An Evaluation of DRete on CUPID for OPS5 Matching

Michael A. Kelly and Rudolph E. Seviora
Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

Rising interest in production systems has led to a number of research efforts aimed at increasing their execution speed. The bottleneck in current implementations is the matching required during each production cycle. CUPID is a multiprocessor architecture designed to execute OPS5 matching using DRete, a distributed version of the Rete matching algorithm. This paper describes CUPID and the DRete algorithm with emphasis on correctness and effectiveness in exploiting parallelism in the match operation. The Monkey and Bananas program was executed on a CUPID simulator running the DRete algorithm. The results show that on a technology-independent comparison basis, and in measured execution speed, that the CUPID/DRete combination is several times faster than a commercial uniprocessor, a VAX 11/785, running compiled OPS83.

1. Introduction

With the increasing use of production systems, there is growing interest in improving their execution speed. A number of researchers are working toward this goal, some concentrating on customized uniprocessor designs[Quin85,Lehr86] and others on multiprocessor architectures[Hill84,Stol84,Ramn86,Gupt87,Oshi87].

The work reported here involves the CUPID[Kell87] multiprocessor architecture running a distributed matching algorithm, DRete [Kell87b]. DRete and CUPID were designed in tandem, each taking advantage of characteristics of the other to ensure optimum performance from the union of the two. DRete is based on the Rete[Forg82] algorithm, a many-pattern/many-object matching algorithm designed for the OPS family of production languages running on a uniprocessor. The performance of CUPID executing DRete has been assessed with respect to expected performance and theoretical limitation,and using a test simulation. The simulated program is the Monkey and Bananas problem. It was chosen because it has been used in the past to grade the performance of several machines, although it does not contain a high potential for parallelism. The OPS program used is shown as an example in the OPS83 User's Guide[Forg85]. As a basis for comparison, this program was also run on a VAX 11/785 using a standard OPS83 compiler.

The next section contains a brief review of production languages, the OPS family in particular, and the Rete matching algorithm. The following two sections describe the DRete algorithm and the CUPID architecture. Sections 5 and 6 give details of the simulator and the simulation results and section 7 is a discussion of these results and their implications.

2. OPS Production Languages and Rete Matching

A production system consists of three parts: 1) a set of rules, or productions, defining available operations on a problem state, 2) a set of data elements,or working memory, which describe the current problem state, and 3) a control mechanism which applies the rules to the data elements. Each rule is made up of a condition portion, describing the situation it applies to, and an action portion, describing operations to be performed on the data.

It is the responsibility of the control mechanism to determine which rules apply at a particular time by matching the data elements to the rule conditions. From the result of the match — the conflict set — the most appropriate rule is selected — conflict resolution — and fired. Firing a rule changes the data, which invalidates the match. The match must be recomputed before another rule can be selected and so execution proceeds in a cycle of match, conflict resolution and rule firing phases.

Of the three phases described, the match phase is by far the most computationally expensive[Forg84]. The Rete algorithm reduces the actual number of match operations done by taking advantage of similarities among rule conditions, and the fact that each rule firing generally changes only a small portion of the data set.

Employing the Rete algorithm means compiling the rule conditions into a network of interconnected condition and memory nodes. Condition nodes may have one input for constant value tests, or two inputs to join data elements, subject to consistent variable bindings. Tokens representing changes to the problem state caused by firing a rule are injected into the top of this network. Memory nodes store partial match information for use in subsequent comparisons.

Using the Rete algorithm, the match phase becomes a large set of node activations. At the node level, activation results from the arrival of a token representing one or more data elements on an input arc. A one-input node may or may not pass a token to the next level of nodes depending on the result of the constant comparison. A memory node stores an incoming token and indicates its arrival to the two-input node it is connected to on the next level in the network. A two-input node tests variable bindings between a newly arrived token and each of the tokens stored in the memory node above it on the opposite side. For each successful comparison, a new token will be sent further down into the network. Tokens emerging from the bottom of the network represent fully enabled rules.

3, The DRete Matching Algorithm

The DRete matching algorithm is a distributed version of the Rete algorithm. In DRete the match phase is partitioned at the token-to-token comparison level. The motivation for this method of partitioning, and the expected performance of the resulting algorithm are derived from the data contained in [Gupt83] and [Gupt87b]. This work is an analysis of a number of well known production systems of various sizes. The characteristics of these programs related to communication requirements, and the quantity of data flowing in the match phase are shown in Table 1. The differences between average and maximum values in this table reflect the variability in execution characteristics typical of production systems.

Table 1: Production System Characteristics

Characteristic	avg.	max.
WM changes/prod. cycle	3.0	10
one-input activations/ΔWM	88	122
two-input activations/ΔWM	35.1	62
tokens in a memory node	63	1467
tests/two-input node	0.89	5

DRete is designed to meet the requirements of the average data with the flexibility to manage the maximums. (These data, and the form of DRete, guided the design decisions in the development of the CUPID architecture.)

In DRete, partitioning of one-input nodes is not necessary since their activations do not involve any stored data. The partitioning is done at the two-input nodes where approximately 80% of the match phase time is spent. This partitioning is carried out in two stages. The first step is to isolate the two-input nodes from each other by providing separate memory nodes for each input they are attached to, as shown in Figure 1.

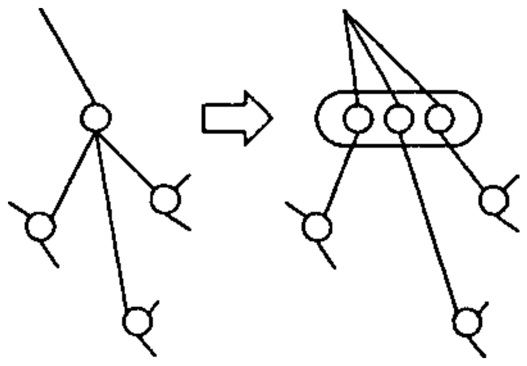


Figure 1: Memory Node Replication

Each two-input node with its two memory nodes is now an independent process with three ports. In the second partitioning step, the two-input nodes are replicated; the number of replicated copies is equal to the number of tokens stored at the two memory nodes connected to it. In this way, each copy of a two-input node is associated with only one data token — either a left or a right token. One node copy on each side is designated the generative copy; it is responsible for storing new tokens. An example of this is shown in Figure 2.

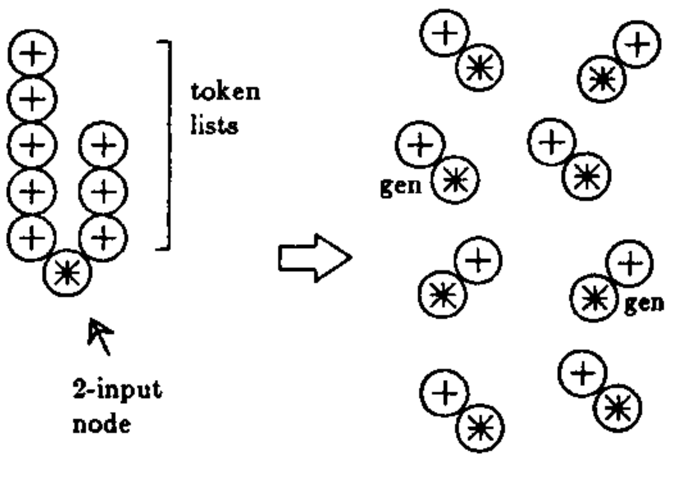


Figure 2: Comparison Node Replication

It is important to note that the size of the image of a two-input node is directly dependent on the amount of data associated with it. Thus, the level of parallelism associated with a node activation is determined by the amount of data it must process.

The node copies are numbered as consecutive pairs; the left side is even-numbered and the right side is odd-numbered. This numbering scheme makes it easy to activate both sides of a node from a single destination address, while allowing for a distinction of which side of the node the node copy represents and to which side the token is arriving.

The operation of the node copies for an AND node is as follows:

1) A positive or negative token arriving at a node copy on the opposite side to its stored token performs a token comparison (if required) between the arriving and stored tokens. It passes a token of the same polarity further into the network if the comparison is successful.

2) A positive token arriving at a node copy on the same side as the stored token will cause the generation of a duplicate node copy with the arriving token attached, if the node copy activated is generative. The generative quality of the activated node copy is then passed to the new node copy. If the node copy activated is not generative, the arriving token is ignored.

3) A negative token arriving at a node copy on the same side as the stored token will cause the deletion of that node copy if it is not generative, or the deletion of only its stored token if the node copy is generative. The negative token will be retransmitted if the node copy had been activated (not necessarily successfully) by a positive token from the opposite side earlier in the same match phase. This second function is performed to insure the correct formation of conjugate pairs by the node as a whole.

For a NOT node, the activity is similar to that described above for the AND node (which keeps node interpretation code to a minimum), but a different node formation is used; it is a two-level structure involving three different types of node copy is used. In the top level, the left node copies (the positive side of the node) store tokens and the right side stores nothing. In the bottom level, only the right node copies, which store tokens potentially blocking the passage of left-side tokens, exist. In a NOT node, the retransmission of negative tokens as described above is not used. Instead, the bottom node copies are put in the path of tokens generated by the top portion of the node. The function of the bottom node copies is to transmit tokens cancelling positive ones sent by the top portion of the node, if data exists which makes the positive tokens invalid.

With two-input nodes partitioned as described, the storing of an arriving token and the comparison of this token with each opposite-side stored token can all be done in parallel.

The level of parallelism available in the match phase as a whole using DRete is related to the product of the number of nodes activated and the number of tokens at each node — with an adjustment for the serial nature of some operations. It is approximately 300, without hashing of tokens, as discussed in [Ke 1187b]. The effect of hashing is to reduce this number, but it provides an overall advantage in execution characteristics, as described in the next subsection.

An important consideration in distributing the match operation is how to balance the workload over a set of processing elements. In DRete, node/token pairs are continually being produced and eliminated as new match state is computed. A dynamic load balancing algorithm[Kell87b] is used to move new node/token pairs away from the processing elements that created them. This increases the probability that node copies activated at the same time are processed in different places.

Recall that new node copies, which are generated during a match phase, are completely independent of each other and their generators. Load balancing, which is done while the host performs the conflict resolution and act phases of the production cycle, consists of transfering the new node copies to other (nearby) processing elements in the multiprocessing environment. Since the generative attribute of a node copy is given to a new node copy before it is transfered to a new processing element, the source of new node copies is not fixed at one processing element.

The load balancing scheme described above was chosen for its effectiveness but also because it is very simple to execute. A more complex load balancing scheme could be too time consuming and so reduce the value of speeding the match phase itself. The simulations revealed that the amount of time spent in the load balancing phases of operation is never greater than 49% of the match time. Since the load balancing phases occur between match phases, overlapping conflict resolution and act phases, they have no serious effect on overall execution speed.

The replication of node and token information and the demands of the load balancing algorithm make a concise encoding of this information very important. The size of a node copy is minimized by storing it as a template to be interpreted by a matching processor. The amount of token information stored is minimized by keeping only those values that are required for matching further down in the network. (For the two program simulated, the average node size is 11.9 16-bit words; the average token size is 6.6 16-bit words. In contrast, the compiled OPS83 versions of this program uses about 250 bytes to encode each comparison node because it generates inline code instead of templates to increase execution speed.)

3.1. DRete and Hashing

Recent research has shown that the number of comparisons between tokens that are necessary during a match phase can be reduced using hashing of tokens[Gupt87b]. Using this method of token partitioning, a node's (node copy's) responsibility during a match phase is reduced from a case where it compares all incoming tokens to all stored tokens to a case where one hash bucket of tokens is compared with a corresponding hash bucket of tokens on the opposite side. (The responsibility for storing tokens is similarly divided.) Comparing an incoming token to only a fraction of the corresponding tokens on the opposite side of the node both reduces the comparisons required and, very importantly, reduces the number of comparisons involved in a single node copy activation. Hashing, however, cannot be used in all cases of two-input node types and cannot always be done in a way which results in equally sized buckets for all data cases.

Using DRete to parallelize the match operation reduces the number of token-token comparisons done during a single node copy activation to one. Over an entire match phase, a node copy's responsibility is to compare all incoming tokens to one stored token. Hashing does not increase a node copy's activation expense significantly, and can be added to DRete at very little increase in storage expense since the size of a node's image is already dependent on the number of tokens with which it is associated. With a combination of DRete and hashing, a node copy's responsibility during a match phase is reduced to matching one hash bucket of tokens to only one other token. Overall, the level of parallelism in DRete is reduced from the 300 previously mentioned, but the net effect is a beneficial one.

DRete with token hashing is potentially much faster than either concept applied alone since it both reduces the total number of comparisons done, and exploits the greatest degree of parallelism available. The work done with DRete to date does not include hashing, but preparations are being made to explore this very promising alternative.

4. The CUPID Architecture

CUPID is a matching processor attached to a host. The host executes the conflict resolution phase of the production cycle and fires the chosen rules. In firing a rule, the host sends changes to the current data set to CUPID. As the new match state is computed, CUPID responds with corresponding changes to the conflict set.

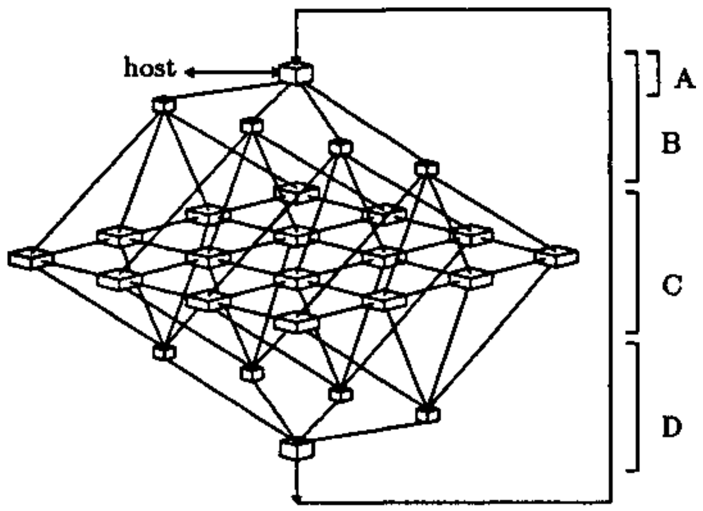
The CUPID architecture is comprised of a large set of small processing elements connected by two independent communication schemes. The number of processing elements in the set is in the order of hundreds to take advantage of the expected level of parallelism available in large programs. The processing elements are arranged as a single two dimensional array, with all node information distributed evenly among them.

One of CUPID's communication systems consists of a pair of unidirectional trees with the processing elements at the leaf positions in both trees. One of these trees is used to broadcast match information to all processing elements over parallel paths. This high speed path ensures fast activation of all network nodes by new match information. The second tree collects the responses of the processing elements to incoming match information. The data path width in this tree widens from serial at the processing element to word-width at the root of the tree to accommodate the expected volumes of data at each level in the tree. Responses which represent data movement within the network of comparison nodes are fed back into the processing elements via a path between the roots of the collection and broadcast trees. Responses representing changes to the set of enabled rules are removed from this stream and sent to a host processor, which awaits such responses from the match phase.

The second communication system is a set of serial links connecting each processing element with its four nearest neighbours. These links support the dynamic load balancing algorithm, carrying node/token pairs from those processing elements that have produced them to less busy processing elements during the inter-match phase interval. Local links are used for this purpose because they match the nature of the transfers: Node copies need not travel further than a nearest neighour and have only one destination. The destination of a node copy is not predetermined; it can be seated and processed anywhere in the processing element array.

Figure 3 is a diagram of the CUPID architecture with a four-by-four array of processing elements. The broadcast tree is shown branching down to the top of each processing element, and the collection tree branching down from the bottom of each processing element. The local interconnect system is also shown, except - for clarity - the links joining opposing edges of the array.

A CUPID processing element consists of a set of functional blocks which combine to satisfy all of the computational requirements of the match operation. Each processing element contains a CPU and local program ROM for interpreting node activations and performing both local and global memory management



A - Filter/Interface Node

B - Broadcast Tree

C - PE Array and Local Network

D - Collection Tree

Figure 3: The CUPID Architecture

4.1. The CUPID Processing Element

functions, a local RAM area to store node/token pairs, and a CAM block with entries indicating which nodes from the original network are represented at this processing element. A processing element also contains a set of state machines for handling data transfers over the various communication ports. One state machine conditionally stores information reaching the processing element from the broadcast tree in RAM. The decision to store or ignore arriving match information is based on the CAM contents. An output state machine is used by the CPU to transfer the results of node activations onto the collection tree. From there, data is distributed to either the host, the processing element array via the broadcast tree, or both. A set of five state machines handle incoming and outgoing node copy transfers over the local links. Interactions between these state machines and the CPU are managed by an interrupt protocol.

A complete design of the CPU and ROM portion of the processing element has been done using an NCR two micron CMOS standard cell library. Preliminary designs of the individual communication state machines have also been done.

The processor[Bond 88] is a three stage pipeline Harvard architecture design with a 16-bit wide data path. It is a RISC processor, satisfying the relatively simple processing requirements of the match operation, and the need for a compact design. It supports two levels of interruption — required by the communication mechanisms — and two special instructions. One of these instructions is a variable condition branch instruction used in place of a sequence of fixed condition branches. This instruction benefits pipeline operation by reducing the number of branches executed. The second special instruction is included to assist in traversing the linked list structure used in storing node/token pairs in RAM. The increased complexity of the processor caused by including these instructions is offset by a reduction in code size, resulting in an overall reduction in the size of a processing element. The CPU design required 1500 cells; the processing element minus memories is comparable in complexity to a commercial 8-bit microprocessor.

Prefabrication simulations of the CPU show a peak execution rate of 8.3 mips. Dynamic measurements of instruction frequency from simulations of CUPID running DRete were used to determine memory reference frequency and branching characteristics during matching. It is estimated that the CPU will execute the required software at 6.7 mips.

4.2. VAX 11/785 Mips

In order to compare CUPID results with those obtained on a VAX 11/785, it is necessary to establish the relative power of their respective processing units. As stated above, the CUPID CPU executes the DRete code at 6.7 mips. To normalize this to the VAX CPU speed, measurements were taken of the VAX running instructions corresponding to those in the instruction set of the CUPID CPU. The speed of the VAX executing each of these instructions is combined with the frequency of their occurrence in the CUPID processing element code to establish a speed for a VAX 11/785 running CUPID software. Table 2 shows the execution speeds for four classes of instructions on the VAX and their relative frequencies in CUPID software. Instructions were run as part of a long stream to ensure that the processor pipeline was full during their execution, except, of course, for the case of the successful branch. (The memory access instruction speed was calculated by multiplying the speed of a memory update on the VAX by the equivalent number of instructions for a CUPID CPU performing the same function.)

Table 2: VAX 11/785 Instr. Speed and Freq.

Instr. Class	Speed [Mips]	Relative Freq.
Mem. Access	2.1	0.312
Register	4.0	0.494
Branch (fail)	2.9	0.101
Branch (pass)	1.5	0.093

From this data it is calculated that the VAX 11/785 would execute CUPID software at 2.7 mips. This is higher than the 1.5 mips rating for the machine and results from the elimination of complex

instructions from the test situation.

Table 3: CUPID Performance vs. VAX 11/785

Array Size	1	2	4	8	16	32	64
VAX Match Time [mS]	94						
CUPID Match Time [mS]	22.4	13.4	8.9	6.4	5.3	4.4	4.1
CUPID Speedup	4.2	7.0	10.6	14.7	17.7	21.4	22.9
Norm. CUPID Match Time [mS]	55.4	33.1	21.9	15.8	13.0	10.9	10.1
Norm. CUPID Speedup	1.7	2.8	4.3	5.9	7.2	8.7	9.3
CUPID Match Time [mS] Tcpu=0	0.24						
CUPID Speedup Tcpu=0	391						

5. The Simulator

The simulator was written using N.2[Endo87], a commercially available package. Register transfer level simulations of CUPID running DRete have been done to establish a coherent software structure for a processing element. This includes both the match and load balancing algorithms as well as memory management for all state machines. The results of these simulations were important in terms of verifying the basic DRete and CUPID concepts, but are too time consuming to run all but the simplest of tests. The hierarchical nature of the simulator made it possible to move to a higher level of simulation using process times and communication timing information from the original simulator. This allows the simulation of larger test programs on arrays of 1-64 processing elements. The accuracy of the higher level simulator was verified by comparing its responses to those obtained from the lower level simulations. The higher level simulator was tuned to agree with the lower level simulator to within 1-2% between individual responses, and to within 0.2% over long (mixed) strings of process activations.

It is by incorporating timing values from the standard cell implementation of the processing element's components into the simulator that a direct comparison of CUPID execution time to that obtained on a standard commercial machine, in this case a VAX 11/785 running compiled OPS83, was possible.

In the simulations, only the match phase is simulated; a dummy process is used in place of the host processor to interact with the processing element array. Changes to working memory caused by rule firings are presented to the processing element array in the order established by running the OPS83 version of the same program.

6. Simulation Results

The Monkey and Bananas benchmark executes in 117 mS on a VAX 11/785 with all output disabled. Estimating the match phase as 80% of this execution time[Forg84], it takes 94 mS.

Table 3 shows the match execution times for the Monkey and Bananas program on the VAX and on CUPID for array sizes of 1-64 processing elements. Below these are the speedup factors of CUPID over the VAX. The next line in the table shows the estimated match time of CUPID with a CPU normalized to the speed of a VAX. (This normalization does not take into account that communication time becomes less of a factor in the total match time as processor speed decreases.) The next line in the table shows the speedup obtained over the VAX on the normalized CUPID.

Because processor speed will continue to increase as technology advances, it is a matter of theoretical interest to calculate the potential speed of CUPID as its CPU clock speed approaches infinity. Speed is then determined by the effectiveness of the communication system and the quantity of data which must be exchanged between processing elements. The numbers for the speed of CUPID in this limiting case are shown at the bottom of Table 3; they were calculated assuming that the communication clock remains at 10 MHz (although this too will increase with time) but that serial paths are widened to word width.

7. Conclusions

The CUPID matching time with one processing element, from Table 3, shows that the CUPID processing element can execute the operations required by the match phase very effectively. This contrasts with other research efforts where a parallel algorithm substantially reduces match speed for the uniprocessor case. The speed of a single processing element is multiplied by the effect of using greater numbers of these processing elements in the CUPID array.

It should be noted that the Monkey and Bananas problem does not contain a high degree of parallelism and, in fact, is fairly sequential in nature. The diminishing returns from increasing the CUPID array size to 64 processing elements is an indication that the limit of parallelism is being reached. Further simulations are under way, using programs which contain higher potentials for parallel execution, to better show the ability of the DRete/CUPID combination to speed match execution.

An 23 fold increase in speed is observed in the eight-by-eight processing element array for the Monkey and Bananas program despite the low availability of parallelism. This verifies that a fairly simple processing element implemented in a modest technology can provide sufficient processing power for production system matching. At the same time, it verifies that the DRete and load balancing algorithms do not introduce an inordinate amount of overhead into the match operation.

The increase in match execution speed provided by the DRete/CUPID combination results in a substantial increase in the overall production system execution speed. Executing the match phase on a computational element separate from the one executing the conflict resolution phase (the host) has the added advantage that conflict resolution can be performed partially in parallel with the match phase. Responses from the match phase can be incorporated into the conflict resolution calculation as they arrive throughout the match phase. The result of this is that the match phase still remains the dominant factor in production system execution time and so increasing its speed further merits additional effort.

References

[Bond88] Bond, D.C., Seviora, R.E., A Standard Cell Implementation of a RISC for Rule-Based Computing, submitted.

[Endo87] N.2 User's Manual, Endot Inc., Cleveland, Ohio, 1985.

[Forg8l] Forgy, C.L., OPS5 Users Manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, 1981.

[Forg82] Forgy, C.L., Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19, 1982, pp. 17-37.

[Forg84] Forgy, C.L., Gupta, A., Newell, A., Wedig, R., Initial Assessment of Architectures for Production Systems, *Proceedings AAAI-1984,* 1984.

[Forg85] Forgy, C.L., OPS83 User's Manual and Report, Production Systems Technologies Incorporated, 1985.

[Gupt83] Gupta, A., Forgy, C.L., Measurements on Production Systems, Technical Report CMU-CS-83-167, Department of Computer Science, Carnegie-Mellon University, 1983.

[Gupt87] Gupta, A., Forgy, C.L., Kalp, D., Newell, A., Tambe, M., Results of Parallel Implementation of OPS5 on the Encore Multiprocessor, Technical Report CMU-CS-87-146, Department of Computer Science, Carnegie-Mellon University, 1987.

[Gupt87b] Gupta, A., Parallelism in Production systems, Pitman Publishing, 1987.

[Hill84] Hillyer, B.K., Shaw D.E., Execution of 0PS5 Production Systems on a Massively Parallel Machine, *Journal of Parallel and Distributed Computing 8,* 1986.

[Kell87] Kelly, MA., Seviora, R.E., A Multiprocessor Architecture for Production System Matching, *Proceedings AAAI '87*, 1987, pp. 36-41.

[Kell87b] Kelly, MA., Seviora, R.E., DRete - A Distributed Matching Algorithm, 23 pages, submitted.

[Lehr86] Lehr, T.F. The Implementation of a Production System Machine, *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences,* 1986, pp. 177-186.

[Rile86] Riley, G.D., NASA Memo FM7(86-5l): Timing Tests of Expert System Building Tools, 1986.

[Oshi87] Oshisanwo, A.O., Dasiewicz, P.P., A Parallel Model and Architecture for Production Systems, *Proceedings International Conference on Parallel Processing,* IEEE, 1987.

[Quin85] Quinlan, J. A Comparative Analysis of Computer Architectures for Production System Machines, Technical Report CMU-CS-85-178, Department of Computer Science, Carnegie-Mellon University, 1985.

[Ramn86] Ramnarayan, R., Zimmermann, G., and Krolikoski, S., PESA-1: A Parallel Architecture for OPS5 Production Systems, *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 1986,* 1986, pp. 201-205.

[Stol84] Stolfo, S.J. Five Parallel Algorithms for Production System Execution on the DADO Machine, *Proceedings National Conference on Artificial Intelligence,* 1984, pp. 300-307.