CMU-CS-84-115

Implementing OPS5 Production Systems on DADO

March 1984

Anoop Gupta
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

DADO is a highly parallel tree-structured architecture designed to execute *production systems* at Columbia University. In this paper, we analyze the performance of DADO when executing OPS5 production system programs. The analysis is based on the predicted performance of three different algorithms for implementing production systems on DADO. We show that the large-scale parallelism in DADO is not appropriate for executing OPS5-like production systems. The reasons are: (1) in current production systems programs, actions of a production do not have global affects, but only affect a small number of productions, and (2) large-scale parallelism almost always implies that the individual processing elements are weak. Since only a small number of productions are affected every cycle, only a few of the large number of processing elements perform useful work. Furthermore, since the individual processing elements are weak, the performance is worse than if a small number of powerful processors are used. The tree-structured topology of the DADO architecture is not found to be a bottleneck.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

List of Figures

1. Introduction

Production systems (or rule-based systems) are widely used in Artificial Intelligence to model learning, and to build expert systems [5, 6, 7, 8, 9, 10, 11]. Most such systems are extremely computation intensive, and their slow speed of execution has prohibited their use in domains requiring high performance and in domains requiring real-time response. The need for faster execution of production systems has led to research in special purpose hardware architectures to support these systems. For instance, there are ongoing projects at Columbia University and Carnegie-Mellon University to speed-up the execution of production systems. As a part of our effort at Carnegie-Mellon, we are also analyzing other proposed architectures for production systems. The architecture we consider in this paper is DADO, a parallel tree-structured architecture proposed by Salvatore Stolfo, Daniel Miranker, and David Elliot Shaw of Columbia University.[1] The analysis of DADO is based on the predicted performance of three algorithms—two algorithms previously published by the Columbia research group [12, 13], and one new algorithm that we have proposed. The performance analysis of the individual algorithms is, in turn, based on the static and run-time data that we have gathered for six OPS5 production system programs [3].

In Section 2 of the paper we describe the DADO architecture, the OPS5 production system language, the Rete algorithm to implement production systems, and comment on the accuracy of our analysis. In Sections 3, 4, and 5 we present three algorithms to implement OPS5 production systems on DADO, and discuss the performance, the advantages, and disadvantages of each of them. In Section 6, we present a summary of the algorithms and our conclusions.

2. Background

2.1. The DADO Machine Architecture

The first paper on DADO appears in the proceedings of AAAI-82 conference, and is entitled "DADO: A Tree-Structured Machine Architecture for Production Systems" [13]. Since then a number of papers describing the software and hardware configuration of DADO have been published [14, 12, 15]. The following description of DADO has been taken from the most recent paper [12].

The proposed DADO machine architecture consists of a very large number (on the order of hundreds of thousands in the full-scale version) of *processing elements* (PEs), interconnected to form a complete binary tree. Each PE consists of its own processor (8 bit wide data paths in the prototype), a small amount of random access memory (8K bytes in the prototype), and a specialized I/O switch that is constructed using a custom

[1] Note: DADO can be considered for uses other than production systems, for example, to implement Prolog [15, 16]. We do not consider the other uses in this paper.

VLSI chip. Figure 2-1 shows the major components of the proposed DADO machine.[2]

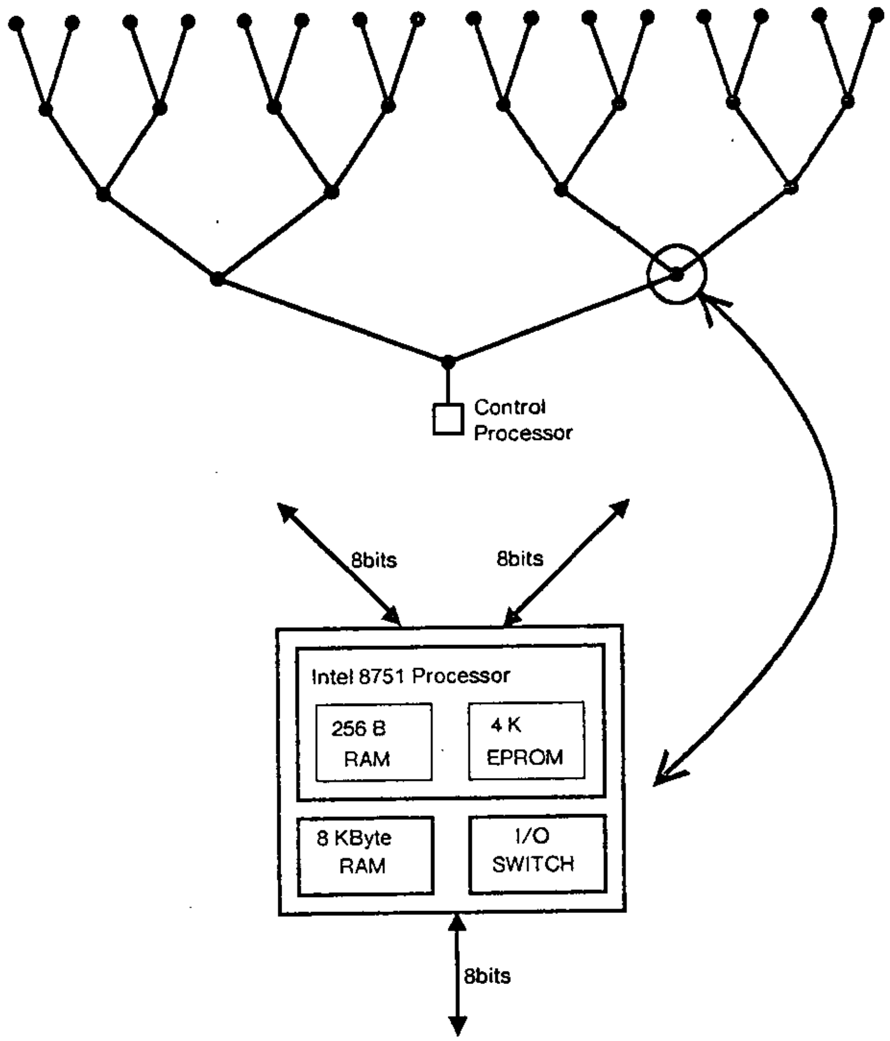


Figure 2-1: The Prototype DADO Architecture

Each PE in DADO is capable of executing in two modes. In the first mode, called *SIMD* (single instruction stream multiple data streams), the PE executes instructions broadcast by an ancestor PE within the tree. In the second mode, called *MIMD* (multiple instruction streams multiple data streams), the PE executes instructions stored in its own memory. There is a distinguished *control processor*, adjacent to the root of the DADO tree, that controls the operation of the complete ensemble.

When a DADO PE enters MIMD mode, its logical state is changed in such a way as to effectively

[2] Note: The DADO tree in Figure 2-1 is drawn upside down compared to the DADO tree in other papers [12, 13].

disconnect it and its descendants from all other PEs in the tree. In particular, a PE in MIMD mode does not receive instructions that might be placed on the tree-structured communication bus by any of its ancestors. Such a PE may, however, broadcast instructions to be executed by its own descendants. A PE in SIMD mode may be in an *enabled* or a *disabled* state. When in an enabled state, a SIMD PE executes instructions broadcast by an ancestor PE. When in a disabled state, a SIMD PE ignores instructions that are placed on the broadcast bus by the ancestor, although descendants of this PE who remain *enabled* will receive and execute such instructions. The DADO machine thus provides support for large-scale MSIMD (multiple SIMD) execution. This flexible architectural design supports the logical division of the machine into distinct partitions, each executing a distinct task.

The DADO I/O switch, which is implemented in custom VLSI, has been designed to support communication between tree neighbors, as well as communication between PEs that are adjacent in a logical linear ordering embedded within the tree. In addition, a specialized combinatorial circuit incorporated within the I/O switch allows for the rapid selection of a single distinguished PE from a set of candidate PEs in the tree.

The language used for programming DADO is called PPL/M (parallel PL/M) [14]. In addition to the standard data structures and control constructs provided by conventional languages, the PPL/M language contains a number of constructs to specify operations to be performed by independent PEs in parallel. For example, the "SLICE" construct in PPL/M is used to define variables and procedures that are resident within each PE. An assignment of a value to a SLICEd variable causes the transfer to occur simultaneously within each enabled PE. Similarly, a call to a SLICEd procedure causes that procedure to be executed in each enabled PE. The "DO SIMD" construct is used to enclose a sequence of instructions that are to be broadcast to all PEs. There exist instructions for common operations (e.g., BROADCAST, CALL, ENABLE, SEND, RECEIVE, etc.) that one would like to perform on a tree of processors. There also exists a special instruction, called "RESOLVE", which uses the combinatorial hardware in the I/O switch to select one out of all the enabled PEs in a single cycle. This instruction is extremely useful in sequentially enumerating the members of a distributed set (a set whose elements are distributed amongst the various PEs in the tree).

In summary, the DADO architecture consists of a large number of PEs organized as a complete binary tree. Each PE has a small amount of memory, and can function either in SIMD or MIMD mode. Each PE has a specialized I/O switch that enables it to broadcast data, select one out of many enabled PEs, receive data, send data, in one instruction cycle. The latest prototype of DADO, called DADO-2, consists of 1023 PEs. Each PE has 8 kilobytes of memory and an instruction cycle time of $2\mu s$.[3] The 1023 processor DADO-2, when

[3] Note that as the number of PEs increases, the instruction cycle will have to be slowed down to compensate for the extra logic levels that the signals have to traverse.

completed, is expected to be of the same hardware complexity as a Digital Equipment Corporation VAX-11/750.

2.2. Production Systems

In the following paragraphs we give a brief overview of production systems and the OPS5 production system language [1].

A production system is composed of a set of *if-then* rules called *productions* that make up the *production memory*, and a database of assertions called the *working memory*. The assertions in the working memory are called *working memory elements*. Each production consists of a conjunction of *condition elements* corresponding to the *if* part of the rule (also called the *left hand side* of the production), and a set of *actions* corresponding to the *then* part of the rule (also called the *right hand side* of the production). The actions associated with a production can add, remove or modify working memory elements, or perform input-output. Figure 2-2 shows an OPS5 production named p1, which has three condition elements in its left hand side, and one action in its right hand side.

```
(p  p1  (C1  ↑attr1 <x>    ↑attr2 12)
        (C2  ↑attr1 15     ↑attr2 <x>)
      - (C3  ↑attr1 <x>)
   -->
        (remove 2))
```

Figure 2-2: A Sample Production

The OPS5 interpreter executes a production system program by performing the following operations:

- MATCH: In this first phase, the left hand sides of all productions are matched against the contents of working memory. As a result we obtain a *conflict set*, which consists of *instantiations* of all satisfied productions.

- CONFLICT RESOLUTION: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.

- ACT: In this third phase, the actions of the production selected in the conflict resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, the first phase is executed again.

In OPS5, a working memory element is a parenthesized list consisting of a constant symbol called the *class* of the element and zero or more *attribute-value* pairs. The attributes are symbols that are preceded by the operator ↑. The values are symbolic or numeric constants. The condition elements in the left hand side of a production are parenthesized lists similar to working memory elements. However, the condition element is less restricted than the working memory element; while the working memory element can contain only constant symbols and numbers, the condition element can contain variables, predicate symbols, and a variety of other operators, as well as constants.

A working memory element matches a condition element if the value of every attribute in the condition element matches the value of the corresponding attribute in the working memory element. If the condition element value is a variable, it will match any value. (A variable in OPS5 is an identifier enclosed within angle brackets < and >.) However, if a variable occurs more than once in a left hand side, all occurrences of the variable must match identical values. A production is said to be *satisfied* when for every condition element in its left hand side, there exists a working memory element that matches it.

The right hand side of a production consists of an unconditional sequence of actions which can cause input-output, and which are responsible for changes to the working memory. Three kinds of actions are provided to effect working memory changes. *Make* creates a new working memory element and adds it to working memory. *Modify* changes one or more values of an existing working memory element. *Remove* deletes an element from the working memory.

2.3. The Rete Algorithm

The most time consuming phase in the execution of production systems is the match, where the left hand sides of all productions are matched against the contents of the working memory. In the following paragraphs we briefly describe the Rete algorithm for performing match [2]. We introduce the terminology and the concepts that are used by two other algorithms described later in this paper.

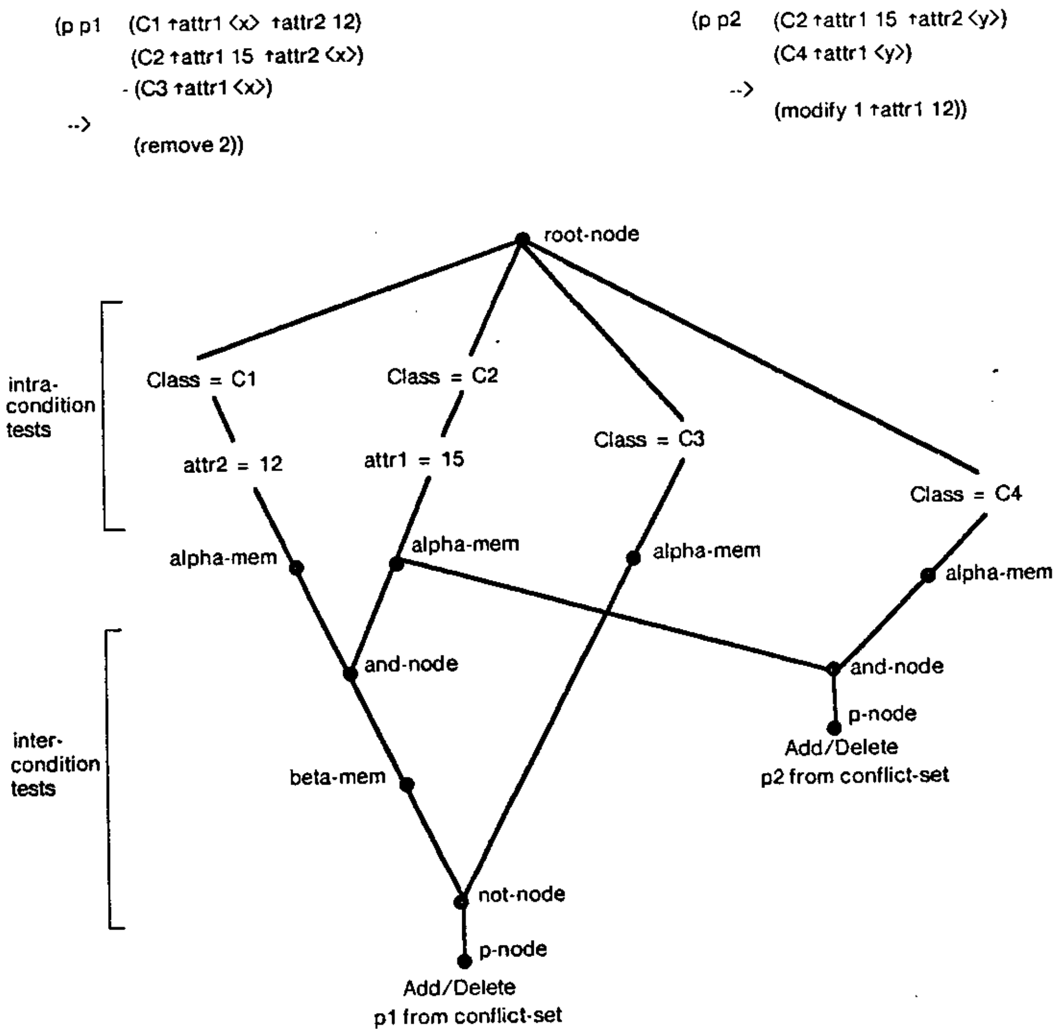
The Rete algorithm is especially designed for OPS5-like production systems. Its main features are:

- It exploits similarity in condition elements of productions to minimize the number of tests that are performed. Tests for satisfaction are shared between similar condition elements, and the failure of a test may disqualify several condition elements.

- It exploits the fact that working memory changes very slowly in OPS5-like production systems. While the working memory may contain several hundred working memory elements, each production firing (on average) makes only two or three changes to working memory. The Rete algorithm stores the result of match from previous cycles, so that it need not perform match again for the unchanged portion of the working memory.

The Rete algorithm uses an augmented discrimination network compiled from the left hand sides of productions to perform match. The network represents a dataflow graph, whose input consists of the changes to the working memory and whose output consists of the changes to the conflict set. The dataflow graph embodies the parallelism that may be used to perform the match. This is why the second algorithm described in this paper attempts to map the Rete network onto the DADO tree of processors.

To generate the network for a production, the network compiler proceeds first with the individual condition elements in the left hand side. For each condition element it chains together test nodes that check: (1) if the attributes in the condition element that have a constant as their value are satisfied, (2) if the attributes in the condition element that are related to a constant by a predicate are satisfied, and (3) if two occurrences of the same variable within the condition element are consistently bound. Each node in the chain performs one such test. The three kinds of tests above are called *intra-condition* tests, because they correspond to individual condition elements. Once the network compiler has finished with the individual condition elements, it adds nodes that check for consistency of variable bindings across the multiple condition elements in the left hand side. These tests are called *inter-condition* tests, because they refer to multiple condition elements. Finally the compiler adds a special terminal node to represent the production to which this network corresponds.



```
(p p1   (C1 ↑attr1 <x>  ↑attr2 12)
        (C2 ↑attr1 15  ↑attr2 <x>)
      - (C3 ↑attr1 <x>)
  -->
        (remove 2))

(p p2   (C2 ↑attr1 15  ↑attr2 <y>)
        (C4 ↑attr1 <y>)
  -->
        (modify 1 ↑attr1 12))
```

Figure 2-3: The Rete Network

Figure 2-3 shows such a network for productions p1 and p2 which appear in the top part of the figure. In this figure, lines have been drawn between nodes to indicate the paths along which data flows (from the top node down). The nodes with a single predecessor (near the top of the network) perform the intra-condition tests. The nodes with two predecessors (the and-nodes and the not-nodes) check for consistency of variable bindings between working memory elements matching different condition elements. The computation performed by these nodes is equivalent to a *join* (as in the relational database literature). The terminal nodes (p-nodes) that indicate that a production is completely satisfied are at the bottom of the figure. Note that when two left hand sides require identical nodes, the compiler shares part of the network rather than building duplicate nodes.

The objects that are passed between nodes in the network are called *tokens*, which consist of a *tag* and a *list of working memory elements*. The tag can be either a +, indicating that something has been added to the working memory, or a −, indicating that something has been removed from it. The list of working memory elements associated with a token corresponds to those elements that the interpreter is trying to match or has already matched against a subsequence of condition elements in the left hand side.

To avoid performing the same tests repeatedly, the Rete algorithm stores the result of the match with working memory as *state* within the network. Tokens that satisfy a single condition element are stored in α-mem nodes. Tokens that satisfy a sequence of condition elements in the left hand side are stored in β-mem nodes. Since state is preserved, only changes made to the working memory by the most recent production firing have to be processed every cycle. Thus, the input to the Rete network consists of the changes to the working memory. These changes filter through the network, and where relevant, the state stored in the network is updated. The output of the network consists of a specification of changes to the conflict set.

2.4. Accuracy of Our Analysis

In our analysis of DADO, we experienced the most difficulty due to the fact that none of the three algorithms is currently implemented on DADO.[4] Although the specification of the algorithm lets us predict the number of coarse steps required, it is very difficult to predict the instruction cycles required by any given coarse step. For example, in algorithm-2, there is a step in which we must allocate a record from memory, copy some information into this record, and store this record into a hash table. To predict the performance we need to know the number of instruction cycles required to execute this step. The number of cycles required depends on many things: the complexity of the hash function; whether the record is allocated from a free-list or whether it is allocated from raw memory; whether the code for storage allocation causes a

[4] We have recently learned that an unoptimized version of the first algorithm is now running on the prototype DADO. (Personal communication from Daniel Miranker at Columbia.)

procedure call or whether it is in-line; etc. There is no simple way to find this number, other than to actually write the code.

Because of the above difficulty, we had two alternatives for the analysis. The first was to give only qualitative results for each of the algorithms. The second was to perform rough calculations (for the kind of steps mentioned in the previous paragraph) and to give approximate but quantitative results. We decided in favor of the second alternative for two reasons:

- We felt that our rough estimates were good enough, that is, the inaccuracy in our estimates would not cause a qualitative change in our conclusions.
- Rough quantitative analysis often results in a better understanding of the problem than a large amount of qualitative analysis.

Thus a large fraction of the numbers in the performance section of the three algorithms are a result of back of the envelope calculations. We believe that a factor of two or three difference in such calculations will not change our final conclusions about the algorithms or the architecture. Finally, our performance analysis is based on estimates of instruction cycles taken by the Intel 8751 processors used in the DADO-2 prototype.[5] The performance may be substantially different if custom processors are used. However, throughout the analysis, we assume no limit on the number of processors that are available. For example, if implementing an algorithm requires 100,000 processors, we assume in our performance analysis that 100,000 processors are available. The performance will be quite different if the number of processors is smaller than that required by the algorithm.

To evaluate the performance of the DADO architecture, we use the static and run-time statistics gathered from six diverse OPS5 production system programs [3]. These programs are:

- R1 [7], a program for configuring VAX computer systems. It consists of 1932 productions.
- XSEL, a program which acts as a sales assistant for VAX computer systems. It consists of 1443 productions.
- PTRANS [4], a program for factory management. It consists of 1016 productions.
- HAUNT, an adventure game program. It consists of 834 productions.
- DAA [5], a program for VLSI design. It consists of 131 productions.
- SOAR [6], an experimental problem solving architecture implemented as a production system. It consists of 103 productions.

[5] Daniel Miranker, who is working on implementing production systems on the prototype DADO machine at Columbia University, after reading an earlier draft of this paper, tells us that our instruction cycle estimates are quite accurate.

For this paper, we have averaged the statistics for the six programs. These averaged statistics are used for all the performance analysis.

3. Algorithm-1: Performing Match in Parallel for All Productions

Algorithm-1 is described by Stolfo and Shaw in the AAAI-82 paper [13]. To implement this algorithm, the DADO tree is divided into three parts: the *upper tree*, the *lower tree*, and the PEs at the *production-memory level* (PM-level). The productions are stored at the PM-level, and the PM-level is chosen such that one PE is available for each production. The descendants of each PM-level PE form a *working-memory subtree* (WM-subtree). A WM-subtree stores working memory elements that are *relevant*[6] to the associated production. The lower tree is used for conflict resolution. The above partitioning of the DADO tree is shown in Figure 3-1.

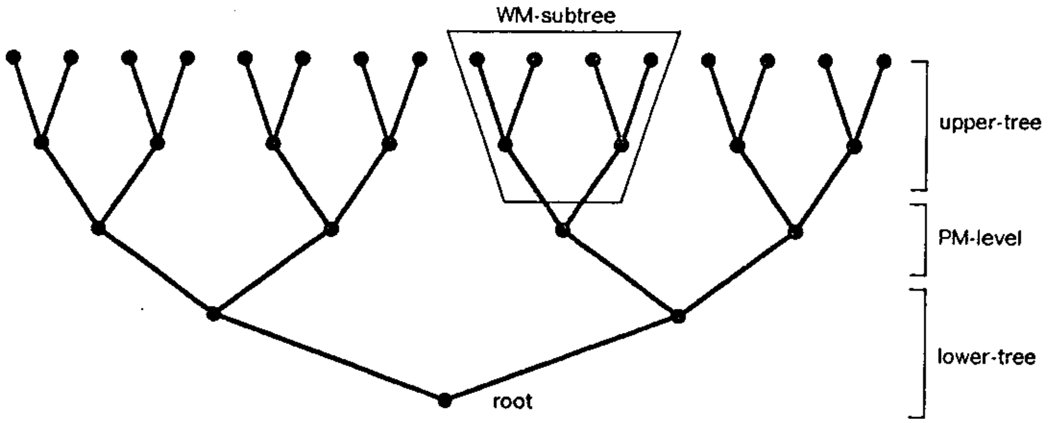


Figure 3-1: Algorithm-1

Starting from the execution of the right hand side of a production, algorithm-1 consists of the following steps:

1. Broadcast changes in working memory to all PEs at the PM-level. The next three steps are executed independently, in parallel, by each PM-level PE.

2. Associatively search and delete the working memory elements to be removed.

3. Store the relevant working memory elements (from the new additions to the working memory) into free PEs in the WM-subtree.

4. Perform match.

5. Synchronize, that is, wait for all PEs at the PM-level to finish match.

[6] A *relevant* working memory element for a production is one that can be a part of some instantiation of that production. Thus, a working memory element that satisfies all intra-condition tests for some condition element of a production is relevant to that production.

6. Use lower tree for conflict resolution.

7. Execute the right hand side of the selected production.

In algorithm-1, the match for all productions is performed in parallel by the PM-level PEs. The *primitive match step* used is the associative lookup of all working memory elements matching a given condition element. To perform the associative lookup, the PM-level PE broadcasts the attribute-value pairs of the condition element to all PEs in the WM-subtree. The PEs in the WM-subtree, in parallel, compare the attribute-value pairs being broadcast to the working memory element stored in their memory. The PEs that do not have a matching working memory element disable themselves. The PEs that are still enabled at the end of the broadcast have matching working memory elements, and can be sequentially accessed using the resolve instruction. The algorithm assumes that each PE in the WM-subtree stores only one working memory element. If more than one such element is stored in any PE, the associative match is slower.

Matching a production with multiple condition elements is much more complex than matching a production with a single condition element. Instead of one, it requires several invocations of the primitive match step. For example, the steps required to match the production

```
(p pr1   (C1     ↑attr1 <x>      ↑attr2 12)
         (C2     ↑attr1 <y>      ↑attr2 <x>)
         (C3     ↑attr1 <x>      ↑attr2 <y>)
    -->
         (remove 2))
```

against the working memory elements

```
WME-1. (C1  ↑attr1 12   ↑attr2 12)
WME-2. (C1  ↑attr1 15   ↑attr2 12)
WME-3. (C2  ↑attr1 7    ↑attr2 15)
WME-4. (C3  ↑attr1 9    ↑attr2 3)
WME-5. (C3  ↑attr1 15   ↑attr2 7)
```

are shown below.

1. Associatively find working memory elements matching the first condition element. In our example, the condition element is "(C1 ↑attr1 <x> ↑attr2 12)", and the matching working memory elements are WME-1 and WME-2.

2. For each working memory element found in the previous step:

   a. Read back the *working memory element identity* (WME-ID) and the values of any newly bound variables from the PE in the WM-subtree. Thus, the first time through this loop, we will read back WME-ID as WME-1, and the value of variable <x> as 12.

   b. Substitute values of variables bound in the first condition element for corresponding variables in the second condition element. Thus, when we are considering WME-1, we substitute the value 12 in place of the variable <x> in the second condition element. When we are considering WME-2, we substitute the value 15 in place of <x>.

c. Associatively find all working memory elements matching the second condition element (with the variable substitutions from the previous step still in place). Thus, in our example, we associatively search for working memory elements matching "(C2 ↑attr1 <y> ↑attr2 15)" and not "(C2 ↑attr1 <y> ↑attr2 <x>)".

d. For each working memory element matching the second condition element:

   i. Read back the WME-ID and the values of any newly bound variables from the PE in the WM-subtree. We now have a pair of working memory elements (formed by the working memory element matching the first condition element and the working memory element matching the second condition element) that satisfy the first two condition elements of the production. In our example, the working memory element pair we get is (WME-2, WME-3).

   ii. Substitute values of variables bound in the first two condition elements for corresponding variables in the third condition element.

   iii. Associatively find all working memory elements matching the third condition element (with variable substitutions still in place). Thus, we search for working memory elements matching "(C3 ↑attr1 15 ↑attr2 7)". The result of the search is WME-5, and the triple (WME-2, WME-3, WME-5) is the final result of the match.

The steps 2-a, 2-b, and 2-c above characterize the *inner loop* of the match process for complex productions. The complexity of the inner loop is used in the next subsection to estimate the performance of algorithm-1.

3.1. Performance

In the following paragraphs, we estimate the performance of the DADO machine. As mentioned in Section 2.4, the performance calculations are based on the average computational requirements of six production system programs.

Since match is the most expensive step in executing production system programs, we first estimate its cost. To perform match, algorithm-1 repeatedly executes three steps (2-a, 2-b, and 2-c) in its inner loop. To determine the number of times that these steps are executed, consider the abstract Rete network shown in Figure 3-2. The labels k1 to k7 stand for the number of tokens in each of the memory nodes. For example, k1 stands for the number of working memory elements matching CE1, k3 stands for the number of working memory element pairs that consistently satisfy the first two condition elements, k5 stands for the number of working memory element triples that consistently satisfy the first three condition elements, and k7 stands for the number of instantiations of the production that are in the conflict set. It can be shown that the number of times the inner loop is executed is equal to $(1 + k1 + k3 + k5)$. Furthermore, it can be shown that the number of relevant working memory elements (the number of working memory elements that are stored in the WM-subtree) is equal to $(k1 + k2 + k4 + k6)$.[7]

[7] Assuming that a given working memory element is not relevant to more than one condition element of the same production

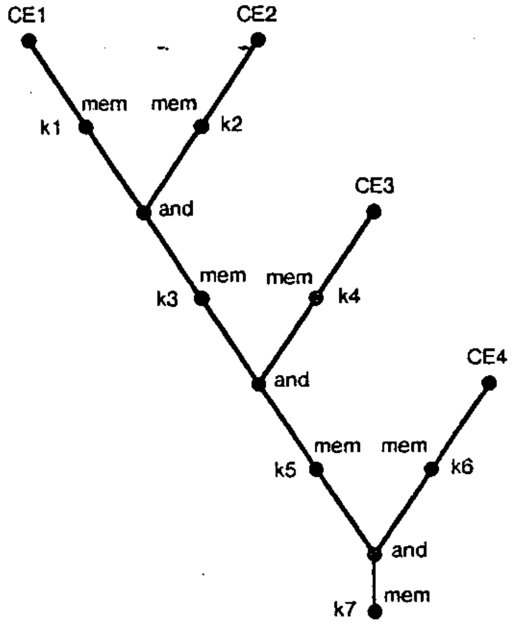


Figure 3-2: Rete Network for a Single Production

The average values for k1, k3, and k5 are 26, 18, and 18 respectively.[8] Then, the number of executions of the inner loop is (1 + 26 + 18 + 18) = 63.[9] We, now estimate the number of instruction cycles taken by each execution of the inner loop (steps 2-a, 2-b, and 2-c).

- In step 2-a, we fetch the WME-ID and the new variable bindings from a PE in the WM-subtree. Assuming that one variable binding (4 bytes) is transmitted back and that the WME-ID is 4 bytes long, we fetch 8 bytes of information. Since it takes approximately 5 instruction cycles to fetch one byte of information,[10] each invocation of step 2-a takes (5 * 8) = 40 instruction cycles.

- In step 2-b, we substitute values for bound variables in the next condition element. We assume this takes 40 instruction cycles.

- In step 2-c, we broadcast the attribute-value pairs of the next condition element to PEs in the WM-subtree. The number of bytes that are broadcast is equal to

  ```
  (attributes per CE) * (bytes per attribute-name + bytes per value),
  ```

  which is equal to 4 * (1 + 4) = 20. In DADO, the match can be done within the same code loop, and it requires approximately 10 instruction cycles per byte. Thus, step 2-c takes 20 * 10 = 200 instruction cycles.

[8] Note: 26 is the average number of tokens found in an α-mem node, and 18 is the average number of tokens found in a β-mem node.

[9] Note: For our calculations, we are using the average values for k1, k3, and k5. To be more accurate, we should use the worst-case sum of (1 + k1 + k3 + k5), as the PE having that combination will take the longest time, and all other PEs will have to wait for it to finish.

[10] This number is extracted from some code in [14].

Using the above information, the cost of processing a single change to the working memory is 63 * (40 + 40 + 200) = 17,640 instruction cycles. Since 2.5 changes are made to working memory per production firing, the total cost of match is 17640 * 2.5, or approximately 44,000 instruction cycles. Assuming the cost of conflict-resolution, broadcasting changes in working memory to PM-level PEs, and evaluating the right hand side of a production to be 2000 instruction cycles, the total cost per production firing is 46,000 instruction cycles or 92ms (recall that the each instruction cycle takes 2μs). Thus, the estimated performance of DADO using this algorithm is 11 production firings per second.

3.2. Evaluation

The advantage most often cited for algorithm-1 is that the time for match is independent of the number of productions in the program. To achieve this, however, the number of processors used is proportional to the number of productions. For instance, to achieve the performance of 11 production firings per second, algorithm-1 requires approximately 100 PEs per production. This large number of PEs is required because the algorithm assumes that there is only one working memory element per PE in the WM-subtree. Finally, the fact that the time for match is independent of the number of productions is not especially significant, because the Rete algorithm implemented on a uniprocessor already does match in time independent of the number of productions [3].

The main reason for the poor performance of algorithm-1 is that it does not save *state* across cycles, that is, it does not save the information gained as a result of matching the left hand sides of productions to the working memory from previous cycles. Since state is not saved, it has to be recomputed every cycle at a large cost. If state is saved, only incremental changes to the old state are necessary. The Rete algorithm exploits this to achieve a performance of about 30 - 50 production firings per second (for OPS83) on a VAX-11/780. Note, however, that saving state is advisable only if state changes slowly. In case the majority of the working memory changes every cycle, saving state will have a negative effect. In most OPS5 production system programs that we have measured, less than five percent of the state changes every cycle.

Another deficiency in algorithm-1 is that it makes poor use of the hardware. This is because measurements show that only a small number of productions are affected on any cycle, that is, the changes to the working memory cause change to the state of only a small number of productions. This implies that the majority of the PM-level PEs and their associated WM-subtrees do no useful work. For example, we observed that, regardless of the total number of productions in the program, only about 50 productions get affected every cycle. This means, that for a production system with 2000 productions, the hardware utilization will be less than (50/2000 = 0.04) four percent. Also the lower tree, which consists of as many PEs as the number of productions, is used only for the duration of conflict resolution.

Overall, algorithm-1 uses a very large number of processors to achieve minimal performance. The latter two algorithms in this paper have better performance and make better use of the hardware.

4. Algorithm-2: Mapping the Rete Network Directly onto DADO

Algorithm-2 is mentioned in the IJCAI-83 paper [12], and was described briefly by Salvatore Stolfo when visiting CMU. In this algorithm, Stolfo, Miranker, and Shaw suggest mapping the Rete network directly onto the DADO tree of processors. Here the authors use a basically good Rete algorithm, and the attention shifts to how well it can be implemented on DADO.

To implement algorithm-2, the DADO tree is divided into three parts: the lower tree, the upper tree, and the PEs at the PM-level. The PM-level is chosen so that at least one PE is available per production. The PEs at the PM-level store the terminal nodes (p-nodes) in the Rete network, and the PEs in the upper tree store all other nodes of the Rete network. The lower tree is used for conflict resolution.

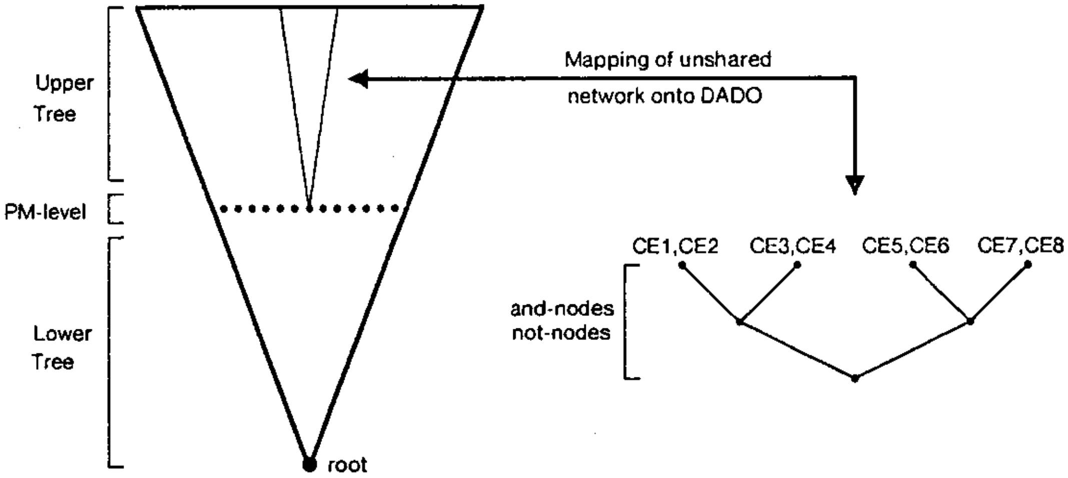


Figure 4-1: Algorithm-2

The mapping of the Rete network onto DADO is shown in Figure 4-1. The mapping requires approximately eight processors per production. This mapping has several special features. First, the nodes responsible for intra-condition tests before an α-mem node are stored in the same PE as the α-mem node. Second, instead of constructing the standard linear Rete network, as shown in Figure 3-2, we construct a network that matches the binary tree topology of DADO, as shown in Figure 4-1. Third, we do not exploit sharing of nodes as is done within Rete networks constructed for uniprocessor implementations. The reason is that we must use fan-out trees to implement the branching associated with shared nodes, which nullifies any savings in PEs that we may have by sharing. Fourth, the memory nodes feeding a two-input node are stored in the same PE as that two-input node. This is because the memory nodes store state that should be easily accessible to the two-input node. If the memory nodes are stored in different PEs the state will not be easily accessible.

The execution of algorithm-2 closely resembles the execution of the uniprocessor Rete algorithm, except that nodes get evaluated in parallel. Starting from the execution of the right hand side of a production, algorithm-2 consists of the following steps:

1. The changes to working memory are broadcast from the root to all other PEs. Each PE in the upper tree stores the new working memory elements in temporary buffers.

2. PEs having α-mem nodes, in parallel, evaluate the sequences of intra-condition tests that are stored in them. At the end of this step, we know the set of satisfied α-mem nodes.

3. Each PE containing a satisfied α-mem node constructs a token and stores it in the token memory (organized as a hash table). A pointer to this token is then passed to the two-input node below. The two-input node checks this token for consistent variable bindings with tokens in its opposite memory. If it finds a token in the opposite memory that has consistent bindings, a new token (formed by pairing the two consistent tokens) is constructed, and sent to the β-mem node below. This last step requires communication between two distinct PEs, since the β-mem node resides on a PE below the one containing the α-mem node (see Figure 4-1). The activation of the β-mem node is then processed in the same way as we processed the satisfied α-mem node.

4. After match for the individual productions is finished, PEs at the PM-level determine the best instantiation, if any, for the associated productions.

5. The PEs at the PM-level wait until all PEs have finished with the previous step.

6. Each PE at the PM-level sends the identity of the associated production and an encoded priority for the best instantiation to its parent. The parent compares the priorities received from its two children, and passes on the priority and identity of the winner to its parent. This is repeated by PEs at all levels in the lower tree. The production identity received by the root processor corresponds to the winner of conflict resolution.

7. The root PE executes the right hand side of the selected production.

4.1. Performance

To estimate the performance of algorithm-2, we first determine the cost of processing a single change to the working memory. Since the match for all productions is performed in parallel, the cost of processing a single change to the working memory is determined by the production that takes the longest time to finish match.

We estimate[11], that in the production taking longest to finish match, tokens flow across two levels of the DADO tree, that is, an α-mem node activates an and-node, which in turn activates another and-node, which

[11] based on our analysis of the six OPS5 production system programs

in turn activates another and-node, which then causes a change to the conflict set. We now estimate the cost of processing such a production:

- In step-1, a working memory element is broadcast from the root to all other PEs. Since the average working memory element consists of 24 attribute-value pairs, 120 bytes of information is broadcast (assuming one byte per attribute-name, and four bytes per value). As it takes approximately 5 instruction cycles to broadcast and store a byte, step-1 takes 120 * 5 = 600 instruction cycles.

- In step-2, PEs storing α-mem nodes determine in parallel the set of satisfied α-mem nodes. This requires evaluating an average of three intra-condition tests. We assume this step takes 100 instruction cycles.

- Step-3 includes the time taken by the activation of all nodes below and at the α-mem node level. The following costs are incurred:

  - o Processing an α-mem node requires allocating storage, constructing the token, computing hash functions of variables, and linking the token into the hash table. We assume this takes 300 instruction cycles. The activation of the two-input node below the α-mem node requires looking up the opposite memory (organized as a hash table) to find matching tokens, and the transfer of the matching token to the β-mem node below (recall that the β-mem node is on a different PE). We assume that lookup takes 200 instruction cycles. The cost of transmitting the matching token (24 bytes) to the β-mem node below takes 5 * 24 = 120 instruction cycles. The total cost of this step is 300 + 200 + 120 = 620 instruction cycles.

  - o The β-mem node receiving the above token must store it in a hash table. As assumed for α-mem nodes, this takes 300 instruction cycles. The activation of the two-input node caused by the new token is assumed to take 200 instruction cycles. The cost of transmitting the matching token to the β-mem node below is 5 * 32 = 160 instruction cycles, which makes the total cost of this step to be 300 + 200 + 160 = 660 instruction cycles.

  - o The β-mem node receiving the above token must store it in a hash table. The cost of this is 300 instruction cycles. The cost of the two-input node activation is 200 instruction cycles. The output of this two-input node is the new instantiation of the production. The total cost of this step is 300 + 200 = 500 instruction cycles.

  The total cost of step-3 is (620 + 660 + 500) = 1780 instruction cycles.

- In step-4, PEs at the PM-level determine the best instantiation, if any, for the associated productions. We assume this takes 100 instruction cycles.

- In step-5, the PEs at the PM-level must synchronize, that is, they must wait for the production which takes the longest time to finish match. Since we are calculating the cost for the production that takes the longest, this step takes no time.

- In step-6, the conflict resolution step, values must propagate through ten levels in the DADO tree (since the PM-level for the average program is level ten). The computation performed by a PE at each level requires receiving 6 bytes of information (the production identity and its encoded priority) from each of its two children and comparing two 4 byte long numbers. We estimate that the total cost for conflict resolution is about 800 instruction cycles.

- In step-7, the execution of right hand side takes approximately 300 instruction cycles.

The total cost of performing match for a single change to working memory is (600 + 100 + 1780) = 2480 instruction cycles. The cost for steps 4-7 is not included because they are executed only once per production firing. Since there are 2.5 changes to working memory per production firing, the cost per production firing is (2.5 * 2480) + 100 + 800 + 300 = 7400 instruction cycles or 14.8ms. This results in a performance of 67 production firings per second. Note that in the above calculations we are assuming that the worst case computation for all 2.5 changes is occurring in the same partition. Since this is very unlikely, we expect the actual performance to be somewhat better.

4.2. Evaluation

As the previous paragraphs show, the performance of algorithm-2 is much better than that of algorithm-1. The reasons for the good performance of algorithm-2 are:

- In addition to production-level parallelism (which is also exploited by algorithm-1), algorithm-2 exploits node-level parallelism, that is, activations of different nodes within the same production are evaluated in parallel. This is because different nodes in the Rete network are allocated on different PEs.

- Unlike algorithm-1, the PEs store state within their memories (as prescribed by the Rete algorithm), so that the state is not recomputed every cycle.

The weak points of algorithm-2 are:

- Memory associated with each PE (8 kilobytes) is too small. Measurements show that occasionally a single memory node may have as many as 500 tokens. Since 500 tokens require 15 kilobytes of memory (assuming each token to be 30 bytes), the suggested implementation of algorithm-2 on DADO will not work in its current form. For algorithm-2 to work, it is necessary to design a scheme to handle overflows. We have examined a couple of schemes, but more research is needed.

- Hardware utilization is very low. This is because the number of processors used is proportional to the number of productions (approximately eight processors per production), while only a small number of productions actually change state on any given cycle. As shown for algorithm-1, we expect the hardware utilization to be around 5%.

- The lower tree (which contains as many PEs as the number of productions) for conflict resolution is unnecessary. Measurements show that on average 2.5 changes are made to the conflict set for every change to the working memory. We believe that it is faster to transmit these changes to a central processor for conflict resolution, than to use the lower tree.

- The calculations for performance in the previous subsection assume that good hashing techniques exist, that is, hash tables can be used to find matching tokens from the opposite memory in a very short time. If good hashing techniques are not found, activation of an and-node may require going through long lists of tokens, impacting the performance considerably.

Overall, algorithm-2 is a big improvement over algorithm-1. It will do especially well when many changes to the working memory are processed simultaneously.

5. Algorithm-3: Using DADO as Multiple Associative Processors

Algorithm-3 was developed by the author in response to two major drawbacks of algorithm-2: (1) the number of PEs being proportional to the number of productions; (2) the memory with all PEs being large enough to store the worst case number of tokens. Algorithm-3 overcomes both these drawbacks. It is based on the observations that any single change to the working memory affects only a small number of productions, and that the number of affected productions is independent of the number of productions in the program.[12]

In algorithm-3, the complete production system program is divided into k partitions, with each partition containing 1/kth of the productions. The number of partitions, k, corresponds to the number of PEs at the PM-level of the DADO tree, and is determined by the amount of *production-level* parallelism[13] present in the program. For example, if the production-level parallelism in a program is 16, then we choose the PM-level to be 4 and divide the production system program into 16 partitions. When partitioning a production system program, similar productions (productions which are likely to be affected by the same working memory elements) are placed in different partitions. This is because we want the processing to be uniformly distributed amongst the partitions and not concentrated within any single partition.

Once the production system program has been partitioned, separate Rete networks are generated for each of the partitions. Each Rete network is then mapped onto a PM-level PE and its associated WM-subtree. The Rete network and its mapping have the following unique features. First, each memory node in the network has a unique label (ID) by which tokens associated with it can be identified. Second, the PM-level PEs do not store the top portion of the Rete network, that is, the portion between the root node and the α-mem nodes. The top-portion of the Rete network is instead distributed amongst the PEs in the WM-subtree. This permits match for satisfaction of α-mem nodes to be performed in parallel. Third, the PM-level PEs do not store tokens associated with the memory nodes in their local memory. Tokens belonging to a memory node are stored in a distributed manner amongst the PEs in the WM-subtree. This permits parallel access to all tokens

[12] Daniel Miranker at Columbia University is also considering a similar approach (personal communication).

[13] The production-level parallelism in a production system program refers to the number of distinct productions that are affected by a single change to the working memory. The production-level parallelism corresponding to multiple changes to the working memory refers to the number of distinct productions affected by all of the changes combined.

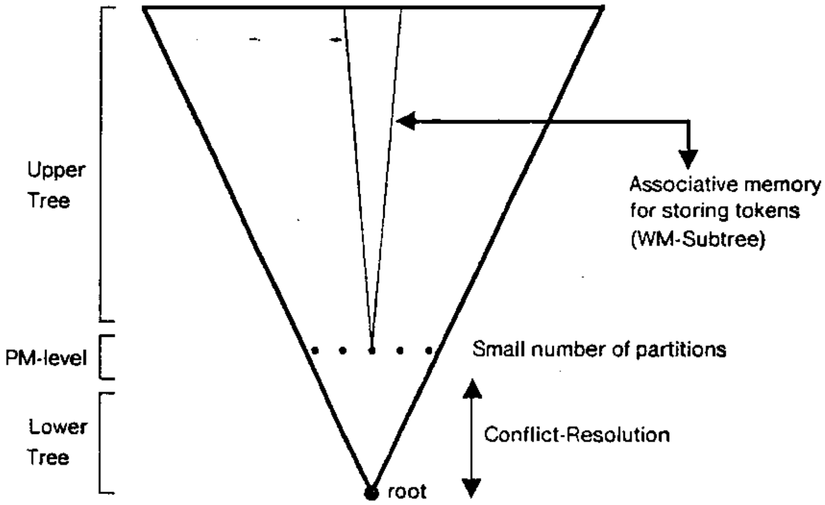


Figure 5-1: Algorithm-3

of a memory node. Fourth, with each and-node and not-node in the Rete network the compiler generates information about *relevant bindings.* Relevant bindings are those variable bindings that are accessed by subsequent two-input nodes for consistency checks.

Tokens in algorithm-3 have the following structure:

```
(<Node-ID>  <WME-IDs ...>  <relevant bindings>)
```

The *node-ID* contains the unique label of the memory node to which the token belongs. The *WME-IDs* are the IDs of the working memory elements constituting the token. The *relevant bindings* are the values of those variables that are required for the consistency checks at the two-input nodes.[14] Within a PE in the WM-subtree tokens are stored indexed by their node-IDs, as shown in Figure 5-2.

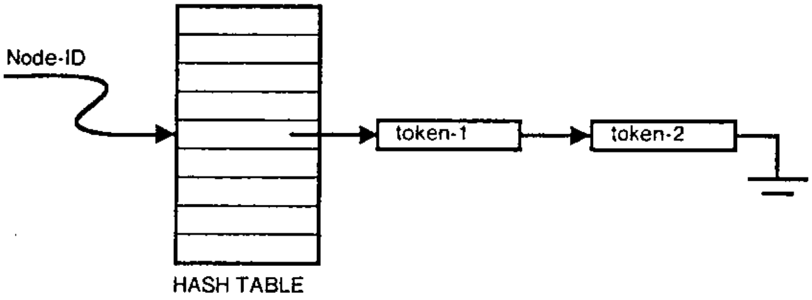


Figure 5-2: Storage of Tokens

[14] Measurements show that the number of relevant bindings is around 2-3 for most production system programs.

In algorithm-3, the addition of a working memory element is processed differently from its deletion. The steps taken to process the addition of a working memory element are:

1. Broadcast the working memory element to all PEs. The steps 2-4 below are executed in parallel within each partition.

2. PEs in the WM-subtree temporarily store the new working memory element and execute the sequence of intra-condition tests to find satisfied α-mem nodes.

3. For each newly satisfied α-mem node in the partition

   a. Fetch node-ID of the satisfied α-mem node from the PE in WM-subtree.

   b. Construct the appropriate token. Find a PE in the WM-subtree that does not already have a token for this node-ID, and store the token there.

   c. Find the two-input node activated by the α-mem node, and the node-ID of its opposite memory node.

   d. Enable all PEs in the WM-subtree having tokens for the opposite memory. Execute the consistency checks required by the two-input node. At the end of this step only those PEs that have matching tokens from the opposite memory remain enabled. The enabled PEs correspond to the activations of the β-mem node below the current two-input node. These β-mem node activations are then handled in exactly the same way as the α-mem node activations in the previous steps (3-a, 3-b, 3-c, and 3-d).

4. Find the best local instantiation within each partition.

5. Use the lower tree for conflict resolution amongst the best instantiations of each partition.

The above steps are very general and can be used to handle both the addition and deletion of working memory elements. We can use the WM-subtrees, however, to implement the deletion of working memory elements in a more efficient way. The steps are as follows:

1. Broadcast the working memory element to all PEs.

2. Determine the set of satisfied α-mem nodes. Using the set of satisfied α-mem nodes, determine the set of β-mem nodes that may potentially get affected.

3. Broadcast the node-IDs of the α-mem and β-mem nodes determined in the previous step to PEs in the WM-subtree.

4. Each PE in the WM-subtree deletes from its memory all tokens containing the WME-ID of the deleted working memory element.[15] The PEs use the node-IDs broadcast in the previous step to quickly access the potential tokens for deletion.

[15] Deletion of tokens from α-mem nodes belonging to negated condition elements requires more complex processing. We do not give the details here, but the processing is similar to when tokens are added.

5.1. Performance

In this subsection we estimate the performance of algorithm-3. We first determine the cost of processing a single change to the working memory. This cost is determined by the *worst* partition, that is, the partition taking the longest time to finish match.

Consider the case when the production system program is divided into 32 partitions. Then, we may characterize the worst partition as follows. Measurements [3] show that the total number of α-mem node activations per change to working memory is 30, which means that on average there are $30/32 = 0.94$ α-mem node activations per partition. We assume that in the worst partition there are 3 activations. The total number of β-mem node activations is 5, which means that on average there are $5/32 = 0.16$ β-mem node activations per partition. We assume that there are 2 activations in the worst partition. The total number of two-input node activations in the worst partition is $3 + 2 = 5$ (the sum of α-mem and β-mem node activations).

We now determine the cost of processing an addition to the working memory in the worst partition.

- In step-1, the working memory element is broadcast to all descendant PEs. Since the average working memory element consists of 24 attribute-value pairs (120 bytes), step-1 takes 600 instruction cycles (5 instruction cycles per byte).

- In step-2, PEs in the upper tree evaluate the intra-condition tests, thus determining the set of satisfied α-mem nodes. Assume that this takes 100 instruction cycles.

  We now deviate from the steps as listed in the previous subsection and instead calculate the cost in terms of node activations.

- Cost of α-mem node activations: There are three such activations in the worst partition. The cost of each activation includes:

  - ○ The cost of transferring the node-ID to the PM-level PE. The transfer of these 4 bytes takes 20 instruction cycles.

  - ○ The cost of constructing the new token (20 bytes) consisting of the node-ID, WME-ID, and relevant bindings. We assume this cost is 150 instruction cycles.

  - ○ The cost of finding a free PE, and shipping and storing the token there. We assume that this cost is 300 instruction cycles. (Recall that the token memory within a PE is organized as a hash table.)

  The total cost of the three α-mem activations is $(20 + 150 + 300) * 3 = 1410$ instruction cycles.

- Cost of β-mem node activations: The processing required is the same as the α-mem node activations. Since there are 2 such activations in the worst partition, the total cost is $(20 + 150 + 300) * 2 = 940$ instruction cycles.

- Cost of two-input node activations: There are 5 such activations in the worst partition. The cost of processing each activation includes:
  - ○ The cost of finding the node-ID of the opposite memory node, which is 50 instruction cycles.
  - ○ The cost of enabling all PEs in the WM-subtree having tokens for that node-ID, which is 50 instruction cycles.
  - ○ The cost of checking consistency of variable bindings. On average there is only one binding that has to be checked for consistency. This requires broadcasting and matching 4 bytes of information. Assuming 10 instruction cycles per byte for broadcasting and matching, this step takes 40 cycles.

  The total cost for this step is 5 * (50 + 50 + 40) = 700 instruction cycles.

The total cost of processing an addition to the working memory is (600 + 100 + 1410 + 940 + 700) = 3750 instruction cycles.

The cost of processing the deletion of a working memory element includes:

- The cost of broadcasting the working memory element to all PEs, which is 600 instruction cycles.
- The cost of finding affected α-mem nodes, which is 100 instruction cycles.
- The cost of retrieving their node-IDs, which is 3 * 20 = 60 instruction cycles (assuming three activations).
- The cost of finding other memory nodes which may be affected. Assume that this is 200 instruction cycles.
- The cost of broadcasting the node-IDs of possibly affected memories, which is 3 * 4 * 20 = 240 instruction cycles (assuming 4 nodes per α-mem activation).
- The cost of deleting the tokens having that WME-ID. We assume that this is 600 instruction cycles.

The total cost of deleting a working memory element is (600 + 100 + 60 + 200 + 240 + 600) = 1800 instruction cycles.

Since there are 2.5 changes in working memory per production firing, the cost of these changes is 1.25 * (3750 + 1800) = 6937 instruction cycles. Assuming that conflict resolution and right hand side evaluation take 500 instruction cycles, the time taken per firing is (6950 + 500) * 2 = 149ms. Thus, algorithm-3 will execute production system programs at the rate of 67 production firings per second.

5.2. Evaluation

The good features of algorithm-3 are:

- It exploits production-level parallelism, that is, activations of different productions are evaluated on separate hardware. This is because similar productions (productions expected to be active at the same time) are placed in different partitions.

- It exploits parallelism in evaluating and-nodes and not-nodes. It uses the PEs in the WM-subtree to access all tokens belonging to a memory node in parallel.

- It exploits parallelism in evaluating the top-portion of the Rete network. This enables algorithm-3 to determine the set of satisfied α-mem nodes in parallel.

- Deletion of working memory elements is handled more efficiently than in uniprocessor Rete. This is made possible by the ability to access all tokens of a memory node in parallel.

- Performance degrades gracefully as the size of a token memory increases. For example, in a DADO with 64 PEs per WM-subtree, the performance does not degrade at all as long as number of tokens in the memory node is less than 64. When the number of tokens is between 64 and 127, processing that memory-node and the associated two-input node takes twice as long. When the number of tokens is between 128 and 191, processing that memory-node and the associated two-input node takes thrice as long, and so on.

- Algorithm-3 does not require the majority of PEs to have large memories. This is because the tokens belonging to a memory node are distributed over the complete WM-subtree instead of being stored in the memory of a single PE. as in algorithm-2. For example, in a DADO machine with 64 PEs per WM-subtree, an α-mem node with 512 tokens will have the tokens distributed over 64 PEs, with only 8 tokens per PE. This is easily supported by the 8 kilobytes of memory present per PE.

- Large production systems can be mapped onto a DADO of fixed size without loss in performance. For example, we do not expect the performance of DADO-2 (1023 PEs) to be different for the R1 program (1932 productions) and the DAA program (131 productions). This is because the processing in algorithm-3 is activation based: that is, the work done in algorithm-3 is proportional to the number of node activations that occur in the Rete network. Since the number of node activations does not increase with number of productions in the program [3], the performance of algorithm-3 does not get worse for larger production system programs.

The weak points of the algorithm are:

- Activations of nodes within a single partition are processed sequentially. For example, if there are four α-mem node activations after an addition of a working memory element, the four activations are processed sequentially. Ideally they should be processed in parallel as in algorithm-2.

- It requires a very large memory to be associated with the PM-level PEs, thus making the hardware non-homogeneous. The large memory is required to store the complete Rete network for a partition. As the number of productions in the program increases, an increasing amount of space is required to store the network.

- The performance of algorithm-3 depends strongly on the quality of the partitioning algorithm. If a majority of node activations occur within a single partition, the performance of algorithm-3 will not be good.

6. Conclusions

In this section, we present a brief summary of the three algorithms for implementing production systems on DADO, followed by a discussion on the appropriateness of DADO architecture for OPS5-like production systems.

6.1. Summary of Algorithms

Algorithm-1 exploits production-level parallelism and has an estimated performance of 11 production firings per second. To achieve this performance, it requires approximately 100 processors per production. Although algorithm-1 exploits production-level parallelism, it performs poorly because it does not save state. The state is recomputed every cycle at a large cost.

Algorithm-2 directly maps the Rete network onto the DADO tree of processors. The algorithm exploits both production-level and node-level parallelism to have an estimated performance of about 70 production firings per second. To achieve this performance, it requires approximately 10 processors per production. The small amount of memory associated with each PE is expected to be a problem when token memories become big (over 250 tokens).

Algorithm-3 uses the DADO tree of processors as associative memory and exploits production-level parallelism to have an estimated performance of about 70 production firings per second. It does not require the number of processors to be proportional to the number of productions, and its performance does not get worse as larger production system programs are put on a fixed size DADO. However, it requires the PM-level processors to have a large amount of memory, thus making the DADO tree nonhomogeneous.

6.2. Conclusions

The DADO architecture has two characterizing features:

- *Large-scale parallelism* in the form of (potentially) hundreds of thousands of processing elements.
- The complete *binary-tree* topology.

Consider the large-scale parallelism aspect of DADO. We argue that large-scale parallelism is not appropriate for OPS5-like production systems (not just for DADO, but in general). Our argument is based on the following two observations:

- Measurements show that parallelism in OPS5-like production systems is limited [3]. This is because in current production system programs, actions of a production do not have global affects, but only affect a small number of productions (approximately 35 productions are affected by each action). Furthermore, the number of productions that is affected is independent of the number of productions in the program.

- Large-scale parallelism in an architecture has a strong influence on the power of the individual processing elements. Large-scale parallelism almost always implies that each processing element is weak, that is, has narrow datapaths, has only a small amount of memory, and has a large cycle time (because most logic families with high integration density are also slow). For example, each processing element in the prototype DADO has only 8 bit wide datapaths, has only 8 kilobytes of memory, and the instruction cycle time is $2\mu s$.

On the basis of the first observation we argue that, since the parallelism in production systems is limited, we cannot effectively use a very large number of processors. On the basis of the second observation we argue that, since we only need a few processors (because of the limited parallelism), it is much better to map the program onto a few powerful processors than to map the program onto a very large number of weak processors, of which only a few get used. For example, a uniprocessor with 32 bit datapaths and a $0.5\mu s$ instruction cycle (which is common), already has a sixteen fold advantage over the DADO processor with 8 bit datapaths and a $2\mu s$ instruction cycle. This is why, the estimated performance of DADO, 70 production firings per second, is not much higher than what we can already achieve on a current VAX-11/780 uniprocessor, 30-50 production firings per second.

The second aspect of DADO, the binary-tree topology, can be viewed in two contexts: (1) in the context of highly parallel architectures such as DADO, and (2) in the context of an architecture consisting of a small number of processors. In the context of highly parallel architectures, we do not find the binary-tree topology a limitation for implementing production system programs. This is demonstrated by algorithm-2, in which the Rete algorithm is mapped onto DADO. Neither the communication between processors, nor the mapping of the Rete network onto DADO is a problem. Furthermore, the physical layout of binary trees is easier than that of other topologies with logarithmic delay properties. In the context of a small number of processors, the topology is non-critical, because if the application warrants, we can even use a crossbar interconnection. Thus, the problems with implementing production systems are not to be solved by making minor modifications to the interconnection scheme.

Finally, we will like to reiterate the assumptions of our analysis, so that the conclusions may not be misleading: (1) The analysis has been done only for OPS5-like production systems. (2) The analysis assumes that the DADO machine uses Intel 8751 processors. The performance will be better if custom processors are used. (3) The analysis assumes that the number of processors in the DADO machine is greater than the number required for implementing the suggested algorithms. The performance will be worse if fewer processors are available. (4) The conclusions are based on the analysis of the three algorithms presented in this paper. It is possible that there are other algorithms which result in better performance.

7. Acknowledgments

I wish to thank Allen Newell, HT Kung, Charles Forgy, and John Laird for valuable comments and careful reading of early drafts of this paper. I am especially grateful to Salvatore Stolfo and Daniel Miranker of Columbia University for their help in providing data for the DADO prototype and comments on early drafts of this paper. However, their help does not imply that they agree with the results presented in this paper.

References

[1] Charles L. Forgy.
OPS5 User's Manual.
Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[2] Charles L. Forgy.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence , September 1982.

[3] Anoop Gupta and Charles L. Forgy.
Measurements on Production Systems.
Technical Report, Carnegie-Melon University, 1983.

[4] P. Haley, J. Kowalski, J. McDermott, and R. McWhorter.
PTRANS: A Rule-Based Management Assistant.
. 1983.
In preparation, Carnegie-Mellon University.

[5] Ted Kowalski and Don Thomas.
The VLSI Design Automation Assistant: Prototype System.
In *Proceedings of the 20th Design Automation Conference.* ACM and IEEE, June, 1983.

[6] John Laird and Allen Newell.
A Universal Weak Method: Summary of Results.
In *International Joint Conference on Artificial Intelligence.* 1983.

[7] John McDermott.
R1: A Rule-based Configurer of Computer Systems.
Technical Report CMU-CS-80-119, Carnegie-Mellon University, April, 1980.

[8] John McDermott.
XSEL: A Computer Salesperson's Assistant.
In J.E. Hayes, D. Michie, and Y.H. Pao (editor), *Machine Intelligence.* Horwood, 1982.

[9] Paul S. Rosenbloom.
The Chunking of Goal Hierarchies: A Model of Stimulus-Response Compatibility.
PhD thesis, Carnegie-Mellon University, August, 1983.

[10] E. H. Shortliffe.
Computer-Based Medical Consultations: MYCIN.
North-Holland, 1976.

[11] Mark Stefik, et al.
The Organization of Expert Systems: A Prescriptive Tutorial.
Technical Report VLSI-82-1, XEROX, Palo Alto Research Center, January, 1982.

[12] Salvatore J. Stolfo, Daniel Miranker, and David E. Shaw.
Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence.
In *International Joint Conference on Artificial Intelligence.* 1983.

[13] Salvatore J. Stolfo and David E. Shaw.
DADO: A Tree-Structured Machine Architecture for Production Systems.
In *National Conference on Artificial Intelligence.* 1982.

[14] Salvatore J. Stolfo, Daniel Miranker, and David E. Shaw.
Programming the DADO Machine: An Introduction to PPL/M.
Technical Report, Columbia University, November, 1982.

[15] Stephen Taylor, Christopher Maio, Salvatore J. Stolfo, and David E. Shaw.
PROLOG on the DADO Machine: A Parallel System for High-Speed Logic Programming.
Technical Report, Columbia University, January, 1983.

[16] Stephen Taylor, et al.
Logic Programming using Parallel Associative Operators.
Technical Report, Columbia University, August, 1983.