CUCS-109-84

Simultaneous Firing of Production Rules on Tree Structured Machines[1]

Toru Ishida*
and
Salvatore J. Stolfo

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

*Visiting from Yokosuka Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation

28 March 1984

Abstract

This paper describes a method to realize the simultaneous firing of production rules on tree structured machines. We propose a simultaneous firing mechanism consisting of global communication and global synchronization between subtrees. We also propose a hierarchical decomposition algorithm for production systems which maximizes total throughput by satisfying two requirements, i.e. maximizing parallel executability and minimizing global communication.

[1]This research has been supported by the Defense Advanced Research Projects Agency through contact N00039-84-C-0165, as well as grants from Intel, Digital Equipment, Hewlett-Packard Valid Logic Systems, AT&T Bell Laboratories and IBM Corporations and the New York State Science and Technology Foundation. We gratefully acknowledge their support.

Table of Contents

1 Introduction

Tree structured machines have been studied and constructed for parallel execution of production systems. Stolfo[1] and Miranker[2] invented several tree structured machine oriented matching algorithms for the DADO machine[3]. Gupta[4] proposed a method to implement the Rete Match algorithm[5], which is used in OPS5[6], on tree structured machines.

This paper is mainly concerned with simultaneous firing of production rules on a tree structured machine. Two problems are discussed in this paper, i.e. how to fire production rules simultaneously, and how to decompose and allocate production rules to many processors. Both problems are solved by using data dependency analysis of production rules.

The simultaneous firing mechanism consists of the following functions.

- Global communication, which is required when a particular processor executes the rule and sends the change of working memory to other processors.
- Global synchronization, which is required when simultaneous firing causes interference and produces a different result from the sequential execution of these rules.

In order to increase the effectiveness of simultaneous firing, the decomposition algorithm of production rules should satisfy the following requirements.

- Maximize parallel executability. There are two kinds of parallelism in production rules. One is fully parallel execution without any data passing between rules, and the other is pipeline execution with a continuous stream of data passing between rules.
- Minimize global communication. It is often pointed out that the effective bandwidth of communication is restricted by the top of the tree (otherwise known as the "binary tree bottleneck").

In this paper, we propose a hierarchical decomposition algorithm for production systems which satisfies the above two requirements.

2 Basic Definitions and Concepts

2.1 Production Systems

A ***production system*** (*PS*) is defined by a set of rules together with a database of assertions, called the ***working memory*** (*WM*). Each production consists of a conjunction of conditional elements, called the ***left-hand side*** (*LHS*) of the rule, along with a set of actions called the ***right-hand side*** (*RHS*). The RHS specifies information which is to be added to or removed from WM when the LHS successfully matches against the contents of WM.

The PS repeatedly executes the following cycle of operations on conventional machines.

- *Match*: For each rule, determine whether the LHS matches the current environment of WM.
- *Select*: Choose exactly one of the matching rules according to some predefined criterion.
- *Act*: Add to or delete from WM all assertions as specified by the RHS of the selected rule.

In this paper, however, we do not assume that only one rule is chosen in the Select phase, but rather propose to execute a considerable number of matching rules simultaneously on a tree structured machine.

2.2 Tree Structured Machines

The ***tree structured machine*** comprises a very large set of processing elements (PEs). Each PE has its own local memory, and can execute its own programs. However, there is no global memory, so that communication is only allowed between the adjacent tree neighbors.

In this paper, we assume that the tree structured machine is functionally divided into two layers, i.e. an ***upper layer*** and a ***lower layer***:

- A lower layer consists of many subtrees, each of which contains a group of production rules and relevant WM elements.
- An upper layer controls global communication and global synchronization between lower layer subtrees.

3 Overview of Decomposition and Allocation Process of Production Rules

3.1 Data Dependency Graph

Data dependency graphs are often used to analyze parallelism in microprograms[7,8] or to represent control structures in non-procedural languages[9,10]. To analyze production rules, we introduce a data dependency graph of production rules which is slightly modified for our own purpose.

A *data dependency graph of production rules* is made of the following three primitives.

- A *production node* (a *P-node*), which represents a production rule.
- A *working memory node* (a *W-node*), which represents a *class* of working memory elements.
- A *directed edge* (or simply an *edge*), which represents a data dependency. There are two kinds of edges.
  - A *directed edge from P-node to W-node*, which represents that the RHS of a production rule modifies (adds or deletes) a class of working memory elements.
  - A *directed edge from W-node to P-node*, which represents that the LHS of a production rule refers to a class of working memory elements.

Fig. 3.1 shows an example of a data dependency graph.

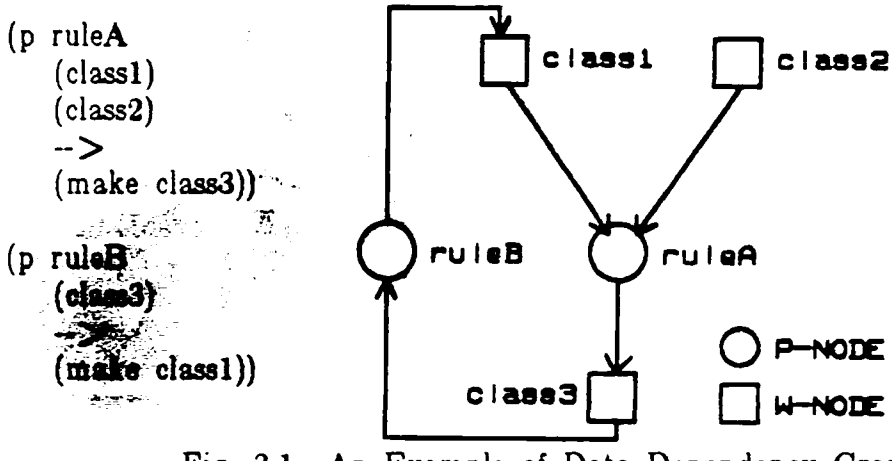


Fig. 3.1 An Example of Data Dependency Graph

3.2 Decomposition and Allocation Process

The decomposition and allocation process of production rules recursively proceeds as follows.

- Decompose the production rules into two groups; allocate one group to the left subtree, and the other to the right subtree.
- Repeat the process in each subtree.

Production rules are represented by a data dependency graph. To decompose the data dependency graph into two groups, the necessary number of W-nodes should be split. The split W-nodes represent the same copy of the original W-node. Thus if a particular W-node is split, the WM elements in that class are stored in both of the right and left subtrees. Fig. 3.2 illustrates this hierarchical decomposition and allocation process.

```
(p ruleA
   (class2)
   -->
   (make class1))
(p ruleB
  -(class1)
   (class2)
   -->
   (make class4))
(p ruleC
   (class3)
   -->
   (make class2))
(p ruleD
  -(class5)
   -->
   (remove class2)
   (make class3))
```

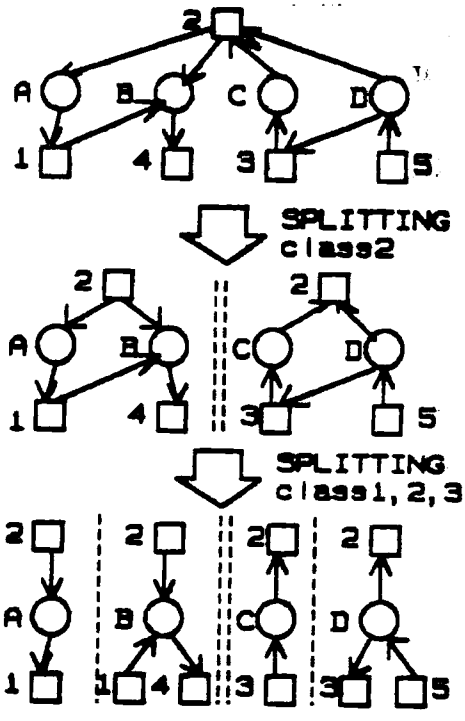


Fig. 3.2 An Example of Data Dependency Graph Decomposition

4 Simultaneous Firing Mechanism

4.1 Global Communication Mechanism

Communication and global communication between two rules are defined as follows.

- If rule B refers to a WM class which is changed by rule A, we say there

is *communication from rule A to rule B*, and represent it by *com(A->B)*.

- If com(A->B) and if rule A and rule B are allocated in different lower subtrees, then the changes of WM must be communicated from rule A to rule B by passing through an upper layer. This kind of communication is called *global communication from rule A to rule B*, and represented by *g-com(A->B)*.

Global communication is processed as follows.

- When working memory elements in a split WM class are modified in some lower subtree, changes are reported to the necessary level of an upper layer.
- Then the changes are broadcast to every lower subtree which contains the split WM class.

Since production rules are decomposed in a hierarchical manner, only a small number of changes are reported to the root node. Fig. 4.1 illustrates global communication among the production rules of Fig. 3.2.

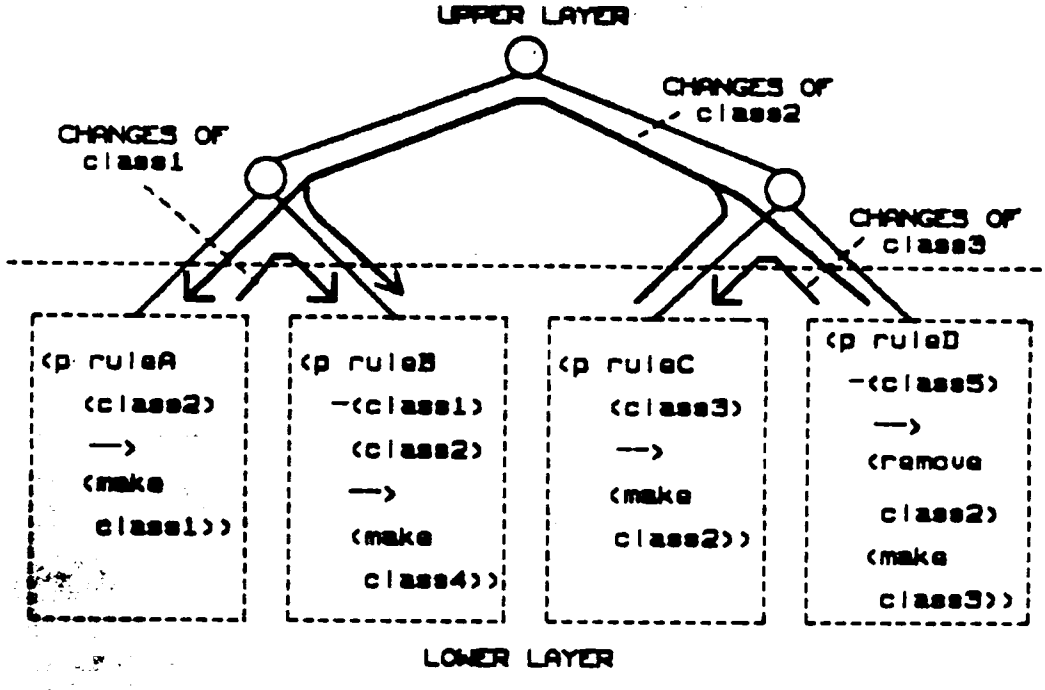


Fig. 4.1 An Example of Global Communication

4.2 Global Synchronization Mechanism

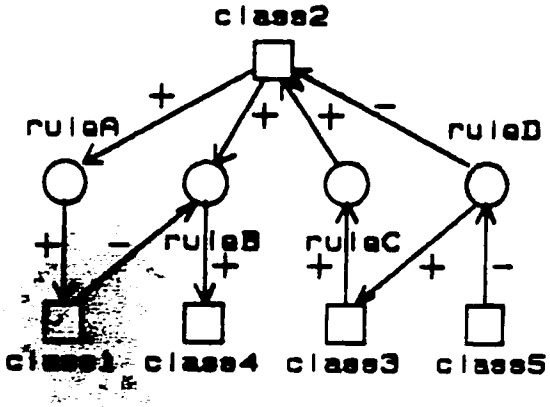
Synchronization between two rules is defined as follows.

- If the result of simultaneous firing of rule A and rule B is different from the result of sequential firings in any order, we say ***synchronization is required between rule A and rule B***, and represent it by ***syn(A<->B)***.
- ***Synchronization set of rule A*** is defined as the set of rules which require synchronization with rule A.

In order to analyze the synchronization requirements on a data dependency graph, we first label '+' or '-' on each directed edge by the following operation.

- If the edge originates at a P-node and terminates at a W-node then label
  - '+', when the rule adds WM elements of the class.
  - '-', when the rule deletes WM elements of the class.
- If the edge originates at a W-node and terminates at a P-node then label
  - '+', when the class is referenced by a positive conditional element of the rule.
  - '-', when the class is referenced by a negative conditional element of the rule.
- The edge which has both of '+' and '-' label is split into two edges, each of which has '+' or '-' label.

Fig. 4.2(1) shows an example of the labeled data dependency graph.



ruleA: (ruleB, ruleD)

ruleB: (ruleA, ruleD)

ruleC: (ruleD)

ruleD: (ruleA, ruleB, ruleC)

(1) Labelled Data Dependency Graph (2) Synchronization Set

Fig. 4.2 An Example of Synchronization Analysis

The following observations can be derived from the labeled data dependency graph.

- If all WM classes lying between rule A and rule B are either '+'(changed

by rule A) and '+'(referred to by rule B), or '-'(changed by rule A) and '-'(referred to by rule B), then the firing possibility of rule B is increased monotonously by executing rule A. Thus, even if rule A is fired during the execution of rule B, interference never occurs.

- Conversely, if some WM classes lying between rule A and rule B are '+'(changed by rule A) and '-'(referred to by rule B), or '-'(changed by rule A) and '+'(referred to by rule B), then the firing possibility of rule B is sometimes decreased. In this case, synchronization is necessary.
- If rule A and rule B change the same WM class, and if the class is '+'(changed by rule A) and '-'(changed by rule B), or '-'(changed by rule A) and '+'(changed by rule B), then the result of simultaneous firing is sometimes different from the result of sequential execution.

From the above observation, we can say that synchronization between rule A and rule B is required if the following conditions are satisfied on the data dependency graph.

- syn(A<->B) is satisfied if there exists a WM class, which is
  - '+'(changed by rule A) and '-'(referred to by rule B), or
  - '-'(changed by rule A) and '+'(referred to by rule B), or
  - '+'(changed by rule B) and '-'(referred to by rule A), or
  - '-'(changed by rule B) and '+'(referred to by rule A), or
  - '+'(changed by rule A) and '-'(changed by rule B), or
  - '-'(changed by rule A) and '+'(changed by rule B).

Fig. 4.2(2) shows an example of synchronization sets obtained by applying the above conditions to the production rules in Fig 4.2(1).

Now, global synchronization can be defined as follows.

- If syn(A<->B), and if rule A and rule B are allocated in different lower subtrees, then synchronizing requests are sent from rule A to rule B or from rule B to rule A by passing through an upper layer. This kind of synchronization is called *global synchronization between rule A and rule B*, and represented by *g-syn(A<->B)*.

Global synchronization is processed as follows.

- When rule A, whose global synchronization set is non-empty, is fired in some lower subtree, the request for the global synchronization is sent to the necessary level of an upper layer.
- The request is broadcast to every lower subtree which contains a rule in the global synchronization set of rule A. Then, the firings of the interference rules are suspended. If interference rules are currently executed, their firing is suspended immediately after the current execution has finished.

- Rule A is executed.
- The firing suspension is released in every subtree.

Because the data dependency graph is decomposed in a hierarchical manner, only a small number of global synchronizations are requested of the root node.

From the above discussion, the condition for simultaneous firing is derived as follows.

- If not syn(A<->B), rule A and rule B can be fired simultaneously. In this case, we say ***rule A and rule B are parallel executable.***

Two kinds of parallel executions, fully parallel execution and pipeline execution, are realized by simultaneous firing.

5 Decomposition Algorithm for Production Rules

5.1 Merits and Demerits of Decomposition

We first discuss the merits and demerits derived from decomposition of production rules. Clearly decomposition is intended to reduce the execution time. For simplicity, we assume the same execution time for all production rules, and call that execution time a ***production cycle***. Thus the merit of decomposition can be expressed by the number of reduced production cycles (represented by P) obtained by simultaneous firing.

On the other hand, the drawback of decomposition is increased global communication (represented by C). We define a ***global communication unit*** as one WM element communication between physically adjacent PEs. For example, one WM element communication between sibling subtrees costs 2 units.

The global communication cost depends on the following situations.

- It depends on the decomposition stage. The PS is decomposed through n stages and allocated on 2^n lower subtrees. If a WM class is split in the first stage, global communication is attained through the root node. However, if the splitting is done in the last stage, global communication is limited within adjacent lower subtrees. If a WM class is split in the i-th stage, one WM element communication costs 2(n-i+1) units.
- It also depends on the decomposition history. If a WM class is split by more than one stage, then only half the cost is required in the second or later stages.

From the above discussion, the total gain of decomposition (represented by G) can be calculated by the following equation.

$$G = c_1 P - c_2 C, \qquad \text{where } c_1 \text{ and } c_2 \text{ are appropriate coefficients.}$$

5.2 Evaluation Algorithm for a Particular Partition

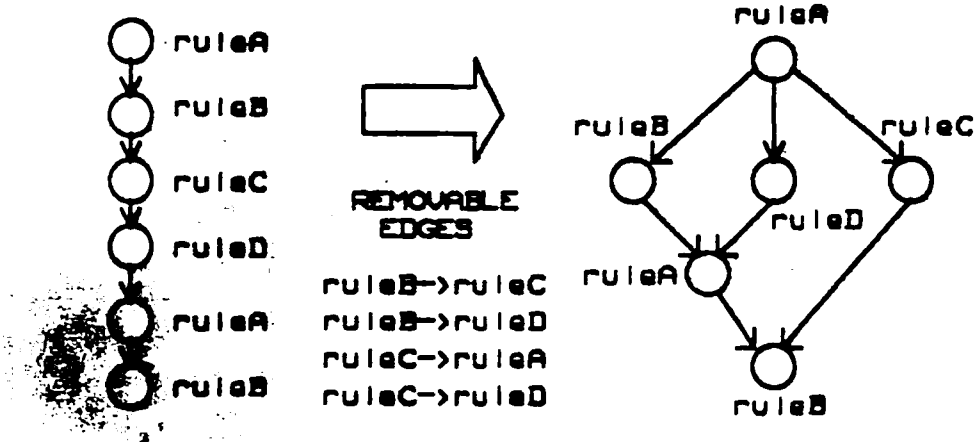
In this subsection, we describe how to evaluate the total gain of a particular partition. We use sample execution traces to calculate the total gain, because the quantity of total gain can not be obtained only by static analysis. The evaluation algorithm is described below.

Step1: Building the Initial Trace Graph

We first define the ***trace graph***, which is made of the following two primitives.

- A ***node***, which represents a production rule firing and
- A ***directed edge***, which represents the firing order of two production rules.

The ***initial trace graph*** can be easily constructed from sample execution traces, by creating nodes and connecting them with directed edges in the original execution order [10]. Fig . 5.1(1) represents an example of initial trace graph.



(1) Initial Trace Graph (2) Transformed Trace Graph

Fig. 5.1 Trace Graph

Step2: Transforming the Trace Graph

Directed edges represent the firing order of production rules. If two firings can be done simultaneously, we can remove the edge between these firings. Conditions and operations for removing edges are as follows.

- If (1) not syn(A<->B) and (2) not com(A->B) then
  - (1) delete an edge which goes from A to B, and
  - (2) add edges which go from A's predecessors to B, and
  - (3) add edges which go from A to B's successors.

Condition (1) indicates the parallel executability of two rules. However, parallel executability does not directly imply that successive two firings actually occur simultaneously. This is the case because if there exists communication between successive two firings, it should be considered that the latter firing is the result of the former firing. Thus condition (2) is necessary. Operation (2) and (3) preserve the order for two firings in connection with other firings.

The ***transformed trace graph***, which represents parallel executability, is obtained by applying the above operation to all edges. Fig 5.1(2) shows the result of this transformation.

Step3: Simulation

The final step of the evaluation is a simulation of simultaneous firing on a given partition. The simulation algorithm is as follows.

- (1) Set C (which represents the global communication cost) to 0 and set P (which represents the reduced production cycles) to the number of original production cycles. Calculate the cost of one WM element communication for every split WM class.
- (2) List the nodes which have no predecessors in the trace graph. If the list is empty, then simulation terminates.
- (3) Classify the listed nodes into the following three groups.
  - Group R: Nodes in this group should be executed in the right subtree.
  - Group L: Nodes in this group should be executed in the left subtree.
  - Group OTHER: Nodes in this group are not objects of current decomposition, i.e. these nodes are executed outside of the current tree.
- (4) If group OTHER is not empty, delete all nodes in group OTHER from the transformed trace graph and go back to (2).
- (5) Choose one rule from group R according to some predefined criterion and delete it from the transformed trace graph. If the rule changes some split WM classes, then count the global communication cost and add to C.

- (6) Choose one rule from group L and do same as (5).
- (7) Decrement P and go back to (1).

Total gain G is obtained from P and C. By applying the evaluation algorithm to all partitioned candidates, we can obtain the best partition. However, this approach is possible only if there exist only a small number of partitioned candidates.

5.3 Practical Decomposition Algorithm

The practical decomposition algorithm reduces the computational complexity by the following strategies.

- First, by approximating the total gain of the decomposition by summing the gains obtained from decomposing every rule pair.
- Second, by not considering every possible partition of the rule set.

The practical decomposition algorithm is described below.

Step1: Calculate Gain of Decomposing Rule Pair

Reduced production cycles (represented by $P^*(A,B)$) and global communication cost (represented by $C^*(A,B)$), which are caused by decomposing two rules (A and B), are calculated without considering other rules by using sample execution traces.

The total gain of decomposing rule A and rule B (represented by $G^*(A, B)$) is expressed by the following equation.

$$G^*(A,B) = c_1 P^*(A, B) - c_2 C^*(A,B)$$

Step2: Allocating Rules One by One

To obtain a nearly optimal partition in an incremental manner, the most influential rule pair should be first allocated. The allocating algorithm proceeds as follows.

- (1) Make a list of all rule pairs and sort it in decreasing order of $|G^*(I,J)|$.
- (2) Repeat the following steps until all rules are allocated.
    - (2-1) Pop the first rule pair (I,J) from the list, and allocate it as follows.
        - If $G^*(I,J) >= 0$, allocate I and J to different subtrees.

- If $G^*(I,J) < 0$, allocate I and J to the same subtree.

- (2-2) Scan the rule pair list from first to last, and do the following for each rule pair (I,J). If all pairs have been examined, go back to (2-1).
  - If both of I and J have been allocated, remove the pair from the list.
  - If both of I and J have not been allocated, simply go on to the next pair.
  - If one of I and J has been allocated, do as follows. Then remove the pair from the list and go back to (2-2).
    - If $G^*(I,J) >= 0$, allocate I and J to different subtrees.
    - If $G^*(I,J) < 0$, allocate I and J to the same subtree.

6 Conclusion

The main results of this research are as follows.

- We show how production rules are decomposed and allocated on a tree structured machine by use of a hierarchical decomposition algorithm. This algorithm provides a solution to the binary tree bottleneck problem.
- We clarify the mechanisms which are necessary to realize simultaneous firings of production rules, i.e. the global communication and synchronization mechanisms. We also show these mechanism are effective for both fully parallel execution and pipeline execution.
- We propose a practical decomposition algorithm of production rules. This algorithm calculates both the merits and demerits of decomposition, and produces a nearly optimal solution. The algorithm is also applicable to the decomposition of large scale expert systems. In this case, one node of a data dependency graph or trace graph represents not only one rule but a rule set.

This research has been conducted as part of the research of the DADO tree structured machine. The next step is the implementation and evaluation of a simultaneous firing mechanism in the actual DADO environment.

Acknowledgments

We would like to thank Dan Miranker and Mark Lerner for their comments on an earlier version of this paper.

References

1. Stolfo, S. J. and Shaw, D. E., "DADO: A Tree Structured Machine Architecture for Production Systems", In Proceedings of the National Conference of Artificial Intelligence, 1982.

2. Miranker, D. P., "Performance Estimates for the DADO Machine", Technical Report, Department of Computer Science, Columbia University, 1984 (in preparation).

3. Stolfo, S. J., "The DADO Parallel Computer", Technical Report, Department of Computer Science, Columbia University, 1983.

4. Gupta, A., "Implementing OPS5 Production System on DADO", Technical Report, Carnegie Mellon University, 1983.

5. Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, Sep. 1982.

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

7. Tsuchiya, M. and Gonzalez, M. J., "Toward Optimization of Horizontal Microprograms", IEEE Trans. Comput., vol. C-25, pp.992-999, Oct. 1976.

8. Isoda, S., Kobayashi, Y. and Ishida, T., "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph", IEEE Trans. Comput., vol. C-32, pp.922-933, Oct. 1983.

9. Charniac, E., Riesbeck, C. K. and McDermott, D. V., "Artificial Intelligence Programming", Laurence Erlbaum Associates, pp. 193-226, 1980.

10. Stolfo, S. J., "Automatic Discovery of Heuristics for Nondeterministic Programs from Sample Execution Traces", Ph.D. Th., New York University, 1979.

Simultaneous Firing of Production Rules on Tree Structured Machines[1]

Toru Ishida*
and
Salvatore J. Stolfo
Department of Computer Science
Columbia University
New York City, N.Y. 10027

*Visiting from Yokosuka Electrical Communication Laboratory
Nippon Telegraph and Telephone Public Corporation

28 March 1984

Abstract

This paper describes a method to realize the simultaneous firing of production rules on tree structured machines. We propose a simultaneous firing mechanism consisting of global communication and global synchronization between subtrees. We also propose a hierarchical decomposition algorithm for production systems which maximizes total throughput by satisfying two requirements, i.e. maximizing parallel executability and minimizing global communication.

[1]This research has been supported by the Defense Advanced Research Projects Agency through contact N00039-84-C-0165, as well as grants from Intel, Digital Equipment, Hewlett-Packard Valid Logic Systems, AT&T Bell Laboratories and IBM Corporations and the New York State Science and Technology Foundation. We gratefully acknowledge their support.

Table of Contents

1 Introduction

Tree structured machines have been studied and constructed for parallel execution of production systems. Stolfo[1] and Miranker[2] invented several tree structured machine oriented matching algorithms for the DADO machine[3]. Gupta[4] proposed a method to implement the Rete Match algorithm[5], which is used in OPS5[6], on tree structured machines.

This paper is mainly concerned with simultaneous firing of production rules on a tree structured machine. Two problems are discussed in this paper, i.e. how to fire production rules simultaneously, and how to decompose and allocate production rules to many processors. Both problems are solved by using data dependency analysis of production rules.

The simultaneous firing mechanism consists of the following functions.

- Global communication, which is required when a particular processor executes the rule and sends the change of working memory to other processors.
- Global synchronization, which is required when simultaneous firing causes interference and produces a different result from the sequential execution of these rules.

In order to increase the effectiveness of simultaneous firing, the decomposition algorithm of production rules should satisfy the following requirements.

- Maximize parallel executability. There are two kinds of parallelism in production rules. One is fully parallel execution without any data passing between rules, and the other is pipeline execution with a continuous stream of data passing between rules.
- Minimize global communication. It is often pointed out that the effective bandwidth of communication is restricted by the top of the tree (otherwise known as the "binary tree bottleneck").

In this paper, we propose a hierarchical decomposition algorithm for production systems which satisfies the above two requirements.

2 Basic Definitions and Concepts

2.1 Production Systems

A ***production system*** (*PS*) is defined by a set of rules together with a database of assertions, called the ***working memory*** (*WM*). Each production consists of a conjunction of conditional elements, called the ***left-hand side*** (*LHS*) of the rule, along with a set of actions called the ***right-hand side*** (*RHS*). The RHS specifies information which is to be added to or removed from WM when the LHS successfully matches against the contents of WM.

The PS repeatedly executes the following cycle of operations on conventional machines.

- *Match*: For each rule, determine whether the LHS matches the current environment of WM.
- *Select*: Choose exactly one of the matching rules according to some predefined criterion.
- *Act*: Add to or delete from WM all assertions as specified by the RHS of the selected rule.

In this paper, however, we do not assume that only one rule is chosen in the Select phase, but rather propose to execute a considerable number of matching rules simultaneously on a tree structured machine.

2.2 Tree Structured Machines

The ***tree structured machine*** comprises a very large set of processing elements (PEs). Each PE has its own local memory, and can execute its own programs. However, there is no global memory, so that communication is only allowed between the adjacent tree neighbors.

In this paper, we assume that the tree structured machine is functionally divided into two layers, i.e. an ***upper layer*** and a ***lower layer***:

- A lower layer consists of many subtrees, each of which contains a group of production rules and relevant WM elements.
- An upper layer controls global communication and global synchronization between lower layer subtrees.

3 Overview of Decomposition and Allocation Process of Production Rules

3.1 Data Dependency Graph

Data dependency graphs are often used to analyze parallelism in microprograms[7,8] or to represent control structures in non-procedural languages[9,10]. To analyze production rules, we introduce a data dependency graph of production rules which is slightly modified for our own purpose.

A *data dependency graph of production rules* is made of the following three primitives.

- A *production node* (a *P-node*), which represents a production rule.
- A *working memory node* (a *W-node*), which represents a *class* of working memory elements.
- A *directed edge* (or simply an *edge*), which represents a data dependency. There are two kinds of edges.
  - A *directed edge from P-node to W-node*, which represents that the RHS of a production rule modifies (adds or deletes) a class of working memory elements.
  - A *directed edge from W-node to P-node*, which represents that the LHS of a production rule refers to a class of working memory elements.

Fig. 3.1 shows an example of a data dependency graph.

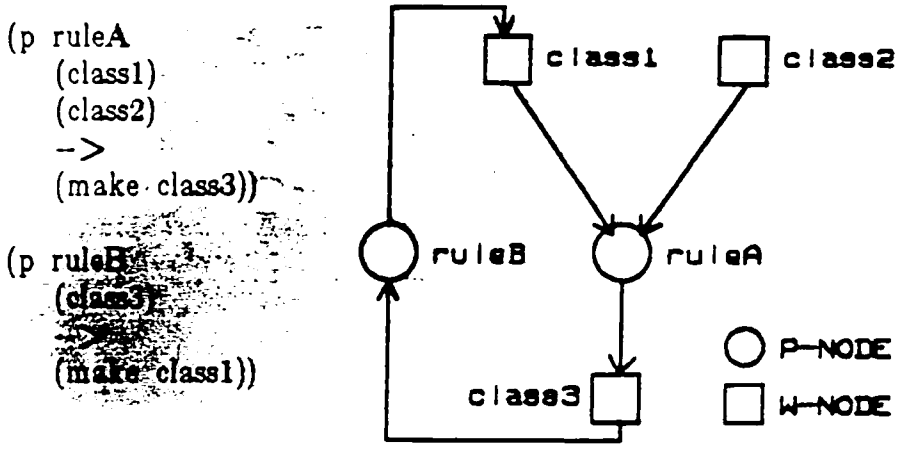


Fig. 3.1 An Example of Data Dependency Graph

3.2 Decomposition and Allocation Process

The decomposition and allocation process of production rules recursively proceeds as follows.

- Decompose the production rules into two groups; allocate one group to the left subtree, and the other to the right subtree.
- Repeat the process in each subtree.

Production rules are represented by a data dependency graph. To decompose the data dependency graph into two groups, the necessary number of W-nodes should be split. The split W-nodes represent the same copy of the original W-node. Thus if a particular W-node is split, the WM elements in that class are stored in both of the right and left subtrees. Fig. 3.2 illustrates this hierarchical decomposition and allocation process.

```
(p ruleA
   (class2)
   -->
   (make class1))
(p ruleB
  -(class1)
   (class2)
   -->
   (make class4))
(p ruleC
   (class3)
   -->
   (make class2))
(p ruleD
  -(class5)
   -->
   (remove class2)
   (make class3))
```

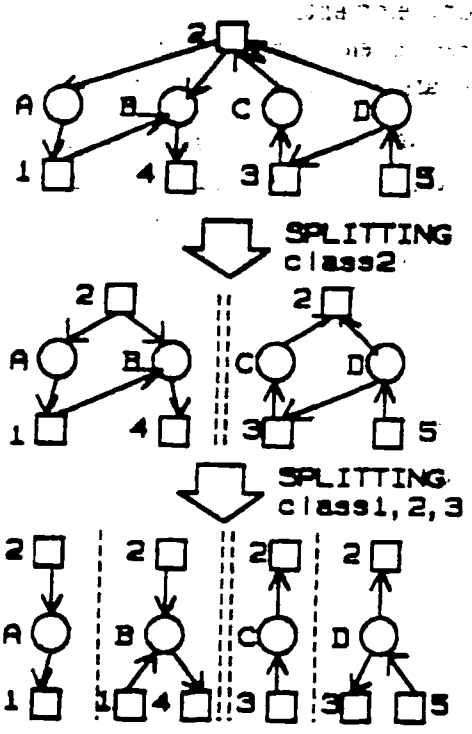


Fig. 3.2 An Example of Data Dependency Graph Decomposition

4 Simultaneous Firing Mechanism

4.1 Global Communication Mechanism

Communication and global communication between two rules are defined as follows.

- If rule B refers to a WM class which is changed by rule A, we say there

is *communication from rule A to rule B*, and represent it by *com(A->B)*.

- If com(A->B) and if rule A and rule B are allocated in different lower subtrees then the changes of WM must be communicated from rule A to rule B by passing through an upper layer. This kind of communication is called *global communication from rule A to rule B*, and represented by *g-com(A->B)*.

Global communication is processed as follows.

- When working memory elements in a split WM class are modified in some lower subtree, changes are reported to the necessary level of an upper layer.
- Then the changes are broadcast to every lower subtree which contains the split WM class.

Since production rules are decomposed in a hierarchical manner, only a small number of changes are reported to the root node. Fig. 4.1 illustrates global communication among the production rules of Fig. 3.2.

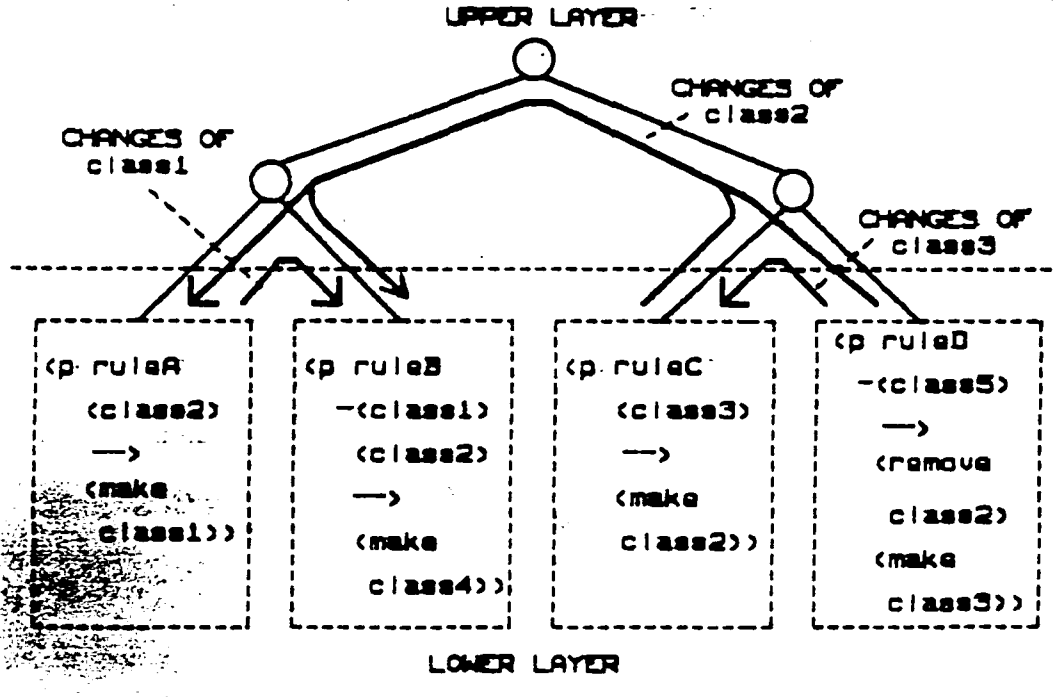


Fig. 4.1 An Example of Global Communication

4.2 Global Synchronization Mechanism

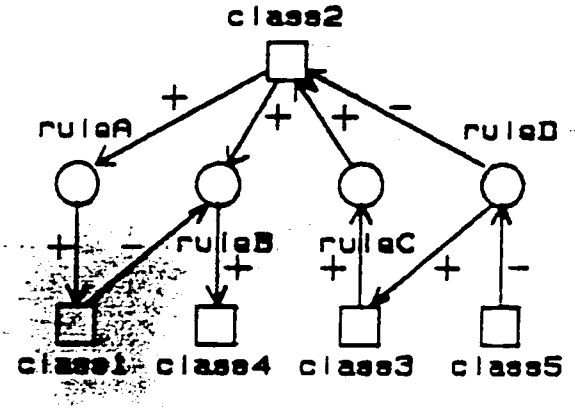
Synchronization between two rules is defined as follows.

- If the result of simultaneous firing of rule A and rule B is different from the result of sequential firings in any order, we say ***synchronization is required between rule A and rule B***, and represent it by ***syn(A<->B)***.
- ***Synchronization set of rule A*** is defined as the set of rules which require synchronization with rule A.

In order to analyze the synchronization requirements on a data dependency graph, we first label '+' or '-' on each directed edge by the following operation.

- If the edge originates at a P-node and terminates at a W-node then label
  - '+', when the rule adds WM elements of the class.
  - '-', when the rule deletes WM elements of the class.
- If the edge originates at a W-node and terminates at a P-node then label
  - '+', when the class is referenced by a positive conditional element of the rule.
  - '-', when the class is referenced by a negative conditional element of the rule.
- The edge which has both of '+' and '-' label is split into two edges, each of which has '+' or '-' label.

Fig. 4.2(1) shows an example of the labeled data dependency graph.



ruleA: (ruleB, ruleD)

ruleB: (ruleA, ruleD)

ruleC: (ruleD)

ruleD: (ruleA, ruleB, ruleC)

(1) Labelled Data Dependency Graph (2) Synchronization Set

Fig. 4.2 An Example of Synchronization Analysis

The following observations can be derived from the labeled data dependency graph.

- If all WM classes lying between rule A and rule B are either '+'(changed

by rule A) and '+'(referred to by rule B), or '-'(changed by rule A) and '-'(referred to by rule B), then the firing possibility of rule B is increased monotonically by executing rule A. Thus, even if rule A is fired during the execution of rule B, interference never occurs.
- Conversely, if some WM classes lying between rule A and rule B are '+'(changed by rule A) and '-'(referred to by rule B), or '-'(changed by rule A) and '+'(referred to by rule B), then the firing possibility of rule B is sometimes decreased. In this case, synchronization is necessary.
- If rule A and rule B change the same WM class, and if the class is '+'(changed by rule A) and '-'(changed by rule B), or '-'(changed by rule A) and '+'(changed by rule B), then the result of simultaneous firing is sometimes different from the result of sequential execution.

From the above observation, we can say that synchronization between rule A and rule B is required if the following conditions are satisfied on the data dependency graph.

- syn(A<->B) is satisfied if there exists a WM class, which is
  - '+'(changed by rule A) and '-'(referred to by rule B), or
  - '-'(changed by rule A) and '+'(referred to by rule B), or
  - '+'(changed by rule B) and '-'(referred to by rule A), or
  - '-'(changed by rule B) and '+'(referred to by rule A), or
  - '+'(changed by rule A) and '-'(changed by rule B), or
  - '-'(changed by rule A) and '+'(changed by rule B).

Fig. 4.2(2) shows an example of synchronization sets obtained by applying the above conditions to the production rules in Fig 4.2(1).

Now, global synchronization can be defined as follows.

- If syn(A<->B), and if rule A and rule B are allocated in different lower subtrees, then synchronizing requests are sent from rule A to rule B or from rule B to rule A by passing through an upper layer. This kind of synchronization is called *global synchronization between rule A and rule B*, and represented by *g-syn(A<->B)*.

Global synchronization is processed as follows.

- When rule A, whose global synchronization set is non-empty, is fired in some lower subtree, the request for the global synchronization is sent to the necessary level of an upper layer.
- The request is broadcast to every lower subtree which contains a rule in the global synchronization set of rule A. Then, the firings of the interference rules are suspended. If interference rules are currently executed, their firing is suspended immediately after the current execution has finished.

- Rule A is executed.
- The firing suspension is released in every subtree.

Because the data dependency graph is decomposed in a hierarchical manner, only a small number of global synchronizations are requested of the root node.

From the above discussion, the condition for simultaneous firing is derived as follows.

- If not syn(A<->B), rule A and rule B can be fired simultaneously. In this case, we say ***rule A and rule B are parallel executable.***

Two kinds of parallel executions, fully parallel execution and pipeline execution, are realized by simultaneous firing.

5 Decomposition Algorithm for Production Rules

5.1 Merits and Demerits of Decomposition

We first discuss the merits and demerits derived from decomposition of production rules. Clearly decomposition is intended to reduce the execution time. For simplicity, we assume the same execution time for all production rules, and call that execution time a ***production cycle***. Thus the merit of decomposition can be expressed by the number of reduced production cycles (represented by P) obtained by simultaneous firing.

On the other hand, the drawback of decomposition is increased global communication (represented by C). We define a ***global communication unit*** as one WM element communication between physically adjacent PEs. For example, one WM element communication between sibling subtrees costs 2 units.

The global communication cost depends on the following situations.

- It depends on the decomposition stage. The PS is decomposed through n stages and allocated on 2^n lower subtrees. If a WM class is split in the first stage, global communication is attained through the root node. However, if the splitting is done in the last stage, global communication is limited within adjacent lower subtrees. If a WM class is split in the i-th stage, one WM element communication costs $2(n-i+1)$ units.
- It also depends on the decomposition history. If a WM class is split by more than one stage, then only half the cost is required in the second or later stages.

From the above discussion, the total gain of decomposition (represented by G) can be calculated by the following equation.

$G = c_1 P - c_2 C$, where c_1 and c_2 are appropriate coefficients.

5.2 Evaluation Algorithm for a Particular Partition

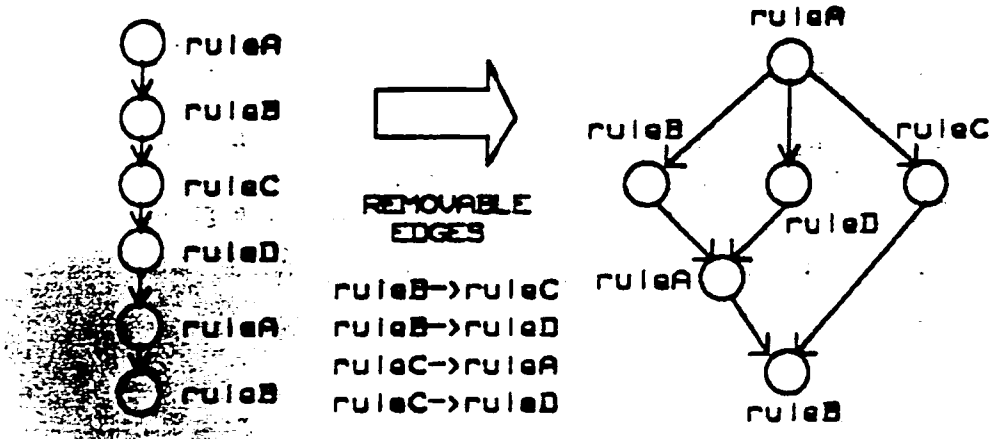
In this subsection, we describe how to evaluate the total gain of a particular partition. We use sample execution traces to calculate the total gain, because the quantity of total gain can not be obtained only by static analysis. The evaluation algorithm is described below.

Step1: Building the Initial Trace Graph

We first define the *trace graph*, which is made of the following two primitives.

- A *node*, which represents a production rule firing and
- A *directed edge*, which represents the firing order of two production rules.

The *initial trace graph* can be easily constructed from sample execution traces, by creating nodes and connecting them with directed edges in the original execution order [10]. Fig . 5.1(1) represents an example of initial trace graph.



(1) Initial Trace Graph (2) Transformed Trace Graph

Fig. 5.1 Trace Graph

Step2: Transforming the Trace Graph

Directed edges represent the firing order of production rules. If two firings can be done simultaneously, we can remove the edge between these firings. Conditions and operations for removing edges are as follows.

- If (1) not syn(A<->B) and (2) not com(A->B) then
  - (1) delete an edge which goes from A to B, and
  - (2) add edges which go from A's predecessors to B, and
  - (3) add edges which go from A to B's successors.

Condition (1) indicates the parallel executability of two rules. However, parallel executability does not directly imply that successive two firings actually occur simultaneously. This is the case because if there exists communication between successive two firings, it should be considered that the latter firing is the result of the former firing. Thus condition (2) is necessary. Operation (2) and (3) preserve the order for two firings in connection with other firings.

The *transformed trace graph*, which represents parallel executability, is obtained by applying the above operation to all edges. Fig 5.1(2) shows the result of this transformation.

Step3: Simulation

The final step of the evaluation is a simulation of simultaneous firing on a given partition. The simulation algorithm is as follows.

- (1) Set C (which represents the global communication cost) to 0 and set P (which represents the reduced production cycles) to the number of original production cycles. Calculate the cost of one WM element communication for every split WM class.
- (2) List the nodes which have no predecessors in the trace graph. If the list is empty, then simulation terminates.
- (3) Classify the listed nodes into the following three groups.
  - Group R: Nodes in this group should be executed in the right subtree.
  - Group L: Nodes in this group should be executed in the left subtree.
  - Group OTHER: Nodes in this group are not objects of current decomposition, i.e. these nodes are executed outside of the current tree.
- (4) If group OTHER is not empty, delete all nodes in group OTHER from the transformed trace graph and go back to (2).
- (5) Choose one rule from group R according to some predefined criterion and delete it from the transformed trace graph. If the rule changes some split WM classes, then count the global communication cost and add to C.

- (6) Choose one rule from group L and do same as (5).
- (7) Decrement P and go back to (1).

Total gain G is obtained from P and C. By applying the evaluation algorithm to all partitioned candidates, we can obtain the best partition. However, this approach is possible only if there exist only a small number of partitioned candidates.

5.3 Practical Decomposition Algorithm

The practical decomposition algorithm reduces the computational complexity by the following strategies.

- First, by approximating the total gain of the decomposition by summing the gains obtained from decomposing every rule pair.
- Second, by not considering every possible partition of the rule set.

The practical decomposition algorithm is described below.

Step1: Calculate Gain of Decomposing Rule Pair

Reduced production cycles (represented by $P^*(A,B)$) and global communication cost (represented by $C^*(A,B)$), which are caused by decomposing two rules (A and B), are calculated without considering other rules by using sample execution traces.

The total gain of decomposing rule A and rule B (represented by $G^*(A, B)$) is expressed by the following equation.

$$G^*(A,B) = c_1 P^*(A, B) - c_2 C^*(A,B)$$

Step2: Allocating Rules One by One

To obtain a nearly optimal partition in an incremental manner, the most influential rule pair should be first allocated. The allocating algorithm proceeds as follows.

- (1) Make a list of all rule pairs and sort it in decreasing order of $|G^*(I,J)|$.
- (2) Repeat the following steps until all rules are allocated.
    - (2-1) Pop the first rule pair (I,J) from the list, and allocate it as follows.
        - If $G^*(I,J) >= 0$, allocate I and J to different subtrees.

- If $G^*(I,J) < 0$, allocate I and J to the same subtree.

(2-2) Scan the rule pair list from first to last, and do the following for each rule pair (I,J). If all pairs have been examined, go back to (2-1).

- If both of I and J have been allocated, remove the pair from the list.
- If both of I and J have not been allocated, simply go on to the next pair.
- If one of I and J has been allocated, do as follows. Then remove the pair from the list and go back to (2-2).
  - If $G^*(I,J) >= 0$, allocate I and J to different subtrees.
  - If $G^*(I,J) < 0$, allocate I and J to the same subtree.

6 Conclusion

The main results of this research are as follows.

- We show how production rules are decomposed and allocated on a tree structured machine by use of a hierarchical decomposition algorithm. This algorithm provides a solution to the binary tree bottleneck problem.
- We clarify the mechanisms which are necessary to realize simultaneous firings of production rules, i.e. the global communication and synchronization mechanisms. We also show these mechanism are effective for both fully parallel execution and pipeline execution.
- We propose a practical decomposition algorithm of production rules. This algorithm calculates both the merits and demerits of decomposition, and produces a nearly optimal solution. The algorithm is also applicable to the decomposition of large scale expert systems. In this case, one node of a data dependency graph or trace graph represents not only one rule but a rule set.

This research has been conducted as part of the research of the DADO tree structured machine. The next step is the implementation and evaluation of a simultaneous firing mechanism in the actual DADO environment.

Acknowledgments

We would like to thank Dan Miranker and Mark Lerner for their comments on an earlier version of this paper.

References

1. Stolfo, S. J. and Shaw, D. E., "DADO: A Tree Structured Machine Architecture for Production Systems", In Proceedings of the National Conference of Artificial Intelligence, 1982.

2. Miranker, D. P., "Performance Estimates for the DADO Machine", Technical Report, Department of Computer Science, Columbia University, 1984 (in preparation).

3. Stolfo, S. J., "The DADO Parallel Computer", Technical Report, Department of Computer Science, Columbia University, 1983.

4. Gupta, A., "Implementing OPS5 Production System on DADO", Technical Report, Carnegie Mellon University, 1983.

5. Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence, Sep. 1982.

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

7. Tsuchiya, M. and Gonzalez, M. J., "Toward Optimization of Horizontal Microprograms", IEEE Trans. Comput., vol. C-25, pp.992-999, Oct. 1976.

8. Isoda, S., Kobayashi, Y. and Ishida, T., "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph", IEEE Trans. Comput., vol. C-32, pp.922-933, Oct. 1983.

9. Charniac, E., Riesbeck, C. K. and McDermott, D. V., "Artificial Intelligence Programming", Laurence Erlbaum Associates, pp. 193-226, 1980.

10. Stolfo, S. J., "Automatic Discovery of Heuristics for Nondeterministic Programs from Sample Execution Traces", Ph.D. Th., New York University, 1979.