CUCS-118-84

Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE

Daniel P. Miranker

Department of Computer Science
Columbia University
New York City, N. Y. 10027

April 9,1984

Abstract

DADO is a highly parallel, VLSI-based, tree structured computer, intended for the rapid execution of production system programs. In this paper we describe a new match algorithm for executing production systems on DADO that is capable of handling both temporally redundant and nontemporally redundant production systems. We argue that the new algorithm is faster than the original DADO algorithm intended for nontemporally redundant systems. We also show that the new algorithm executed on parallel hardware is faster and more space efficient than parallel implementations of the RETE match algorithm, which is appropriate for temporally redundant systems.

This research is supported cooperatively by: Defense Advanced Research Projects Agency under contract N00039-82-C-0427, New York State Science and Technology Foundation, Intel Corporation, Digital Equipment Corporation, Valid Logic Systems Inc., Hewlett-Packard, Bell Laboratories and International Business Machines Corporation.

1 Introduction

DADO is a highly parallel, VLSI based computer comprising a large number of processing elements (PE's) interconnected in a complete binary tree. Adjacent to the root of the DADO tree is a conventional coprocessor which acts as a file server and performs the usual activities of a host. Thus, DADO may be viewed as a peripheral device of a conventional machine. Communication may occur between PE's in the DADO machine along the tree edges. In addition any PE in the DADO machine may broadcast data to all of its (logically) connected descendants in the tree, or may be instructed to report a value to all of its ancestors.

In the DADO 1 prototype, now operational at Columbia, there are 15 PE's each composed of an 8 bit processor, a ROM resident operating system, 8K bytes of RAM, and an I/O section. The DADO 2 prototype presently under construction utilizes 16K bytes of RAM at each of 1023 PE's. Under the control of software, a PE may operate in one of two modes: *master* or *slave*. In master mode the PE runs a computer program stored in its local memory. However, instructions embedded within the master's program may be broadcast to descendant PE's operating in slave mode for immediate execution. Each of the slave PE's execute the instruction on different data stored in there local RAM in a manner similar to an array processor, or the ILLIAC IV (Lowrie, 1975). This type of parallelism is known as single instruction stream multiple data stream (SIMD) execution. Furthermore, the machine can be arbitrarily partitioned into a number of independent subtrees. The root of such a subtree logically disconnects itself from its parent and becomes the master of the PE's logically connected below. This type of machine has become known as a multiple SIMD (MSIMD) architecture (Flynn, 1972).

The DADO machine has been designed as a special purpose processor capable of achieving significant performance improvements in the execution of production system programs.

A production system (PS) (Newell, 1973) is defined by a set of rules, (or productions), and a collection of dynamically changing facts, called the *working memory* (WM). A rule in a production system consists of a left hand side (LHS) and a right hand side (RHS). The LHS is a collection of condition elements to be matched against the contents of the WM. The RHS contains actions effecting changes in the WM. A production system repeatedly executes the following cycle of operations:

1. Match: For each rule, compare the LHS against the current WM. Determine if the WM satisfies the LHS.

2. Select: The set of satisfied rules is called the *conflict set*. Some subset of the conflict set is chosen according to some predefined criteria.

3. Act: Add to or delete elements from the WM as specified by the RHS of the selected rules.

An example rule using the OPS5 production system language (assumed to be familiar to the reader) (Forgy, 1981) is shown in figure 1. The pair of WM-elements matching the condition elements is called an instantiation of the rule.

```
(p categorize-job-sizes                                  ; rule name
 (message ^job <x> ^size <y> ^status new)                ; condition element,
                                                         ; <x>,<y>
                                                         ; are pattern variables
 (class-definition ^size <y> ^class-name medium)         ; condition element
-->
 (make job ^job-name <x> ^class medium)
 )

This rule says if
       there is a WM-element in the system representing a message about a new job,
       and the job's size matches the class definition for medium size jobs,
then
       create a new WM-element tagging the job with the class name medium.
```

Figure 1: An Example Production Rule.

1.1 Temporal Redundancy

A distinguishing property of production systems is **temporal redundancy**. A PS is considered temporally redundant if on each cycle few changes to WM are made. R1/XSEL, which incrementally builds a solution to the VAX configuration problem (see (McDermott, 1982)), is typical of temporally redundant PS's. Systems which search through large databases, such as ACE (Stolfo and Vesonder, 1982), or sensor based systems as would be found in a robot, tend to be nontemporally redundant.

2 The Original DADO Algorithm

The approach to the parallel execution of PS's on the DADO computer is to logically divide the tree into three distinct components. One of these components consists of all PE's at a particular level within the tree, called the PM-level. The PM-level delimits an upper and lower portion of the tree (see figure 2).

A *fine grain* DADO computer would be one with perhaps a hundred thousand very simple PE's, each having 1 to 2K bytes of memory. For a fine grain DADO the original DADO algorithm (Stolfo and Shaw, 1982) suggested each PE at the PM-level be used to store a single production. The portion of working memory relevant to the rule contained in the PM-level PE is distributed uniformly in the subtree below. A working memory element is considered relevant if it satisfies the

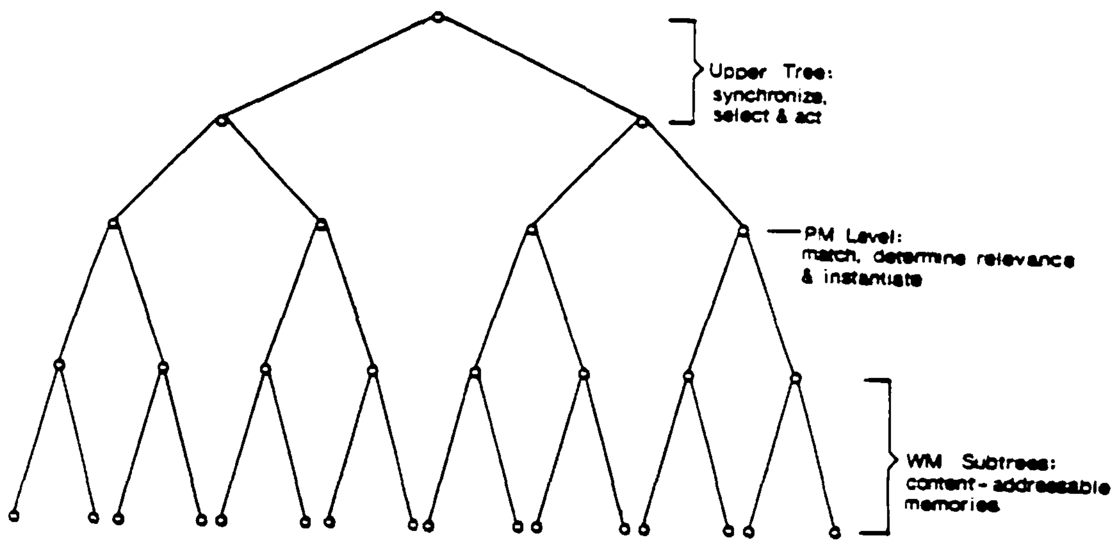
Upper Tree: synchronize, select & act
PM Level: match, determine relevance & instantiate
WM Subtrees: content-addressable memories

Figure 2: Functional Divisions of the DADO Tree.

constants and the intra-condition (Forgy, 1982) restrictions of any condition element in the rule. The subtrees formed by the PM-level are to be considered as a collection of independent parallel associative processors (Foster, 1976) providing parallel access to the WM-elements.

The upper portion of the tree is used for synchronization and selection. The details of the original algorithm are described in the following abstract algorithm.

During the act portion of the production system cycle, additions to WM are performed by broadcasting the WM-element to all the PE's in the tree. The PM-level PE's determine if the WM-element is relevant to their rule. If so, the WM-element is stored in an available PE. Deletions from WM are processed by broadcasting the WM-element to all PE's in the tree. The PE's then compare the broadcast element to the element in their local store. If it is the same the PE marks itself as free.

During the match phase, each PM-level PE enters master mode and broadcasts to its slave PE's the first condition element of the rule. The slave PE's compare the pattern against the working memory elements and report to the PM-level master, the success of the match and the bindings of any variables in the condition element. Each variable binding is then substituted in the remaining condition elements, and the match routine is called recursively for the remaining condition elements.

Since no state is saved between cycles, and the algorithm exploits massive parallelism during the match phase, the original algorithm is considered to be best suited for nontemporally redundant production systems where many changes to WM occur on each cycle of execution.

```
1. Initialize: Distribute a match routine and a partitioned subset of rules to each PM-level
   PE. Set CHANGES to the initial WM elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do;
    a. Broadcast WM-change to the PM-level PE's and an instruction to match.
    b. The match phase is initiated in each PM-level PE:
        i. Each PM-level PE determines if WM-change is relevant to its local set of
           rules by a partial match routine. If so, its WM-subtree is updated
           accordingly. [If this is a deletion, an associative probe is performed on the
           element (relational selection) and any matching instances are deleted. If this
           is an addition, a free WM-subtree PE is identified, and the element is
           added].
       ii. Each condition element of the rules stored at a PM-level PE is broadcast to
           the WM-subtree below for matching. Any variable bindings that occur are
           reported sequentially to the PM-level PE for matching of subsequent
           condition elements (relational equi-join).
      iii. A local conflict set of rules is formed and stored along with a priority rating
           in a distributed manner within the WM-subtree.
    c. end do;
4. Upon termination of the match operation, the PM-level PE's synchronize with the upper
   tree.
5. Select: The max-RESOLVE circuit is used to identify the maximally rated conflict set
   instance.
6. Report the instantiated RHS of the winning instance to the root of DADO.
7. Set CHANGES to the reported action specifications.
8. end Repeat;
```

Figure 3: Original DADO Algorithm.

3 The RETE Match

A *medium grain* implementation of DADO as opposed to a *fine grain* implementation would comprise on the order of tens of thousands PE's, each made up of a state of the art processor circuit with about 10K bytes of local memory. A medium grain DADO permits a parallel implementation of the RETE match.

The RETE algorithm compiles the left hand sides of the production rules into a discrimination network. Changes to Working Memory serve as the input to the network. The network in turn reports changes to the conflict set. There are two primary categories of nodes in the match network; test nodes and memory nodes. When a working memory change enters the network a "plus" or a "minus" sign is appended to the working memory element indicating whether the element is to be added or deleted from memory. A pointer to the change, called a token, is then replicated and passed to a number of entry points into the network.

The RETE algorithm first compiles into the network sequences of tests which perform partial matches of condition elements. These tests are called single input tests since they consider only one attribute of a condition element and one token at a time. Thus each node has only a single arc entering and leaving the node. The match network for the rule in figure 1 is shown in figure 4.

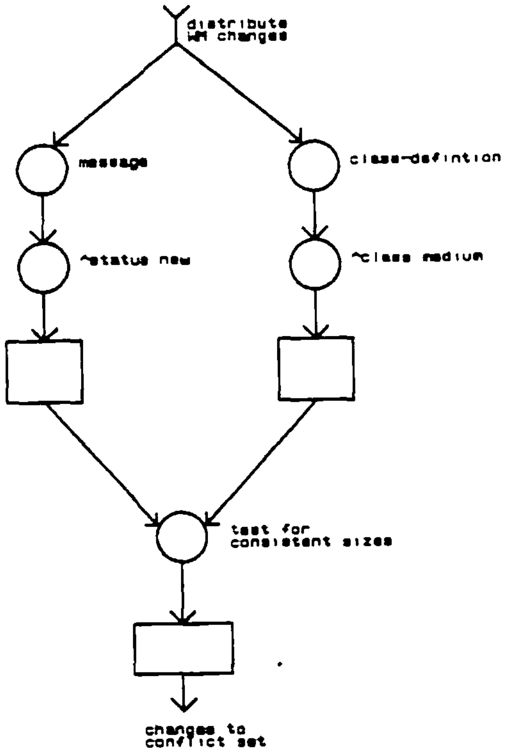

Figure 4: RETE Match Network for the Rule in Figure 1.

Subsequently the RETE algorithm generates intercondition test nodes, called two input nodes. Within two input nodes pattern variables are bound between two condition elements. Associated with each input arc of a two input test node is a token memory. "Plus" tokens that have satisfied the one input tests and are proceeding on to a two input test are added to a memory node called an alpha-memory. Note that the alpha-memories contain precisely those WM-elements that are *relevant* to a particular condition element. Minus tokens have reached an alpha-memory node have a corresponding plus token already present in the alpha-memory. The corresponding plus token is removed.

When a token enters a two input node it is compared against the tokens in the memory on the opposite arc. Any tokens which have consistent variable bindings are paired with the first token, to form a new token that propagates through the network. Token memories that store paired sets of tokens are called beta-memories. Tokens that propagate from the last two input nodes reflect changes to the conflict set.

4 The TREAT Algorithm

The original DADO algorithm does not save any state across production system cycles. In a temporally redundant PS, where few WM changes are made on each cycle, the original algorithm must recompute many comparisons of WM. The opposite is true of the RETE algorithm. The RETE algorithm saves sufficient state in the match network to guarantee that no comparison of two working memory elements is recalculated at a later cycle. If large changes to working memory are made, a large overhead is incurred maintaining the state information.

The Temporally REdundant Associative Tree algorithm (TREAT) for production systems on DADO attempts to synergistically merge the advantages of the two aforementioned algorithms. The approach of the TREAT algorithm exploits the observation that most of the state saving effects of the RETE match is achieved by partially matching the condition elements and retaining the conflict set between cycles. In other words it is important to construct the alpha-memories and to remember the conflict set between cycles, but the beta-memories are of little use. We will argue below that state saved by the construction of beta-memories is less beneficial than the overhead involved in their maintenance.

Further, though TREAT may have to recompute some comparisons, there are many processors available to do the computation and the delay required for the computation may be negligible. In a VLSI machine based on an intelligent memory paradigm, the tradeoff between having memory to store all the contents of the beta-memories or having sufficient processors to recompute them quickly could lean towards the latter.

The first observation that lead to the development of TREAT is that when a new element is added to WM any new rule instantiations entering the conflict set must necessarily contain the new WM-element. Therefore, the new WM-element may be used as a seed which acts as a constraint when building new rule instantiations. By constructing the alpha-memories we can quickly compute the set of condition elements which match the new WM-element. When the match proceeds with the remaining condition elements, only the subset of WM that has partially satisfied each condition element is considered.

By similar reasoning, if a WM-element is removed from WM, than any rule instantiations removed from the conflict set must also contain the WM-element. The TREAT algorithm stores the conflict set in a distributed fashion in the DADO tree. When a WM-element is deleted, the conflict set is examined in a parallel associative manner and all conflict set elements containing the WM-element are removed from the conflict set concurrently.

If rules contained only positive condition elements, the two actions above would be sufficient.

When a WM-element partially matches a negated condition element, the algorithm is slightly more complicated. If the action of a RHS adds a WM-element that matches a negated condition element then some rule instantiations in the conflict set may have to be removed. Unlike the removal of a WM-element that matches a positive condition element, the negated condition does not appear explicitly in the conflict set. To determine which conflict set elements must be removed the condition element is temporarily considered to be positive and the new WM-element is used as a seed to build rule instantiations. These rule instantiations are then compared against the conflict set. Any instantiation appearing in the conflict set is removed.

The fourth case is when a WM-element is removed, and it partially matches a negated condition element. In this case removing the element may permit rule instantiations to enter the conflict set. These new rule instantiations are precisely those that would enter if the condition element were positive and the WM-element had just been added.

There may, however, be another WM-element similar to the removed element which prevents these new instantiations from entering the conflict set. Such an element would necessarily satisfy the negated condition element precisely the same way as the removed element, i.e. have all the same constant and variable values as the removed element. Before building the new rule instantiations WM is quickly scanned for such an element.

There is a pathological case where there may not be a single similar WM-element that prevents a rule from becoming instantiated but a pair of elements that do. This situation occurs when the same pattern variable appears in two negated condition elements of a single rule. The algorithm may be expanded to handle this circumstance, but in practice rules of this form are not found* . For pedagogical reasons we have restricted ourselves to the simple case. The interested reader may refer to (Miranker, 1984).

5 Comparison of TREAT and RETE

Both TREAT and RETE may be easily explained by considering the terminology of relational database theory. If the entire WM is considered to be tuples in a relational database, then the partial match of TREAT and single input tests of RETE may be considered as a sequence of select operations. The alpha-memories contain the resulting relationships. If two condition elements have a common variable, the act of finding pairs of WM-elements with consistent variable bindings may be viewed as a join of the corresponding alpha-memories.

*Four OPS5 expert systems have been examined and no rules of this type have been found.

In RETE when a tuple enters from one arc of a two input node it is compared against all the tuples stored in the memory associated with the other arc. Successful pairs of tuples are placed in the beta-memory. The two input test nodes of RETE incrementally build the partial join results and thus, the beta-memories contain partial join results of the query.

In this context, during the act cycle, the TREAT algorithm places changes to WM in "new" alpha-memories. The match cycle is performed by doing a join reduction with each new alpha-memory and the old alpha-memories corresponding to the remaining condition elements of the same rule. After the reduction the new alpha-memory is concatenated with the old.

The join reduction may be done in any order. It has been shown in relational database systems (Zloof, 1977) that the best way to process a query of this type is to order the join operations by increasing cardinality of the relations. It is a byproduct of this optimization that permits TREAT to perform well for both temporally redundant and nontemporally redundant systems. If changes to the WM are few on each cycle the new alpha-memories will contain 1 or 2 tuples. The optimization will then use the new alpha-memory as the seed of its query. If however there are large changes to WM the query will still be performed in optimum order rather than sequencing through the changes.

We note that TREAT must perform a search when WM-elements are added or deleted. If more than half the working memory changes per cycle the original DADO algorithm may still prove to be better.

To maintain consistent beta-memories RETE must perform the joins in a fixed order. The order is determined at compile time when no information is available about the constituent relations. Indeed it is not possible to statically determine the characteristics of the relations (Stolfo, 1984). Thus, it is unlikely that RETE performs the join in optimal order.

5.1 Implementing TREAT and RETE on a Medium Grain DADO

The following is common to both algorithms. It has been noted that in many OPS5 programs the production level parallelism is between 20 and 30. That is, no more than 30 different rules may be satisfied at a given time. The PM-level is selected at the fifth level of the tree with 32 PE's available. The rules are partitioned among the PM-level PE's. It is assumed that there is a good partitioning algorithm that prevents two rules that may be satisfied simultaneously from being placed in the same partition. (See (Ishida and Stolfo, 1984) for example.)

Within each partition the condition elements and associated alpha-memories are numbered uniquely. The select operations for each condition element are distributed among the PE's in the

WM-subtree. During the act cycle changes to working memory are broadcast to all PE's which in parallel perform their local select tests. Any successful selection is reported to the PM-level processor and the WM is stored in the appropriate alpha-memory.

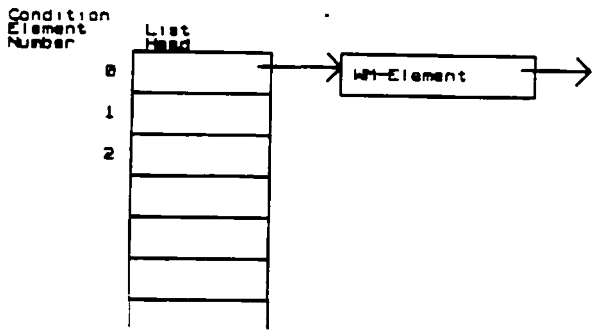

Figure 5: Working Memory Elements Indexed by Condition Element Number.

The alpha-memories are stored in a distributed fashion in a subtree, indexed by preassigned number (see fig. 5). An effort is made to place at most one WM-element per alpha-memory in a single PE. If this is impossible the disparity in the number of elements in different PE's is never greater than 1. In the TREAT algorithm a distinct partition exists for the new elements. In the RETE match the beta-memories are also numbered and stored in a fashion similar to the alpha-memories.

A join step is performed by broadcasting one tuple of one relation to every PE in a subtree. The PE's then compare the broadcast tuple to any tuples of the second relation stored in their local memory. In TREAT, if the comparison is successful, the second tuple is reported to the PM-level PE and the query continues in depth first fashion. In RETE the second tuple is reported, but the pair of tuples must also be assembled and stored in a beta-memory. We summarize TREAT by the abstract algorithm in figure 6.

5.2 Partitioning Algorithms

The TREAT algorithm provides a simple way to detect active rules and provide information for partitioning algorithms.

In the TREAT algorithm it is easy to maintain a running count of the size of each of the alpha-memories. For a particular rule if any of the alpha-memories corresponding to its condition elements are empty, then the rule can not contribute to the conflict set, and no work is performed for that rule. The rule is considered to be nonactive. The overhead for recognizing active rules is small. When updating the size of an alpha-memory we need only test for transitions from zero to one and from one to zero. Upon this transition a test must be made of the other alpha-memories for a rule. If they are nonempty then the rule is added or removed from an active list.

```
1. Initialize: Distribute the match routine and a partitioned subset of rules to each PM-
   level PE. Load the partial match tests for each condition element in a PE below the
   PM-level PE containing the associated rule. Set CHANGES to the initial WM elements.
2. Repeat the following.
3. Act: For each WM change in CHANGES do;
   a. Broadcast the WM change to all PE's.
   b. Each PE performs the partial match tests stored in its local memory.
   c. For each successful partial match test, place the change in the corresponding
      "new" alpha-memory. Each PM-level PE does this independently of the others.
   d. end do;
4. Match: Process deletes.
   a. For each nonempty "new" alpha-memory do;
   b. Associatively probe the old alpha-memory for elements appearing in the new
      alpha-memory. Remove them.
   c. Case: If the alpha-memory corresponds to a positive or a negative condition
      element.
      i. Positive: Associatively probe the conflict set for elements containing
         elements of the new alpha-memory. Remove them.
      ii. Negative:
          1. Associatively probe the old alpha-memory for elements with the same
             variable bindings as any in the new alpha-memory. If found remove
             the element from the new alpha-memory.
          2. Perform a join reduction, in optimal order, of the new alpha-memory
             and the old alpha-memories of the same rule.
          3. Add these new instantiations to the conflict set.
   d. end do.
5. Match: Process adds.
   a. For each nonempty "new" alpha-memory do;
   b. Perform a join reduction, in optimal order, of the new alpha-memory and the old
      alpha-memories of the same rule.
   c. Case: If the alpha-memory corresponds to a positive or a negative condition
      element.
      i. Positive: Add these new instantiations to the conflict set.
      ii. Negative: Associatively probe the conflict set for each of these new
          instantiations and remove if found.
   d. end do.
```

Figure 6: Abstract Algorithm Illustrating TREAT.

It is this test that provides a mechanism for developing adaptive partitioning algorithms. A good partition algorithm would keep the number of active rules in different partitions the same. If the production system monitor discovers two rules in a partition are active at the same time, the monitor may then pass this information to the partitioning algorithm. The partitioning algorithm may then be careful not to place these two rules in the same partition for the next run.

6 Expected Performance of TREAT for OPS5

Using statistics generated by studying OPS5 production systems, Gupta (gupta, 1983) has detailed performance estimates for a fine grain DADO employing the original DADO algorithm as well as performance estimates of a medium grain DADO employing the RETE match. In this section we make performance estimates of the TREAT algorithm on OPS5 by elaborating on this study. It should be noted that the study was based on OPS5 whose semantics is targeted for a sequential implementation of the RETE match. The study is not indicative of the performance of a DADO machine employing a less restrictive language that has the ability to express more parallelism in the problem. On going research aims towards the eventual implementation of a production system formalism we have come to call HerbAl (in honor of Herbert Simon and Allen Newell). HerbAl will permit the expression of parallel constructs not presently capable of OPS style systems.

To make use of the data from the OPS5 study we must first determine how many more comparisons does TREAT require than RETE as a result of eliminating beta-memories. Since DADO has many processors matching a broadcast data against their local store in parallel, the basic unit that should be counted is the required number of parallel matches.

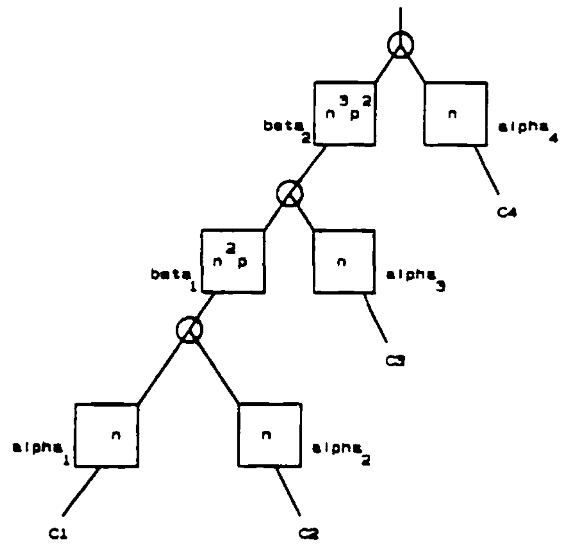

Figure 7: RETE Network Representing an Average Rule.

Since the performance of these algorithms is statistical in nature we can only make a qualitative statement based on the expected performance of an average case. The average rule in an OPS5 system has four condition elements. The RETE match network representing the memories and two input test nodes for such a rule is illustrated in figure 7. Let's assume that each alpha-memory contains n WM-elements and that there is uniform probability p that any 2 tuples match. Then beta-relations B1, and B2 will contain n^2p, n^3p^2 tuples. If a WM-element partially matching C1 is added to the system it will be compared against n tuples in alpha_2, np of them can be expected to match. These in turn must be compared to the n tuples of alpha_3. This step requires n·np comparisons and can be expected to produce n^2p^2 results. These results in turn must be compared to the n tuples in alpha_4. The total number of comparisons for an element entering C1 is then $n + n^2p + n^3p^2$. If the element partially matches C2 the number of comparisons is the same. If we add one more WM-element with even probability to the four condition elements and do a similar analysis for elements partially matching C3 and C4 the average expected number of comparisons will be:

$$n + 0.5n^2p + 0.75n^3p^2$$

The analysis for an element partially matching C1 did not make use of the results stored in the beta-memories. Since the TREAT algorithm retains no beta-memories, the number of comparisons required for the TREAT algorithm for a new element partially matching any of C1 through C4 is the same as the RETE algorithm for an element partially matching C1. Note that half of the time, when a new element partially matches C1 or C2, the number of comparisons for the two algorithms is identical. The number of comparisons for TREAT is:

$$n+n^2p+n^3p^2$$

These equations reflect the number of individual comparisons. Since we are interested in the number of parallel matches and we assume there is no more than one tuple of a relation in a processor we must divide the equations by n. Resulting in:

$$\text{Parallel Matches RETE} = 1 + 0.5np + 0.75n^2p^2$$
$$\text{Parallel Matches TREAT} = 1 + np + n^2p^2$$

The TREAT algorithm is only slightly worse. Asymptotically the two algorithms perform very much the same. The average values for n and p derived from six large production systems (Gupta, 1984) is $n = 25.6$ and $p = 0.039$. In this case the expected number of parallel matches is 2.23 and 2.98 for RETE and TREAT respectively.

However, this is an "unrealistically" average case. Indeed, for the R1 program Gupta reports an average of 56 WM-elements per alpha-memory with a standard deviation of 61. If we remove the assumption that all alpha-memories contain the same number of tokens, what is the likelihood that the RETE match has compiled the four joins in the optimal order? The compilation is done by the lexical order of the condition terms. Therefore it is fairly likely that the optimal order is not used.

Since TREAT will optimize the order of the joins on every cycle it is a fair assumption that despite the lack of beta-memories the number of parallel match operations performed by TREAT is on average the same as RETE. We conclude that the beta-memories do not reduce the average number of parallel matches, therefore it is not worthwhile to expend the time and space required to construct and maintain the beta-memories.

6.1 Performance Estimates for TREAT on a Medium Grain DADO

The parallel implementation of RETE includes the parallel associative look up of conflict set elements to be removed when a WM-element is deleted. With the exception of the construction of the beta-memories the activities of the two algorithms are almost identical. Gupta estimates for RETE that the average cost of adding a WM-element to be 3750 instructions. Of these 940 instruction are needed to construct the beta-memories. A detailed explanation of this may be found in (Miranker, 1984). The estimate for the cost of deleting a WM-element is 1800 instructions. Of these 330 instructions are required to process the beta-memories. On average there are 2.5 changes in WM per cycle. Rule selection and right hand side evaluation is assumed to take 500 instructions. The total number of instructions per cycle is then:

$$1.25\ ((3750-940)+(1800-330)) + 500 = 5850 \text{ instructions.}$$

These instruction counts are based on a DADO PE constructed out of an 8 bit 1 address processor running at an instruction rate of 2msec per instruction. Where Gupta has predicted performance for the DADO 2 prototype using the RETE match to be 67 production cycles per second, the TREAT algorithm is capable of 85 production cycles per second. Furthermore, it has been noted that the size of the beta-memories is often expansive (gupta, 1983). Thus TREAT is more space efficient as well.

Similar arguments modifying the original DADO algorithm for a fine grain DADO have been able to show an improvement from 11 production cycles per second to 105 production cycles per second. Space does not permit a complete analysis here. The reader is encouraged to see (Miranker, 1984) for details.

We note with interest that the above analysis was performed for the current DADO prototypes whose constituent processor technology is five years old. This technology was chosen to expedite prototyping within the limits of a university environment. With suitable changes to current processor technology, (1 MIP, 32 bit processors), future DADO machines will run 8 to 16 times faster than the estimates detailed above.

7 Conclusions

The TREAT algorithm overcomes disadvantages of the original DADO algorithm by saving state across production system cycles. However, the internal structures of TREAT are simpler than those of the RETE match. As a result TREAT may dynamically optimize the order of match operations on the WM and thus efficiently execute both temporally redundant and nontemporally redundant production systems.

Using the expanded abilities of TREAT and DADO a new more powerful production system language, HerbAl, is being designed to capture more parallelism than is possible to express in OPS.

We note that Gupta reports that a VAX-780 running the fastest OPS interpreter to date, OPS83 (Forgy83, 1983), is capable of only 30 to 50 production cycles per second (gupta, 1983). A DADO machine, using similar processor technology, is expected to perform 85 production cycles per second on OPS style systems. Yet such a DADO machine is considerably simpler and less expensive than a VAX-780.

References

Flynn M. J. Some Computer Organizations and Their Effectiveness. *The Institute of Electrical and Electronic Engineers Transactions on Computers*, September 1972, , .

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

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

Forgy, C. L. *OPS83.* Technical Report, Carnegie-Mellon University, 1983. unpublished manuscript.

Foster, Caxton C. *Content Addressable Parallel Processors.* : Van Nostrand Reinhold 1976.

Gupta, A. *Implementing OPS5 Production Systems on DADO.* Technical Report, Department of Computer Science, Carnegie-Mellon University, 1983.

Gupta,Anoop. *Implementing OPS5 Production Systems on DADO.* Technical Report, Carnegie-Mellon University, 3 1984.

Ishida T., and S. J. Stolfo. *Simultaneous Firing of Production Rules on Tree-structured Machines.* Technical Report, Department of Computer Science, Columbia University, 1984. (Submitted to AAAI 1984).

Lowrie,D.D.,T. Layman, D. Daer and J.M. Randal. A Programming Language for Illiac IV. *Comm. ACM*, 1975, *18 3*, ,

McDermott J. R1: A Rule Based Configurer of Computer Systems. *Artificial Intelligence*, September 1982, *19(1)*, 39-88.

Miranker, D.P. *The Performane Analysis of TREAT: A DADO Production System Algorithm.* Technical Report, Columbia University, 1984. in preperation.

Newell, A. Production Systems: Models of Control Structures. In W. Chase (Ed.), *Visual Information Processing*, : Academic Press, 1973.

Stolfo S. Learning control of production systems. *Cognition and Brain Theory*, 1984, , .

Stolfo S. J., and D. E. Shaw. *DADO: A Tree-structured Machine Architecture for Production Systems.* Proceedings National Conference on Artificial Intelligence, Carnegie-Mellon University, August, 1982.

Stolfo S. J., and G. Vesonder. *ACE: An Expert System Supporting Analysis and Management Decision Making.* Technical Report, Department of Computer Science, Columbia University, 1982. (To appear in the Bell System Technical Journal).

Zloof, M. M. Query-by-example: a data base language. *IBM System J.*, 1977, *16:4*, 324-343.