural parameters used. Section V presents simulation results and demonstrates the impact of communication overhead on speedup. In Section VI, we identify factors that limit parallelism and suggest solutions for some of them. Finally, we conclude the paper with a summary of the results.

II. Background

In this paper, we concentrate on the parallelism available in OPS5 [7] and OPS5-like languages. These languages are widely used for academic as well as industrial applications. Section II-A describes the basics of OPS5. Section II-B describes the *Rete* algorithm. Rete is the standard algorithm used for both uniprocessor and multiprocessor implementations of OPS5. The mappings proposed in this paper are also based on Rete.

A. OPS5

An OPS5 production system is composed of a set of *if–then* rules, or productions, that constitute the *production memory* (*PM*) and a set of data-items, called the *working memory* (*WM*). The data-items in WM, called *working memory elements* (*wmes*), are structures with a fixed set of named access functions, called attributes, much like Pascal records. Each production is composed of a set of patterns or *condition elements*, corresponding to the *if* part of the rule *(the left-hand side or LHS)*, and a set of actions corresponding to the *then* part of the rule *(the right-hand side or RHS)*. Fig. 1 shows an OPS5 production and its *instantiation* (defined later). The production has three condition elements and one action element. (In the figure, only the text in lower case is part of the OPS5 syntax; the rest are our comments to help the reader understand OPS5 syntax.)

AN OPS5 PRODUCTION

```
(production-name clear-the-blue-block
LHS  (block ^name <block1> ^color blue)
     (block ^name <block2> ^on <block1>)
     (hand ^state free)
                ATTR.TEST
          CE
     -->
RHS  (remove 2))
```

INSTANTIATION FOR THE OPS5 PRODUCTION

```
(
 (block ^name b1 ^color blue)
 (block ^name b2 ^on b1)
 (hand ^state free ^name robot-1-hand)
)
```

Fig. 1. An OPS5 production and its instantiation.

A condition element is composed of a set of tests for a wme's attribute-value pairs. These tests are of two types: *constant tests* and *equality tests*. A constant test checks whether a particular attribute of the wme has a given constant value which is either a symbol or a number. For instance, the test ^*color blue* in the production in Fig. 1 is a constant test. An equality test binds a variable (syntactically, any symbol enclosed in angled brackets is a variable: e.g., ⟨var⟩) to the value of a particular attribute and checks whether the bound value is consistent with all other values bound to the same variable within the production. A condition element may optionally be negated, i.e., preceded with a minus sign (−). A nonnegated condition element is matched by a wme that satisfies all its tests. A negated condition element is matched if there is no wme that satisfies all its tests. If all the condition elements of a production are matched, then the production is said to be *matched*. The set of wmes that conjunctively match a production is referred to as an *instantiation* of the production. Fig. 1 shows an instantiation. The set of all instantiations, active at any given time, is called the *conflict set*.

The execution of an OPS5 program consists of a sequence of computational cycles each of which has three phases—match, resolve, and act. The match phase updates the conflict set with all the new instantiations. The resolve phase uses a selection procedure called *conflict resolution* to choose a single instantiation from the conflict set, which is then *fired*. When the instantiation of a production is fired, the right-hand side *actions* associated with the production are executed and the instantiation is removed from the conflict set. The right-hand side actions may add wmes to working memory, delete wmes from working memory, perform input/output, call a user-defined function or terminate the program. If any changes have been made to the working memory during the act phase, a new cycle is begun; else the program terminates.

B. Rete

The Rete matching algorithm [13] is a highly efficient

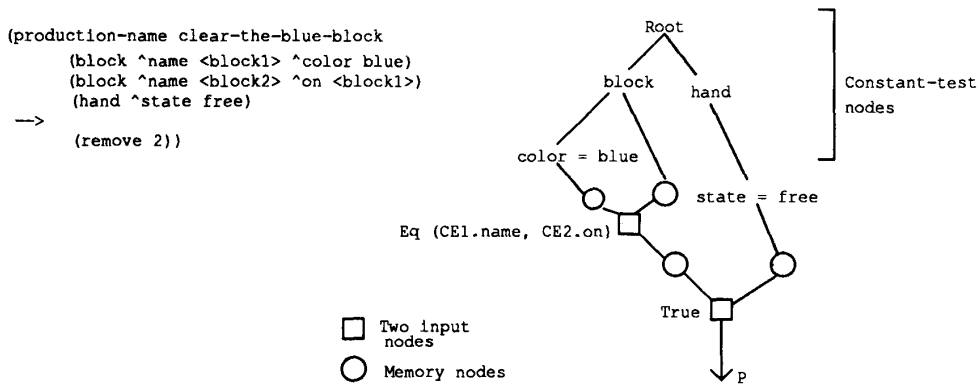


Fig. 2. An example production and its network.

algorithm for the match phase that has been shown to be suitable for parallel implementation [15], [17]. The algorithm gains its efficiency from two sources. First, it stores the partial results of match from previous cycles for use in subsequent cycles, exploiting the fact that only a small fraction of working memory changes on each cycle. Second, it attempts to perform tests that are common to condition elements of multiple productions only once, by sharing them in a directed acyclic network structure, called the *Rete network*.

The Rete network is a special kind of dataflow graph that is compiled from the left-hand sides of productions, before the production system is actually executed. A sample production and the network for this production are shown in Fig. 2. The algorithm performs match by creating and passing *tokens* along the arcs of this network. Tokens are *partial instantiations* of productions. They consist of a *tag*, a *list of wme IDs* (each wme has a unique ID assigned to it), and a *list of variable bindings*. The tag is either a + (plus) or a − (minus) indicating the addition or deletion of the token. The list of wme IDs identifies the wmes matching a subsequence of the condition elements of the production. The list of variable bindings associated with a token corresponds to the bindings for the variables in those condition elements that have already been matched.

There are primarily three types of *nodes* in the Rete network.

1) *Constant-test nodes:* These are used to perform the constant tests of the condition elements and always appear in the top part of the network.
2) *Memory nodes:* These store the results of the match phase from previous cycles as state. This state consists of a *list* of the tokens that match a part of the LHS of the associated production. This way only changes made to the working memory by the most recent production firing have to be processed every cycle. As shown in Fig. 2, memory nodes appear on both sides of a two-input node.
3) *Two-input nodes:* These test for joint satisfaction of condition elements in the LHS of a production. Both inputs of a two-input node come from memory nodes. When a token arrives from the *left memory*, i.e., on the left input of a two-input node, it is compared to each token stored in the *right memory* (and vice versa when a token arrives from the right memory). New tokens are created for those token pairs that have consistent variable bindings. The new tokens flow down the links from this node to its successors.

At the beginning of a match phase, the changes to working memory are introduced as tokens activating the root node (see Fig. 2). They flow through the constant-test nodes, generating tokens that get stored in memory nodes. These tokens then flow into the two-input nodes. The combined activity of storing a token in a memory node and performing the tests at the following two-input node, potentially generating successor tokens, is referred to as a *two-input node activation*. An activation can be either *left* or *right* depending on the memory in which the token is stored.

III. Mapping Production Systems on Message-Passing Machines

This section presents two mappings for production systems on message-passing machines. Section III-A presents the mapping for fine-grained machines (e.g., the J-machine [12]) and Section III-B presents the mapping for medium-grained machines (e.g., Nectar [3]).

A. Mapping for Fine-Grained Machines

One possible, perhaps intuitive, scheme for implementing production systems on message-passing computers arises from viewing the Rete match algorithm in an *object-oriented* manner where the nodes of the Rete network are objects and tokens are messages. This scheme maps a single object (node) of the Rete network onto a single processor. Unfortunately, there are two problems with this scheme:

1) Often the processing of a change to working memory generates multiple activations of the same node of the Rete network. Since each node has been assigned to a single processor, this causes the processing of these activations to be serialized. As a consequence, the processor handling these activations can become a bottleneck.
2) The mapping requires one processor per node of the Rete net. The processor utilization of such a scheme is

expected to be very low. This can be rectified, to some extent, by allocating multiple nodes per processor but this would lead to further serialization since this would cause all activations of *all* nodes assigned to a processor to be processed in sequence.

While the second problem can potentially be solved by sophisticated node distribution heuristics, there is no such solution for the first problem, which is expected to be quite serious in practice. To overcome the limitations of above mapping, we propose an alternative mapping based on a *concurrent distributed hash table* [11]. This hash table is used to store the tokens that would otherwise have been stored in the left and right memory nodes of the Rete network. The hash function that is applied to the tokens uses the variable bindings tested for equality at the destination two-input node, and the unique node-identifier corresponding to the destination two-input node as parameters. This scheme has two advantages. First, it allows the quick detection of the tokens that are likely to succeed the tests at the destination two-input node. Hashing the contents of the associated memory nodes, instead of storing them in linear lists, reduces the number of comparisons performed during individual node-activations [15]. Since most of the time in the match phase is spent processing two-input node activations, this improves the performance of Rete. Second, since the values of the variables being tested are used as parameters, even tokens that flow into the same two-input node but have different values for the tested variables get hashed to different buckets. Since tokens going to different buckets can be processed in parallel, this directly addresses the first problem that was cited for the object-oriented approach. Similarly, the use of the node-id as a parameter results in the distribution of tokens destined for different two-input nodes to different hash buckets allowing them to be processed in parallel. The concurrent distributed hash table thus enables the exploitation of fine-grain concurrency.

The concurrency can be further enhanced by partitioning the hash table into two—a *left* hash table which contains the left tokens and a *right* hash table that contains the right tokens. This allows a left and a right token that are hashed to the same hash bucket to be processed concurrently.

At an abstract level, the computation in the two-input nodes of the Rete network can now be described in terms of operations on the two global hash tables. An activation is processed by:

- Hashing the token and storing it in the indicated bucket (a left token in a left bucket and a right token in a right bucket).
- Scanning the corresponding bucket (bucket with the same index) in the opposite hash table for matching tokens. If any such tokens are found, the successor tokens are generated and routed to the appropriate pair of hash buckets.

From the above, it can be seen that the left bucket and the right bucket of the same index are required together to process an activation. The match phase of the Rete algorithm is thus reduced to token passing between hash bucket pairs.

We now describe how this hash table based version of the Rete algorithm can be mapped onto a fine-grained message-passing machine. A high-level picture of the mapping is presented in Fig. 3. As shown in the figure, the available processors are partitioned into a *control processor*, a small set of *constant-test processors*, a small set of *conflict-set processors*, and a large number of *match processors*. The constant-test processors perform the constant tests specified by the Rete net. The constant nodes of the Rete net are partitioned and assigned to the constant-test processors. The match processors perform the work of two-input nodes in the Rete network. The conflict-set processors collect the instantiations generated and select the instantiation to be fired. The control processor is responsible for evaluating the right-hand side actions.

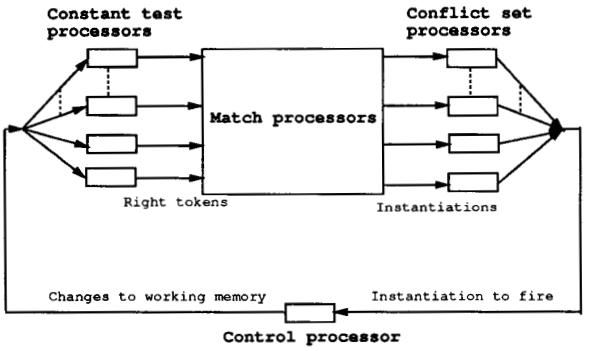


Fig. 3. A high-level view of the fine-grained mapping.

The match processors use the hash table described previously to perform the function of Rete's two-input nodes. These processors are paired up to form a pool of processor pairs. The range of hash indexes is partitioned among the pairs in this pool, i.e., the left and right hash buckets for the same hash index are assigned to the same processor pair. One of the processors in a processor pair is designated as the *left* processor and the other as the *right* processor. The left processor stores, in its main memory, the left hash buckets for the indexes assigned to the pair. Similarly, the right processor stores the right hash buckets for the indexes assigned to the processor pair. Tokens can be sent out from either processor in the pair, but tokens destined for the pair are always sent to the left processor only. This restriction is necessary to avoid the generation of duplicate tokens [15].

A processor pair together performs the activity of a single *node activation*. Consider the case when a token corresponding to the left-activation of a two-input node arrives at a processor-pair. The left processor immediately transmits the token to the right processor. The left processor then copies the token into a data-structure and adds it to the appropriate hash bucket. Meanwhile, the right processor compares the token with contents of the appropriate right bucket to generate tokens required for successor node activations. The right processor then calculates the hash indexes for the newly created tokens, and sends each token to the processor pair that owns the buckets to which it hashed. The activities performed by the individual processors of the processor pair are called *micro-tasks*, and all the micro-tasks that end up on different processors are performed in parallel.

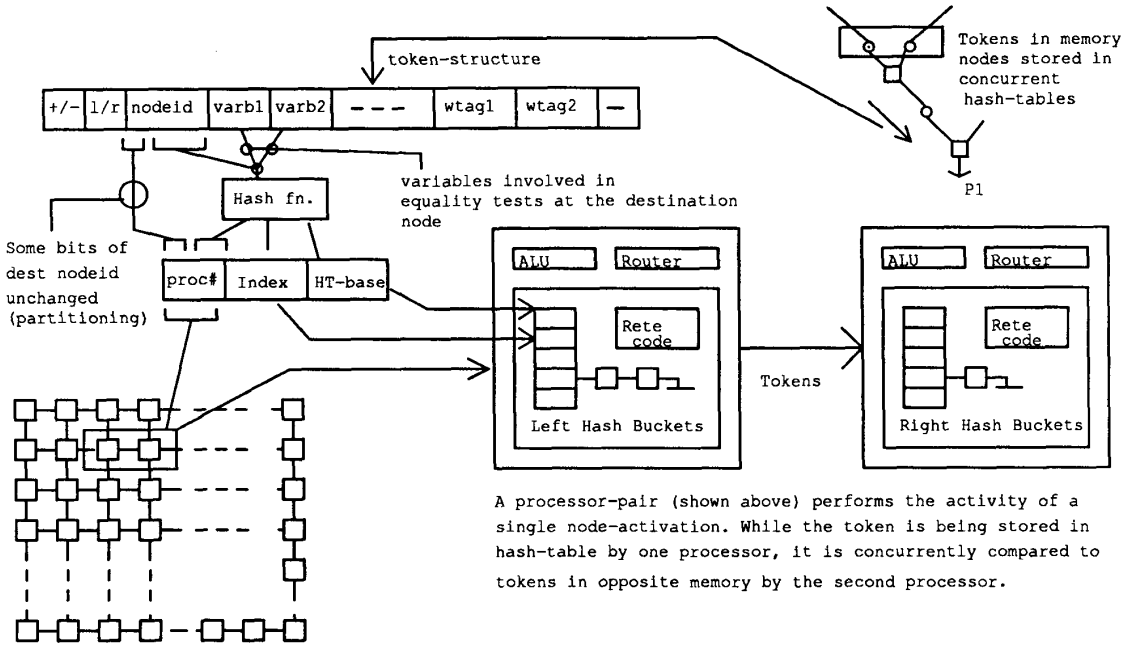


Fig. 4. A detailed view of the fine-grained mapping.

Before the instantiation to be fired can be selected, the system must ensure that the match phase is completed. Detecting termination in a distributed system is a complex problem in itself [25]. For the purposes of this mapping, we chose an acknowledgment-based scheme. When a token is processed by a processor-pair, the processor that stores it keeps a count of the number of successors generated. If no successors are generated for a token, the processor sends an acknowledgment (ack) message back to the processor that generated the token. When a processor receives an ack message corresponding to a successor of a particular token, it decrements the counter corresponding to that token. When the counter corresponding to a token reaches zero, the processor sends an ack message to the processor that generated the token. Thus, after last activation in a match cycle is processed, a single stream of ack messages flows back to the control processor. The control processor then informs the conflict set processors about the termination of the match phase.

The most efficient known implementation scheme for the Rete algorithm compiles the Rete network to procedures that directly implement the tests specified by the network. This scheme achieves its speed at the cost of larger executable images. Since in our hash-table based implementation, a token destined for some node can hash into an arbitrary bucket, each processor must contain the entire executable image. Unfortunately, this can be a potential problem for fine-grained machines, where the memory per processor is small. To handle this problem, we propose two solution strategies that can be used to reduce the memory requirements:

- Partition the nodes of Rete such that any given processor can get tokens from only one partition. This way we ensure that a processor need only store code corresponding to the nodes in its partition. This partitioning is easily achieved if the hash function preserves some bits from the node-id (see Fig. 4). To avoid contention, however, nodes belonging to a single production should be put into different partitions.
- A major cause of the large memory requirement is the inlined code that is used to implement the tests performed at each two-input node. Space can be saved by making a space–speed tradeoff and replacing the inlined code with a small number of function calls.

B. Mapping for Medium-Grained machines

The mapping presented in the previous subsection is suitable for machines with a large number of processors and small amounts of memory per processor. To respond to the smaller number of processors and larger amounts of memory per processor in medium-grained machines, the mapping presented in the previous subsection has to be modified.

1) *Match-processors:* In mapping for the fine-grained machines, a processor-pair is used to process a single node-activation. Since the number of processors in medium-grained machines is much smaller and processor utilization is important, we assign both the left and right buckets to a single processor instead. This modification is facilitated by the larger memory that is associated with each processor.
2) *Conflict-set processors:* During most of the match phase, the control processor is free. Therefore, it is feasible for it to take over the task performed by the conflict set processors in the mapping for the fine-grained machines.
3) *Constant-test processors:* On medium-grained machines,

there is little point in dedicating several processors to constant test evaluation which takes only a small part of the overall time. Therefore, we let all the match processors perform the constant tests. This modification is also facilitated by the larger memory associated with each processor.

A high-level view of this mapping is presented in Fig. 5. The figure shows a *control processor*, responsible for performing conflict-resolution on the instantiations, performing the right-hand side actions and other functions of the OPS5 interpreter. The match processors implement the distributed hash table described previously.

The match procedure proceeds as follows:

1) At the beginning of match phase, the control processor broadcasts the working memory changes generated in the previous cycle to all match processors.
2) All the match processors perform the constant tests. The tokens generated from the constant tests result in several two-input node-activations, most of which are right-activations (see Fig. 2). These tokens are then hashed. If the required hash bucket is not resident in the processor's main memory, then no further processing is done on that token. (Recall that the range of hash indexes is partitioned and assigned to the match processors.) Otherwise, the processor moves on to step 3.
3) For each node activation involving a bucket in the match processor's memory, the token is added to (or deleted from) the bucket. The token is then matched with tokens in the corresponding hash bucket in the opposite hash table. If new tokens are generated, they are hashed and sent to the processor that owns the corresponding bucket.
4) Tokens received by a processor are evaluated exactly as in step 3.
5) All production instantiations are sent back to the control processor. The control processor determines when the match phase has terminated and proceeds on to the resolve phase.

In step 2, work is duplicated across the match processors—all the processors evaluate all constant tests. Since the constant tests take a small amount of time, this duplication of effort is not a significant problem. Further reduction in proportion of time spent in constant tests is also possible [15]. For example, implementing constant tests using hashing can speed them up by a factor of 5.

IV. Simulations

Ideally, we would have liked to evaluate both our mappings. However, we did not have access to a simulator for a fine-grained machine and were able to evaluate only the medium-grained mapping.

The machine model that we assumed for our simulation had the following characteristics:

- The machine has between two and twenty-five processors.
- The time required to send a message is independent of the source and the destination of the message.
- The interconnection network has enough bandwidth for the message traffic.

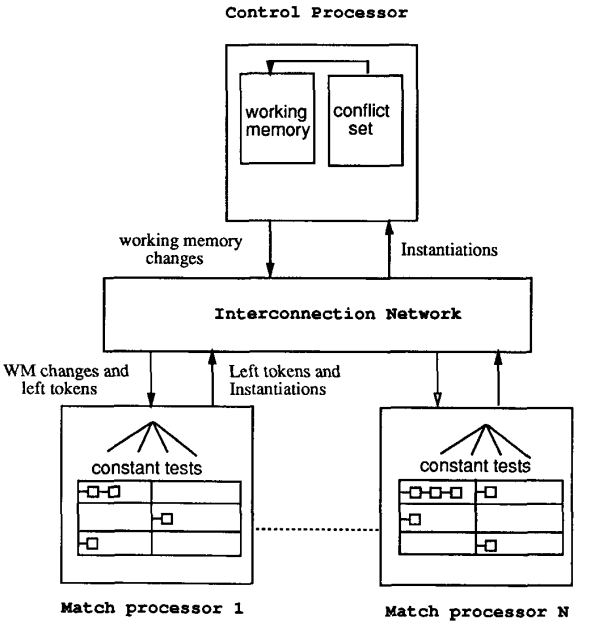


Fig. 5. Mapping for medium-grained machines.

The goal of the simulations was to study the performance of the mapping and to investigate the impact of variation in message passing overheads on the speedups. Hence, the simulations were parameterized by the number of processors and the time required to communicate a message from one processor to another. Since match phase is computationally dominant, we decided to restrict the simulations to just this phase.

The simulator used was based on the simulator for Nectar [6] developed by the Nectar group at Carnegie Mellon. Nectar uses a hierarchical crossbar interconnection scheme. It has low network latencies, low message passing overheads, and high bandwidth. Our simulations are relatively independent of the details of the design of Nectar. In fact, the only Nectar-related assumptions we make are that the time required to send a message is independent of the location of the source and the destination and that the interconnection network has adequate bandwidth for the message traffic.

To perform the simulations, we needed estimates for the cost of individual operations required to process node activations. These estimates were developed using profile information from our implementation of OPS5 on the Encore Multimax [17]. The estimates for the NS32032 processors used in the Multimax were then scaled for the Sparc processors used in Nectar.[1] The cost of processing the constant test nodes was obtained using estimates from [15] (assuming that the constant test nodes are implemented using hashing). The simulator uses the estimates from Table I.

Since we parameterize the simulations by the communication overhead, the absolute values of these estimates are not critically important; rather, it is the ratio of the time spent in communication and the time spent in computation that governs

[1] The Sparcs used in Nectar are the MB86910 from Fujitsu running at 12.5 MHz and rated at 7.5 MIPS [32].

TABLE I
COST ESTIMATES USED FOR THE SIMULATIONS

Operation	Cost Estimate
Perform constant tests	30 μs
Add or delete one left token	32 μs
Add or delete one right token	16 μs
Match token and generate successor (per successor)	16 μs

TABLE II
COMMUNICATION OVERHEADS USED FOR THE SIMULATIONS

Runs	Communication Overhead
Run 1	0 μs
Run 2	8 μs
Run 3	16 μs
Run 4	32 μs

the actual speedups. We divided the time spent in communication into two parts—the interconnection network latency and the software overhead. The interconnection network latency was set at 0.5 μs (this value was provided by the Nectar group) except for the base case for which the interconnection latency was set to zero. The time spent in software overhead was varied and the values used are listed in Table II.

The simulator for Nectar is extremely detailed and slow. As a result, it was not possible for us to simulate whole production system programs. For example, simulation of five match–resolve–act cycles took between two and six hours of cpu-time. Since full program runs consist of between 350 and 650 such cycles, we resorted to simulation of only the characteristic segments of the benchmark programs. Based on previous research [17], we consider match–resolve–act cycles that fall in one of three prominent categories. The first category consists of cycles that have a large number of tokens generated in them. These cycles provide significant concurrency and we refer to them as *large cycles*. The second category consists of cycles that have a small number of tokens (100 or less) processed in them. These cycles are referred to as *small cycles*. The third category consists of cycles that exhibit *cross-product(s)*. A cross-product is generated when a token arriving at a two-input node finds a large number of matching tokens in the opposite memory, thus generating a large number of successor tokens. Overall, rather than presenting speedups for any specific production system, we present speedups for the above categories of match–resolve–act cycles. Based on the mix of such cycles in a specific production system, a qualitative estimate of the speedup for that program can be obtained.

We used a trace-driven approach for the simulation. For the purposes of this study, three traces were chosen. The traces are from the production system programs referred to in [17]. The traces we used are:

1) *Large cycles* (**Rubik**): This segment was taken from the execution trace of Rubik, a program to solve the Rubik's cube. The segment represents four consecutive large cycles.
2) *Small cycles* (**Weaver**): This segment was taken from the execution trace of Weaver, a program to perform VLSI routing. The segment represents four consecutive small cycles.
3) *Cross-product* (**Tourney**): This segment was taken from the execution trace of Tourney, a program that does scheduling for a tournament. One cycle with a large cross-product was chosen from this program. This particular cycle has a large number of tokens that hash to the same bucket and the uneven distribution of tokens is a major problem for parallelization [17]. Four small cycles that surround the cross-product cycle were also included in the segment.

In the rest of this paper, we refer to these segments by the name of the programs from which they were taken—this should not be construed to mean that the numbers presented are for the entire programs.

The input to the simulator consisted of a detailed trace of the activity of the hash table used for the Rete network, corresponding to actual production system runs. These traces were obtained from our implementation on the Multimax. Fig. 6 shows a small fragment of one of the traces used. Given this input, the simulator builds a big hash table like data-structure. The buckets of this table are distributed across the processors in a round-robin manner. The algorithm for medium-grained machines is then simulated on this data structure.

Our simulations have two limitations. First, the traces driving the simulator were obtained from a single processor run. On a parallel machine, however, tokens could be generated in different order and this could lead to a skew in the simulation. However, we expect the distortion to be small since in previous experiments, on shared memory machines [17], we have observed that a change in the order of token generation does not cause a large variation in the speedup. Second, we do not simulate *termination detection*. We assume that termination detection is instantaneous. Given the small number of processors considered in the simulations, we believe that the distortion due to this is also small.

V. RESULTS

Fig. 7 shows the speedups obtained for the three systems with zero communication overhead. This graph shows the best possible speedup, given the proposed mapping. As expected, Rubik has the largest overall speedup. The results from runs simulating a single match processor with zero communication overhead are used as the base case for calculating all the speedups presented in this paper.

Fig. 8 shows the impact of increasing the communication overhead in the three trace segments taken from Rubik, Tourney, and Weaver. The (0 μs) graphs are the ones shown in Fig. 7. The impact of the communication overhead is comparatively low in Rubik (loss of 30% in speedup), somewhat higher in Tourney (loss of about 45% in speedup), and even higher in Weaver (loss of about 50% in speedup). This difference can be explained by Table III, which shows the right and left activations for each trace. Recall that only the left activations

```
Hash bucket = 27
  input token: nodeid = 27, node type = AND, direction = add, left
  output token: nodeid = 121, hash bucket = 77
                nodeid = 121, hash bucket = 143

Hash bucket = 10
  input token: nodeid = 15, node type = AND, direction = delete, right
  output token: --

Hash bucket = 25
  input token: nodeid = 150, node type = NOT, direction = add, right
  output token: nodeid = 75, hash bucket = 77
```

Fig. 6. Trace input to the simulator.

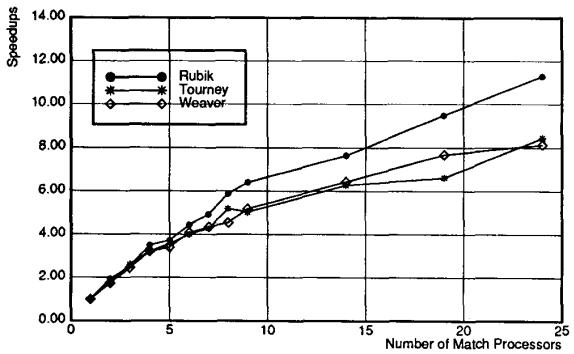


Fig. 7. Speedups with zero communication overhead.

need to be communicated to other processors. Since Rubik has a smaller percentage of left activations (of all activations), it is not affected as much as Tourney and Weaver which have much larger percentages of their activations as left activations.

VI. Analysis

Fig. 8 shows that up to 8–12 fold speedups are available in the three characteristic segments. In this section, we identify some of the factors that limit speedups and present some suggestions for improving the speedups.

A. Slow Generation of Tokens

If there is a single activation that generates a large number of successor tokens at the beginning of a match phase, the processor that handles that activation becomes a bottleneck. This problem manifested itself in the simulation of the Weaver trace which was chosen to study the phenomenon of small cycles. While such dependencies between tokens can occur in any cycle, they become critical in small cycles since there are few other tokens available to keep the processors busy. In one of the cycles from the Weaver trace, majority of the activations (120 out of about 150) are generated by only three left-activations. It is possible to avoid this problem by transforming the Rete network node that generates the successors. The transformation splits the bottleneck node so that the generation of the successors can proceed in parallel. This can be achieved by one or more of the following methods.

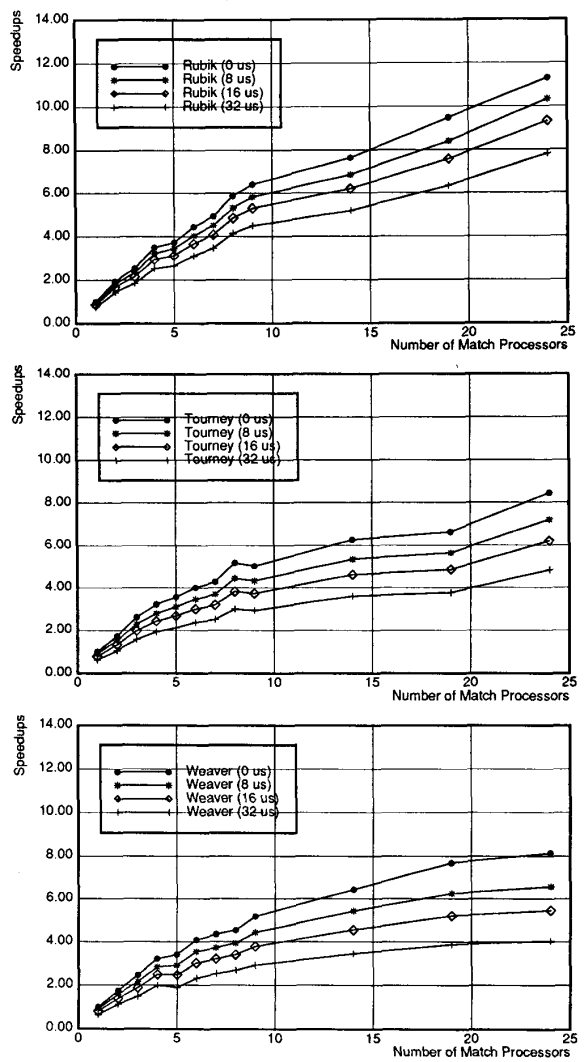


Fig. 8. Speedups with varying overheads: Rubik (top), Tourney (middle), Weaver (bottom).

1) *Unsharing the Rete network nodes* where a large number of successor tokens are generated. For instance, in

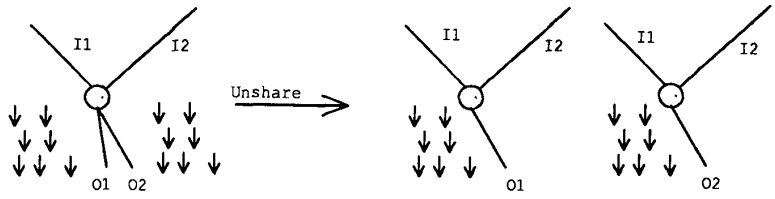


Fig. 9. Unsharing nodes of the Rete network.

TABLE III
TOKENS IN THE SECTIONS OF THE THREE PROGRAMS

Program	Left activations	Right activations	Total Activations
Rubik	2388 (28%)	6114 (72%)	8502
Tourney	10667 (99%)	83 (1%)	10750
Weaver	338 (81%)	78 (19%)	416

Fig. 9, the two outputs O1 and O2 share the two-input node formed by joining conditions I1 and I2. If we unshare in the manner shown, the outputs for O1 and O2 are generated independently, thus removing the bottleneck of generating a large number of tokens from one site. Some work is duplicated, but given the limited number of tokens generated in a small cycle, this duplication should not be a problem.

2) *Replicating the Rete network nodes* where a large number of successor tokens are generated. This is similar, in effect, to unsharing mentioned above. There are several ways of replicating nodes, the most effective of which is *copy-and-constraint* [36]. Using information about the set of values that a variable being tested at a two-input node can take, this method replicates the two-input node. Each of the replicas of the node handles tests for a subset of set of values the variable might take. Each of the replicas thus generates a subset of the activations generated by the original node.

Fig. 10 shows the impact of using the first scheme on speedup achieved for Weaver. As can be seen, there is a substantial improvement.

B. Uneven Distribution of Tokens

Fig. 10 shows that the slopes of the speedup curves decrease as the number of processors is increased. This indicates that the average idle time for a processor increases with the number of processors. This, in turn, suggests an uneven distribution of tokens to the processors. Fig. 11 shows the distribution of left tokens in two cycles for Rubik. The x-axis marks the processor number and the y-axis marks the number of activations evaluated by a processor in a cycle (for instance, processor 1 evaluated approximately 20 activations in both cycles).

In addition to the uneven distribution across the processors in both the cycles, we see a peculiar behavior viz. the processors busy in one cycle are seen to be idle in the next cycle and vice versa. The graphs for Tourney are quite similar. The distribution in Weaver is less uneven.

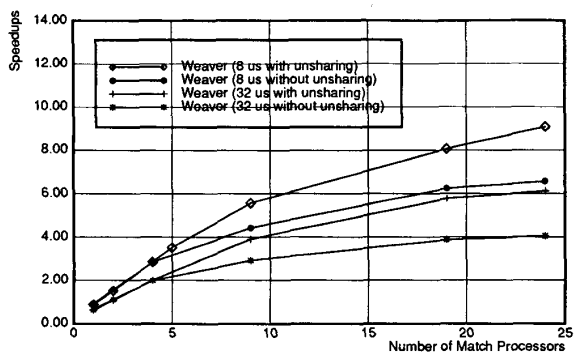


Fig. 10. Speedups with the unsharing.

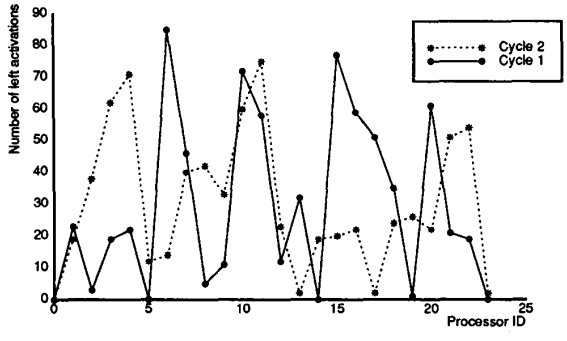


Fig. 11. Distribution of tokens in two independent cycles for Rubik.

The uneven distribution of tokens to processors occurs in two different ways:

1) *Poor distribution of active buckets to processors:* Analysis of the hash bucket traces fed into the simulator showed that in any given cycle, only a small number of buckets are *active*, i.e., have any modifications made to them, and there is little correlation between the active buckets from one cycle to another. The static round-robin bucket assignment algorithm that we use fails to achieve a balanced distribution of active buckets to the available match processors. Typically, this problem occurs for buckets belonging to the left hash table and is manifested in traces from both Tourney and Rubik.
2) *Poor distribution of tokens to buckets:* The same analysis also showed that, in some cases, even though the distribution of active buckets was balanced, a processor turned out to be a bottleneck because a large number of tokens happened to hash to a single bucket belonging to it. This phenomenon had two causes:

- *Cross-Product:* If there is no variable tested at a two-input node, then a token flowing into a two-input node will match all the tokens in the opposite memory, thus generating a large number of successor tokens. All of these tokens are destined for the same two-input node. If this destination node does not test a variable, or if the value of the variable being tested at this node is the same for all the successor tokens, the hashing scheme cannot discriminate between them and all the tokens hash to the same bucket. This happens in the trace segment taken from the Tourney.
- *Multiple-modify Effect*: There are three different ways of causing a change in OPS5 working memory: *add*, *delete*, and *modify*. Rete implements a modify action as a sequence of delete and add actions. Due to the state-saving nature of Rete, once a wme is deleted, all the tokens containing the wme have to be deleted too. Since modify actions usually change only a small part of a wme, the modified wme usually satisfies most of the tests satisfied by the original wme. So, when the subsequent add action is executed, most of the tokens generated hash identically to those that were deleted.

The poor active-bucket distribution suggests a redistribution of the hash table buckets so that the active buckets are distributed evenly among processors. For the simulations, we had originally used a round-robin strategy to distribute the hash buckets to the available processors. The distribution in Fig. 11 indicates the failure of this strategy to distribute active buckets evenly among processors. A *random* distribution of the buckets to the processors was tried as an alternative, but failed to provide a noticeable improvement.

To address the question, "how much improvement could be achieved with a significantly better distribution?," we decided to use an *off-line* algorithm for distributing buckets to processors to determine better distributions. Since determining the optimal distribution of the buckets to processors can be reduced to the multiprocessor scheduling problem [14], an NP-complete problem, we chose a *greedy* algorithm. The algorithm orders the buckets in the decreasing order of activity (activity is defined as the number of tokens processed) in the cycle. It then proceeds down this list assigning each bucket to the processor with the least activity. We used this algorithm to obtain a series of bucket distributions, one per cycle, for every trace. These distributions have a very low variance and hence should be close to the optimal distribution.

These sets of distributions improved the speedups by a factor of about 1.4. While the speedup is significant, we expect this will be difficult to achieve in practice. Run-time load-balancing could possibly be employed but the dynamic cost of migrating the buckets and the associated data will most probably outweigh any improvements. In addition, the irregularity of the workload does not allow an *on-line* computation of a good distribution for the next cycle. Thus, it appears that we cannot take advantage of the existing dynamic load-balancing schemes.

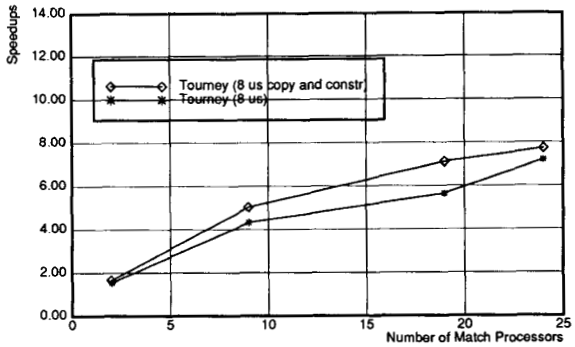


Fig. 12. Speedups with the copy and constraint.

The problem of poor distribution of tokens to buckets can be solved by the copy and constraint mechanism. This is a source-level transformation which splits the productions which suffer from the cross-product effect into multiple copies, each matching only a part of the data the original production matched. This allows additional discrimination to be introduced in the hash function. Fig. 12 shows the speedups achieved with this mechanism.

Another cause for sublinear speedups in the systems, besides the uneven task distribution, is the existence of precedence constraints among the tokens. These constraints are imposed by the structure of the Rete network. Though, some of these constraints can be eliminated by methods described in [15], [17] such as *bilinear networks*, not all can be done away with since they are inherent in the Rete algorithm.

VII. Conclusions

In this paper we have addressed the issue of efficiently implementing production systems on the new generation of message-passing computers which have vastly improved message-passing and message-handling latencies. We showed that the conceptually intuitive mapping where each node in the Rete network is mapped to a single processor has drawbacks and that more sophisticated mappings are needed. We presented one such mapping based on the notion of a concurrent distributed hash table. This mapping was further refined for both fine-grained and medium-grained machines and simulation results were presented for the medium-grained version. The simulations showed that, for a small number of processors, the message-passing machines can provide speedups comparable to those obtained on shared memory implementations [17].

It is interesting to reflect on some of the tradeoffs involved in the message-passing and the shared-memory mappings for production systems. The shared-memory mapping maintains task-queues for the node activations and hash tables for the tokens in shared memory. These centralized data-structures can be potential bottlenecks and their absence is the principal advantage of the message-passing mapping. On the other hand, the message-passing mapping suffers from poor static partitioning of the hash table which results in performance loss due to improper load-balancing. The problem does not

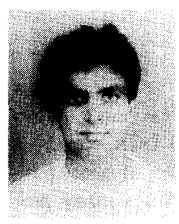
arise in the shared-memory mapping where a processor may pick up a task corresponding to any hash bucket. However, uneven distribution of tokens to buckets is a problem for both the message-passing and the shared-memory mappings, i.e., if multiple tokens are hashed to the same bucket, they are executed sequentially.

ACKNOWLEDGMENT

We would like to thank F. Bitz for building the simulator and P. Steenkiste for several stimulating discussions. We would like thank P. Rosenbloom and A. Newell for many helpful comments on earlier drafts of this paper.

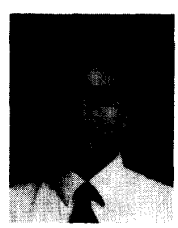
REFERENCES

[1] A. Acharya and M. Tambe, "Production systems on message-passing computers: Simulation results and analysis," in *Proc. Int. Conf. Parallel Processing*, 1989, pp. 246–254.
[2] J. R. Anderson, *The Architecture of Cognition*. Cambridge: MA, Harvard Univ. Press, 1983.
[3] E. Arnould, F. Bitz, E. Cooper, H. T. Kung, R. D. Sansom, and P. Steenkiste, "The Design of Nectar: A network backplane for heterogeneous multicomputers," in *Proc. Third Int. Conf. Architectural Support for Programming Languages Oper. Syst.*, 1988, pp. 205–216.
[4] W. C. Athas and C. L. Sietz, "Multicomputers: Message-passing concurrent computers," *IEEE Comput. Mag.*, vol. 46, pp. 9–24, Aug. 1988.
[5] J. Bachant and J. McDermott, "R1 revisited: Four years in the trenches," *AI Mag.*, vol. 5, no. 3, pp. 21–32, 1984.
[6] F. Bitz, Simulator for the Nectar multiprocessor, personal communication, Comput. Sci. Dep., Mar. 1988. Carnegie Mellon Univ., Sept., 1988.
[7] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Reading, MA: Addison-Wesley, 1985.
[8] P. L. Butler, J. D. Allen, and D. W. Bouldin, "Parallel architecture for OPS5," in *Proc. Fifteenth Int. Symp. Comput. Architecture*, 1988, pp. 452–457.
[9] S. Ceri and J. Widom, "Deriving production rules for constraint maintenance," IBM Res. Division, Tech. Rep. RJ 7348, Jan. 1990.
[10] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a message-driven processor," in *Proc. 14th Int. Symp. Comput. Architecture*, 1987, pp. 189–205.
[11] W. J. Dally, "A VLSI architecture for concurrent data structures," Ph.D. dissertation, Dep. Comput. Sci., California Instit. Technol., Jan. 1986.
[12] W. J. Dally *et al.*, "The J-machine: A fine-grain concurrent computer," in *Proc. IFIPS Congress*, G. X. Ritter, Ed., 1989, pp. 1147–1153.
[13] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.
[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, Freeman, 1978.
[15] A. Gupta, "Parallelism in production systems," Dep. Comput. Sci., Carnegie Mellon Univ., Mar. 1986. Also available in *Parallelism in Production Systems*. Los Altos, CA: Morgan-Kaufman, 1987.
[16] A. Gupta and M. Tambe, "Suitability of message-passing computers for implementing production systems," in *Proc. Nat. Conf. Artif. Intell.*, Aug. 1988, pp. 687–692.
[17] A. Gupta, M. Tambe, D. Kalp, C. L. Forgy, and A. Newell, "Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis," *Int. J. Parallel Programming*, vol. 17, no. 2, 1988.
[18] W. Harvey, D. Kalp, M. Tambe, D. McKeown, and A. Newell, "The effectiveness of task-level parallelism for high-level vision," in *Proc. ACM/SIGPLAN Symp. Principles Practices Parallel Programming*, Mar. 1990.
[19] Intel Scientific Computers, Beaverton, OR, "The iPSC/2 brochures and application software reference material," order number 28 110–001
[20] T. Ishida and S. Stolfo, "Toward the parallel execution of rules in production system programs," in *Proc. Int. Conf. Parallel Programming*, Aug. 1985, pp. 568–575.
[21] R. Jobbani and D. Siewiorek, "Weaver: A knowledge-based routing expert," in *Proc. Int. Conf. Design Automation*, 1985.
[22] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Comput. Networks*, vol. 3, pp. 267–286, 1979.
[23] J. E. Laird, A. Newell, and P. S. Rosenbloom, "Soar: An architecture for general intelligence," *Artif. Intell.*, vol. 33, no. 1, pp. 1–64, 1987.
[24] J. E. Laird, E. S. Yager, C. M. Tuck, and M. Hucka, "Learning in tele-autonomous systems using Soar," in *Proc. NASA Conf. Space Telerobics*, Jan. 1989.
[25] F. Mattern, "Algorithms for distributed termination detection," *J. Distributed Comput.*, vol. 2, pp. 161–175, 1987.
[26] J. McDermott, "R1: A rule-based configurer of computer systems," *Artif. Intell.*, vol. 19, no. 2, pp. 39–88, 1982.
[27] D. M. McKeown, W. A. Harvey, and J. McDermott, "Rule based interpretation of aerial imagery," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-7, no. 5, pp. 570–585, Sept. 1985.
[28] D. P. Miranker, "Treat: A new and efficient match algorithm for AI production systems," Dep. Comput. Sci., Columbia Univ., 1987.
[29] D. Moldovan, "RUBIC: A multiprocessor for rule-based systems," *IEEE Trans. Syst., Man, Cybern.*, vol. 19, no. 4, pp. 699–706, July 1989.
[30] A. Newell *Unified Theories of Cognition*. Cambridge, MA: Harvard Univ. Press, 1990.
[31] K. Oflazer, " "Partitioning in parallel processing of production systems," Ph.D. dissertation, Dep. Comput. Sci., Carnegie Mellon Univ., Mar. 1987.
[32] S. Schlick, personal communication, School Comput. Sci., Carnegie Mellon Univ.
[33] F. Schreiner and G. Zimmerman, "Pesa-1: A parallel architecture for production systems," in *Proc. Int. Conf. Parallel Processing*, Aug. 1987, pp. 166–169.
[34] C. Seitz *et al.*, "The hypercube communications chip," Tech. Rep. Display File 5182:DF:85, Dep. Comput. Sci., California Instit. Technol., Mar. 1985.
[35] C. L. Seitz, "The Cosmic Cube," *Commun. ACM*, vol. 33, no. 12, 1984.
[36] S. Stolfo, "Initial performance of the DADO-2 prototype," *IEEE Comput. Mag.*, vol. 20, no. 1, pp. 75–83, 1987.
[37] J. Xu and K. Hwang, "A simulated annealing method for mapping production systems onto multicomputers," in *Proc. Sixth IEEE Conf. Artif. Intell. Appl.*, Mar. 1990, pp. 130–136.



Anurag Acharya received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1987.

Currently, he is a graduate student at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. His research interests include programming languages, parallel processing, and production systems.



Milind Tambe received the M.Sc. (Tech) degree in computer science from Birla Institute of Technology and Science, Pilani, India, in 1986 and the Ph.D. degree from the School of Computer Science at Carnegie Mellon Universiy.

He is a research associate in the School of Computer Science at Carnegie Mellon Unversity. His interests are in the areas of integrated AI systems and efficiency of AI programs, especially rule-based systems. He has authored or co-authored over 20 refereed journal or conference papers and technical reports in these areas.

Dr. Tambe is a member of the American Association for Artificial Intelligence.

Anoop Gupta is an Assistant Professor of Computer Science at Stanford University. Prior to joining Stanford, he was on the research faculty of Carnegie Mellon University, where he received the Ph.D. degree in 1986. His primary interests are in the design of hardware and software for large scale multiprocessors. He is currently leading the design and construction of the DASH scalable shared-memory multiprocessor at Stanford. He has also worked extensively in the area of parallel applications.

Dr. Gupta was the recipient of a DEC faculty development award from 1987 to 1989, and he received the NSF Presidential Young Investigator Award in 1990.