Data-Driven Parallel Production Systems

JEAN-LUC GAUDIOT, MEMBER, IEEE, AND ANDREW SOHN, STUDENT MEMBER, IEEE

Abstract—Much effort has been expended on developing special architectures dedicated to the efficient execution of production systems. While data-flow principles of execution offer the promise of high programmability for numerical computations, we demonstrate here that the data-driven principles can also be applied to symbolic computations. In particular, we consider a mapping of the RETE match algorithm along the line of production systems. Bottlenecks of the RETE match algorithm in multiprocessor environment are identified, based on which the algorithm is parallelized. The modifications to the actor set as well as the program graph design are shown for execution on the Tagged-Token Data-flow Computer. The results of a deterministic simulation of this multiprocessor architecture demonstrate that artificial intelligence production systems can be efficiently mapped on data-driven architectures.

Index Terms—Data-flow principles of execution, expert systems, parallel processing, pattern matching, production systems.

I. INTRODUCTION

A MAJOR obstacle in the processing of artificial intelligence applications lies in the large search/match time compared to other processing time (such as decision making time, operations on data structure, etc.). In rule-based production systems, for example, it is often the case that the rules and the database needed to represent a particular production system in a certain problem domain would be very large (e.g., on the order of hundreds to thousands of rules and assertions). It is thus known that simply applying software techniques to the matching process would yield intolerable delays. Indeed, the time taken to match patterns over a set of rules can reach 90% of the total computation time spent in expert systems [9]. The need for faster execution of production systems has spurred research in both the software and hardware domains.

The conventional control flow model of execution is limited by the "von Neumann bottleneck" [5]. Architectures based on this model cannot easily deliver large amounts of parallelism [2]. The data driven model of execution has therefore been proposed as a solution to these problems. These principles have been surveyed by [16]. The purpose of this paper is to demonstrate the applicability of data-flow principles of execution and of architecture design to the solution of artificial intelligence (AI) oriented problems. For this purpose, a subset of production systems problems, the RETE match algorithm has been chosen.

Section II contains a brief introduction to production systems, a presentation of the RETE algorithm, of the data-flow principles of execution as well as of the architectural principles of the MIT Tagged Token Data-flow Architecture. Section III discusses the suitability of data-flow interpreters to the implementation of the RETE algorithm and mapping the algorithm to dataflow architectures. There, we identify the problems associated with the RETE algorithm in a multiprocessor environment and give solutions to these problems through the allocation and distribution policies we have developed. A specific example which uses our strategies is worked out in Section IV. In Section V, the program graph design techniques of the RETE algorithm in a data-flow environment is described and simulations are carried out. Performance observations obtained for a data-driven environment are compared to those of a conventional control-flow approach. Concluding remarks as well as future research topics are discussed in Section VI.

II. BACKGROUND

Necessary backgrounds on production systems, the RETE algorithm, and the data-flow principles of execution are briefly discussed in this section.

A. Production Systems

There are many ways of solving problems and representing knowledge in AI applications. Among them, production systems have recently gained much attention. This is partly due to the fact that these techniques have been used to implement well known rule-based expert systems: R1 [25], MYCIN [8], PROSPECTOR [13], etc. Also, production systems render simpler the representation of knowledge and ease the implementation of control mechanisms.

A production system (PS) paradigm consists of three modules: production memory (PM), working memory (WM), and inference engine (IE). PM (or rulebase) is composed entirely of conditional statements called productions (or rules). These productions are similar to if-then statements in conventional programming languages in that some predefined actions are performed if all the necessary conditions are satisfied. The left-hand side (LHS) is the condition part of a production rule, while the right-hand side (RHS) is the action part. Both LHS and RHS consist of one to many elements, called patterns.

The productions operate on WM which is a database of
assertions called working memory elements (WME's). Both patterns and WME's have a list of elements, called attribute-value pairs (AVP's). The value to an attribute can be either constant or variable; the former is in the lower case while the latter in the upper case (which are similar to the Prolog representation). Consider the following simple production system:

Example 1:

<table>
<thead>
<tr>
<th>Production Memory</th>
<th>Working Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule1: ((c X) (d Y))</td>
<td>Condition pattern 1 WME1: ([p 1] (q 2) (r +))</td>
</tr>
<tr>
<td>([c X] (d Y))</td>
<td>Condition pattern 2 WME2: ([r =] (d +))</td>
</tr>
<tr>
<td>([p 1] (q 2) (r X))</td>
<td>Condition pattern 3 WME3: ([r =] (d +))</td>
</tr>
<tr>
<td>([d X] (d Y))</td>
<td>Action pattern 1 WME4: ([b 3])</td>
</tr>
<tr>
<td>([Add (c 1) (d Y)])</td>
<td>Action pattern 1 WME5: ([b +])</td>
</tr>
<tr>
<td>([Add (c 1) (d Y)])</td>
<td>Action pattern 1 WME6: ([p 1] (q 3) (r 7))</td>
</tr>
</tbody>
</table>

The rule above will perform the action with the corresponding instantiations when all the condition patterns are verified in the working memory. A typical execution cycle of production systems is composed basically of three steps: pattern matching, conflict resolution, followed by rule firing:

• Pattern Matching: The LHS's of all the production rules are matched against the current WME's to determine the set of satisfied productions. WME1 in the Example 1 above can satisfy the condition pattern3 with the variable \(X\) instantiated to * whereas WME6 cannot. This step will eventually identify three WME's, WME1, WME3, and WME5 which all satisfy the above rule with \(X\) and \(Y\) instantiated respectively to * and +.

• Conflict Resolution: If the set of satisfied productions is nonempty, one rule is selected for execution in the next step. Otherwise, the execution cycle halts because there are no satisfied productions. In Example 1, only one rule is satisfied. It is therefore selected.

• Rule Firing: The actions specified in the RHS of the selected productions are performed. In the case of Example 1, a new WME, \([c 1] (d +)\), is added to the working memory and WME4, \([b +]\), is deleted from the working memory upon rule firing.

The inference engine will halt the production system either when there are no satisfied productions or when the desired solution is found. In this paper, we limit ourselves to the matching step only since, as pointed out earlier in the introduction, it is that takes most of the computation time in the evaluation of production systems.

B. The RETE Match Algorithm

The RETE match algorithm [9] is a highly efficient approach used in the matching of objects in production systems. The simplest possible matching algorithm would consist in going through all the rules and WME's one by one until one (or several) match(es) has (have) been found. The RETE algorithm, however, does not iterate over the WME's to match all the rules. Instead, it constructs a condition dependency network, saves in memory the information concerning the changes in the working memory between production cycles, and then utilizes them at a later time. This is based on the observation, called temporal redundancy [7], that there is little change in the working memory between production cycles. The RETE algorithm further reduces the matching time by sharing identical tests among productions. It stems from the fact that the productions have many similar or identical parts, called structural similarity. This second improvement, however, is strongly affected by the problem domain and by the processing mechanism used.

Constructing the Condition-Dependency Network: Given a set of rules, a network similar to the one shown in Fig. 1 is built which contains information extracted from the LHSs of the rules. The network consists of several types of nodes:

- Root Node (RN) distributes incoming data-tokens (or WME's) to sequences of children nodes, called one-input nodes. Note that we shall interchangeably use data tokens and WME's throughout the paper.

- One-Input Nodes (OIN) test intraelement features contained in a condition pattern, i.e., compare the value of the incoming WME's to some preset value in the pattern. For example, the first condition pattern of Example 1 contains 3 intraelement features and therefore 3 OIN's are needed to test them. The test result of the one-input nodes are propagated to nodes, called two-input nodes.

- Two-Input Nodes (TIN) are designed to test intercondition features contained in two or more condition patterns. The variable \(X\), which appeared in both condition pattern1 and condition pattern3 of Example 1, must be instantiated to the same value for possible rule instantiation. Attached to the two-input nodes are left and right memories in which WME's matched through one-input nodes are saved. The result from two-input nodes, when successful, are passed to nodes, called terminal nodes.

- Negated Two-Input Nodes (NTIN) operate in the same fashion as regular two-input nodes except that they are designed to process patterns preceded by -, which means not. NTIN tests to determine whether no WME satisfies it. This kind of nodes requires special attention. One way of implementing NTIN's is by the use of counters associated with WME's. Whenever there is a match (instantiation) in this type of two-input nodes, a counter in the counterpart memory is incremented by one instead of joining and passing the matched tokens. The counter will be decremented when the matched token is removed from the memory by the action of the other production. As the counter reaches zero, the rule is said to be instantiated and is put into conflict set to resolve the conflict, if any. We will discuss this in detail in Section IV.

- Terminal Nodes (TN): Each terminal node represents a rule and triggers it when all the preceding nodes have done their tests over the incoming WME's. A predefined conflict resolution strategy is then invoked to select and fire a rule.

Processing in the Network: A condition-dependency network for the rule of Example 1 has been constructed in Fig. 1. Each of the three branches emanating from the
root node corresponds to each of the three condition patterns. Assume that the set of WME’s in Example 1 is received at the root node at time $t_0$. The RETE algorithm will proceed as follows:

- **At time $t_1$**, WME1 \( [(p \ 1) \ (q \ 2) \ (r \ *)] \) is distributed to 3 nodes; OIN1, OIN4, and OIN6 by the root node. Three nodes compare the first AVP \( (p \ 1) \) of WME1. OIN1 is successful in matching while the other two fail. Both OIN4 and OIN6 thus produce no resulting tokens. On the other hand, since it has been successful, OIN1 passes the WME1 token to OIN2, where the test succeeds again. When WME1 gets to OIN3, the test is again successful because it has an attribute \( r \). Now the variable \( X \) is bound to a value \( +* \). WME1 is then passed to TIN2 and stored in RM2. At the same time, TIN2 initiates the comparison tests against WME’s in LM2, if any. Since we have assumed that WME1 was the first incoming token, no further processing takes place.

- **At $t_2$**, WME2 \( [(r \ =) \ (d \ +)] \) is distributed to 3 nodes. None of the three nodes succeeds. WME2 is immediately rejected and no further processing takes place.

- **At $t_3$**, WME3 \( [(c \ *) \ (d \ +)] \) is distributed and only OIN4 succeeds since the incoming token has an attribute \( c \) in it. OIN4 binds a variable \( X \) to the value \( +* \) and passes WME3 to OIN5, where the test is successful because WME3 carries an attribute \( d \). So OIN5 binds a variable \( Y \) to the value \( +* \), and then passes WME3 to TIN1. As soon as WME6 reaches TIN1, it is stored in LM1 and an interelement feature test is initiated to check whether any WME has arrived at RM1. No WME is found in RM1, and therefore the processing stops.

- **At $t_4$**, WME4 \( [(b \ 3)] \) is distributed to the network, where all others but OIN6 fail. OIN6 binds the variable \( Y \) to 3 and passes WME4 to TIN1 to store in RM1. TIN1 then begins executing the interelement feature test against WME3 of LM1. However, the values bound to \( Y \) are not the same; the one in LM1 was bound to \( +* \), the other inRM1 to 3. The test thus fails.

- **At $t_5$**, WME5 \( [(b \ +)] \) is distributed to the network and OIN6 succeeds. Upon binding the variable \( Y \) to \( +* \), OIN6 passes WME5 to TIN1. WME5 is now stored in RM1. TIN1 executes the interelement feature test against WME3 of LM1 and is successful since both \( Y \) variables are bound to \( +* \). WME3 and WME5 are now combined into a single token WME3, 5, which is in turn sent to TIN2 to be stored in LM2.

- **At $t_6$**, the interelement feature test is initiated to check whether any WME(s) have arrived at RM2. WME1, which was passed to RM2 at $t_1$, is found in it. The two values to which the variable \( X \) is bound are found the same, i.e., \( +* \). TIN2 will notify a terminal node that the rule is satisfied with \( X \) instantiated to \( +* \) and \( Y \) to \( +* \). A conflict resolution step will immediately proceed to find which rule to fire.

### C. Data-Flow Principles of Execution

Programmability is a major issue in the design of large scale multiprocessor systems. Programmers cannot be expected to be able to schedule and synchronize the hundreds or thousands of tasks that are required to fully utilize the resources of such a machine. The dataflow model of computation was introduced to alleviate this problem. Dataflow principles of execution offer the runtime synchronization of operations based on their data dependencies. This allows a very large number of different tasks to be efficiently and safely allocated to the entire machine.

Data-flow computing is an alternative to the control flow model of execution. It is inherently parallel and the sequencing of an instruction is based upon the availability of its arguments. Data-flow principles can be characterized by the following two statements:

- Operations execute only when all required operands are available.

- Actors which perform predefined functions are purely functional and execution produces no side-effects.

Data-flow programs are represented by directed, acyclic graphs which consist of actors connected together with arcs. Arcs represent the data dependencies between actors and carry tokens which are the data values being passed between actors. The architecture model of the Tagged-Token Data-flow Computer [1], [3] was adopted as the machine model of the simulator. It contains 64 Processing Elements (PE’s) interconnected by a packet switching 6-dimension hypercube network. The structure of each PE which consists of six units is depicted in Fig. 2.

In what follows, the functionality of each unit in the PE is briefly described:

- The Matching/Store Unit (MSU) in which incoming tokens are associatively compared with previously arrived...
tokens. Matched tokens are then sent, along with the opcode to the next unit.

- Instruction Fetch Unit (IFU), when all the data tokens are received from MSU, retrieves a corresponding instruction stored in the program memory and sends it to ALU.
- The Arithmetic/Logic Unit (ALU) receives the ready instruction packets and processes them according to the opcode of the template.
- The Token Formatting Unit (TFU) receives results from ALU and forwards them to MSU or to another PE through the message passing network.
- The I-Structure Controller (ISC) handles array access operations. Such actors as SELECT or APPEND are processed by this unit.
- Program Memory (P-Mem) stores instructions in the form of templates.

In addition to the above simulator specifications, several assumptions are made, which are considered reasonable in this data-flow processor. Each PE has several simple functional units which can enable the parallel matching of attribute-value pairs. The number of simple functional units in the PE would range from 1 to 10 due to the fact that there are no more than 5 attribute-value pairs in any condition of the left hand side of the rules.

III. DATA-FLOW IMPLEMENTATION OF THE RETE ALGORITHM

Based on the background information discussed in the previous section, we present the suitability of mapping production systems on data-flow processors. The necessary mapping schemes to suit the RETE match algorithm and the data-flow multiprocessor are identified in this section. Bottlenecks in the RETE algorithm are identified and possible solutions are suggested.

A. Suitability of the Data-Driven Execution Model

The applications of data-flow computers studied thus far fall basically into the area of numerical computations such as signal processing [16], partial differential equation solvers [18], matrix manipulation, etc. Indeed, data-flow execution is generally thought to be more applicable to numerical applications rather than symbolic processing because:

- The data structures used in symbolic computations are irregular and undeterministic compared to the fairly regular and predictable data structures created and used in numerical computations.
- The basic entity used in numerical computations is a number (either floating point or fixed point) while in symbolic computations, it is an object or a set of objects. It requires good modeling techniques to represent the structure of objects into numerical values.

For the foregoing reasons, the following are identified as necessary modifications to a data-flow multiprocessor in order to accommodate symbolic computations:

- Due to the larger size of the data elements, data tokens must be allowed to carry more information than the single scalar element allowed in the basic Tagged Token Dataflow Architecture.
- Fewer primitive functions are needed in symbolic computations than are required for numerical computations, where complex functions are often executed. In order to effectively utilize this advantage, the ALU needs major modifications. By adding several simple functional units to each PE, throughput will substantially increase.

As we have demonstrated in the previous section, data-flow principles of execution and the RETE match algorithm present a natural match both at the level of the implementation and at the level of execution principles. Indeed, executing the RETE algorithm on a data-flow multiprocessor has many advantages over execution on a conventional control-flow computer:

- The execution principles of the RETE algorithm are driven by incoming data tokens, i.e., execution may proceed whenever data are available. In any situation, multiple firings of actor in data-flow and comparison tests in the RETE algorithm are possible unless PE’s are busy.
- Both are based on the single assignment principle, i.e., no data modifications except arrays.
- Both a data-flow machine and the RETE algorithm need dependency graphs which are obtained from the problem domain.
- The requirement for the memorization capability in two-input nodes of the RETE algorithm assumes a good structure handling technique. This can be effected by using the I-Structure Controller in the dynamic data-flow machine.
- The dynamic data-flow architecture allows an easy manipulation of the counters attached to the WME’s. The counter for negated-pattern processing can be treated the same as other tags in the dynamic architectures.

B. The RETE Algorithm in a Multiprocessor Environment

Mapping production systems onto multiprocessor systems has been investigated in several ways in the recent literature. Direct mapping employed by [37] for DADO uses “full distribution,” which allocates a production to an available PE. In this approach, the production-level
parallelism can be easily achieved by having several PE's operate simultaneously on WME's to match productions. In [20] a relevancy between the rules and the WME's is identified and used to directly allocate rules to PE's. The relevancy is defined as "a working memory element is relevant to a production if it matches at least one of its condition elements."

It has been suggested by [6] that the semantic network can be directly viewed as a data-flow graph. Each node in the semantic network corresponds to an active element capable of accepting, processing, and emitting value tokens traveling asynchronously along the arcs. The other approach suggested by [40] may be considered an indirect mapping. In this approach, all productions are analyzed and grouped according to the dependency existing between productions to enable parallel firing of rules. The work reported in Parallel Inference Machine (PIM) of the Fifth Generation Computer System which in essence attempts to parallelize the production system paradigm does not involve the data-flow principles of execution [12]. We will not consider the PIM project further in this paper.

The mapping scheme adopted for our simulation, however, is different from the aforementioned approaches. The motivation for the choice of an alternative method is in two facts. First, the architecture we have adopted is based on data-flow principles of execution. Since the parallel model employed in this paper exploits parallelism at the production level, condition level, and further subcondition level (attribute-value pair level), the mapping scheme must be efficient to utilize all the possible forms of parallelism inherent to both data-flow principles and the RETE algorithm.

Second, the RETE algorithm presents two bottlenecks which substantially degrade the performance of the production system in our parallel machine:

- Since the root node distributes tokens one at a time to all PE's, tokens will pile up on the input arcs as shown in Fig. 3. This is due to the fact that rules cannot be copied to all PE's. Indeed, it is recognized that even a medium sized expert system can include from several hundred to thousand rules. Each rule contains at least several conditions. Replicating such a database would prove prohibitively expensive. Also, each condition can contain information matched in an earlier production system cycle. This information is dynamic and cannot be effectively replicated unless all PE's perform the same operation at the same time. Performance would then be reduced to that of a single processor system!

- The second inefficiency can also be seen in Fig. 3. Assume that \( m \) tokens are received and matched on the left input arc of the two-input node. Further assume that a token is received and matched on the other input of the two-input node. The arrival of this last token will trigger the invocation of \( m \) comparisons with the values received and stored in the left memory LM1 of the two-input node. On the average, there will be \( O(m) \) such tests. Should the situation have been reversed and \( n \) tokens be in the right memory RM1, a token on the left side would provoke \( O(n) \) comparisons. The internal workings of this two-input node are therefore purely sequential. In order to avoid the wasted time in searching through the entire memory, an effective allocation of two-input nodes and one-input nodes should be devised.

It has been reported that there is another major bottleneck in the RETE algorithm [26]. It is based on the observation that the memory management for two-input nodes takes substantial amounts of time when deleting WME's. To overcome this problem, the TREAT algorithm has been introduced in order to support the conflict set. However, at this time, we will limit ourselves to the two bottlenecks we identified above and will not consider this further improvement in our simulation and performance evaluation. We now propose a production allocation policy and a WME distribution policy which will overcome these bottlenecks.

C. Allocation of Productions

Productions are partitioned into LHS's and RHS's. LHS's are further partitioned into patterns. All patterns are logically grouped together according to the number of attribute-value pairs (AVP's) in the patterns. There are two ways of allocating productions onto the PE's: redundant and minimum allocations. The first one, redundant allocation, does not follow the structural similarity discussed in Section II-B. There is no sharing of productions in this strategy. All patterns are copied and independently allocated. The major advantage to using this strategy stems from the fact that there is less communication overhead between PE's. However, this will consume a lot of processor space and be costly as the number of productions that share patterns or part of patterns increases.

The second policy, minimum allocation, follows the structural similarity. The major advantage behind adopting this concept is in the fact that reducing the computation time in the matching step can also be achieved by keeping all the PE's busy. At the same time storage usage
can be substantially reduced. However, this will increase overhead in interprocessor communication.

In this paper, we will apply the redundant allocation strategy for production allocation. Allocating one-input nodes is quite straightforward. A condition that has AVP's (or OIN's) is allocated to PE[i, j], where j ≥ 0. There are however many factors affecting the allocations of two-input nodes. The I-structure controller is used to solve the second bottleneck issue since two-input nodes require a structure handling capability due to the saving of information about changes in the working memory. A two-input node is split into two memories: left and right memories. A memory MEM[i, j] is allocated to PE[i, j] where the corresponding one-input nodes are allocated. Allocating a memory to a PE will ensure an even distribution of processing load across the processor space. At the same time, we can realize parallel matching in condition level. In what follows, we informally describe our allocation algorithm:

Procedure Allocate_Patterns_to_PE's

1. NRULE ← A number of rules in the system;
2. For i = 1 to NRULE do
3.  NCOND ← A number of conditions in the RULE[i];
4.  For j = 1 to NCOND do
5.    NAVP[i, j] ← A number of AVP's in CONDITION[j] of RULE[i];
6.    n ← USED[NAV[i, j]]; A number of PE's used in GROUP[i];
7.    For k = 1 to NAVP[i, j] do
8.      PE[NAV[i, j, n] ← OIN[i, j, k]; one-input node allocation
9.      PE[NAV[i, j, n] ← MEM[i, j]; memory allocation
10. USED[NAV[i, j]] ← n + 1;

Terminal nodes are not explicitly allocated to PE's for our simulation. Instead, we make the last cycle of a two-input node notify a matching status. If a last two-input node for a certain rule says matched, then the rule is said to be instantiated. Fig. 4 depicts the condition dependency network for the rules of Example 2 shown below. Note that Rules 1 and 3 in Example 2 are designed to demonstrate the performance of the negated two-input node. Ellipses in Fig. 4 correspond to one-input nodes. Two input nodes are represented by boxes whereas negated two-input nodes are represented by double boxes. Numbers labeled on nodes are used to indicate where nodes are allocated in the processor space.

Example 2:

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Example 2 Diagram" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Based on the above allocation policy, the network is allocated to PE's, shown in Fig. 5. PE's are partitioned into 5 different groups, where PE's in GROUP a contain patterns having n AVP's. GROUP1 is not used in our example since no condition pattern has only one AVP. Consider the first pattern of Rule 2, [(p 1) (q 2) (r X)], for example. The sequence of nodes in the pattern and the left memory for that pattern are labeled 11 through 14 in Fig. 4 (11 through 13 are one-input nodes). Since the pattern has 3 AVP's, it is classified into GROUP3 and allocated to PE1 of GROUP3, denoted by PE3.1. The second pattern of Rule 2 in Example 2 has 2 AVP's and right memory, labeled 15 – 17. It is classified into GROUP2 and allocated to PE2 of GROUP2, denoted by PE2.2.

In the above allocation policy, we notice that the number of PE's needed to allocate productions is proportional to the number of interelement feature tests in the productions. For example, suppose that a certain system has n productions and that there are on the average m interelement feature tests per production. For each interelement feature test, two memories are needed. The number of PE's needed to allocate n productions would then be 2mn. For the three rules shown in Example 2, there are 3 rules and on the average 2 interelement feature tests. In total, 12 PE's are used, as depicted in Fig. 5.

C. Dynamic Working Memory Elements Distribution

Although the RETE algorithm is designed to save computation time in matching patterns over WME's there is a bottleneck at the root node as discussed at the beginning of this section. In order to overcome this barrier, we propose one scheme which simultaneously distributes many different tokens to many PE's at a time, provided that many WME's are available at the same time for distribution. It is based on the fact that certain WME's eventually fall into PE's in a certain group, where they may be
matched. WME's that have i AVP's never match patterns that have j AVP's such that i < j.

Whenever the new WME's that are generated due to the rule firings become ready for distribution to the network, PE's perform the following operations:

Procedure Distribute WME[\(n\)] to PE's
1) NAVP[\(n\)] \(\leftarrow\) a number of AVP's in WME[\(n\)]
2) Attach NAVP[\(n\)] tag to WME[\(n\)]
3) Route WME[\(n\)] to PE[\(i, j\)] for all \(j\) such that \(i = \text{NAV}[n]\).

Assume that the three rules in Example 2 are compiled and allocated to the PE's according to the allocation policy described in Section IV-B. Suppose further that a set of WME's shown below is available and is about to be distributed into the network in Fig. 4 at a certain time \(t\).

We will show the efficiency of our distribution policy as follows:

**Working Memory:**

WME1: \[[p \ 1] \ (q \ 2) \ (r \ *)\]
WME2: \[[p \ 1] \ (q \ 2) \ (r \ =)\]
WME3: \[[p \ 1] \ (q \ *) \ (r \ 3)\]
WME4: \[[l \ 5] \ (m \ 6) \ (n \ +) \ (o \ *)\]
WME5: \[[l \ 5] \ (m \ 6) \ (n \ 2) \ (o \ =)\]
WME6: \[[a \ +] \ (c \ 5) \ (e \ 7) \ (j \ 0)\]
WME7: \[[c \ +] \ (d \ 6)]\]
WME8: \[[c \ 3] \ (d \ +)\]
WME9: \[[c \ 3] \ (d \ =)\]
WME10: \[[a \ +] \ (b \ 6)]\]
WME11: \[[a \ 2] \ (b \ =)\]
WME12: \[[a \ 2] \ (b \ =)\]

If the RETE algorithm distributes one WME at a time to the network through the root node, it would take 12 \(t\)'s (or time units) to distribute them. Furthermore, a number of comparison tests which are performed at the very first one-input nodes (1, 4, 9, 11, 15, and 19) will reach 108 (= \(9\times12\)) WME's. This is depicted in Fig. 6, where one WME at a time is sequentially distributed to all PE's.

For example, when WME1 is distributed, all 9 PE's to which patterns are allocated make a comparison test simultaneously. Only two PE's, PE3.0 and PE3.1, will succeed in matching. This forces the machine to operate in Single-Instruction-stream-Multiple-Data-stream (SIMD) execution mode although it has a Multiple-Instruction-stream-Multiple-Data-stream (MIMD) processing capability. Applying our distribution policy, the 12 WME's are partitioned into 3 groups and the group numbers are assigned to WME's. WME's 7-12 get group #2 while 1-3 get #3, and 4-6 get #4. The total number of comparison tests performed at the very first one-input nodes in three sequences reduces to 36 (= \(6 \times 4 + 3 \times 2 + 3 \times 2\)), as shown in Fig. 7. There are three bins in Fig. 7, where each bin corresponds to a certain group. In each group, WME's are sequentially distributed to PE's belonging to the corresponding group in the PE space. However, between groups WME's are simultaneously distributed.

If we defined the speed-up for the distribution policy \(S = N_s / N_p\), where \(N_s\) and \(N_p\) are, respectively, numbers of comparisons to be performed for sequential distribution and parallel distribution, we will obtain \(S = 108 / 36 = 3\) for the given set of WME's. The number of groups in working memory determines the speed-up \(S\). In the worst case, only one WME can be distributed to all PE's at a time as shown in Fig. 6. Note that in the original RETE algorithm, a sequential distribution, analogous to our worse case, would be implemented. Instead, our improvement provides the extra parallelism although this scheme depends heavily on the fact that WME's will be evenly classified to all groups.

**IV. Example**

In this section we will show the execution mechanism of the RETE match algorithm in a data-flow multiprocessor system. Based on our execution mechanism, we will then partly approximate the execution time. In the next section, this approximation will be compared to simulation observation.

**A. Execution Sequence**

We use the three rules in Example 2 and the 12 WME's shown above, as well as the corresponding network shown in Fig. 4. The allocation and distribution policies discussed in the previous section are used to show an implementation of the RETE match algorithm in data-flow multiprocessors.

In the following execution sequence we use three time
units. They are defined as follows. A unit time \( t \) is the time taken for a token to pass through any physical unit in the PE. A loop is the time taken by a WME to go through a PE and come back to the input switch. Again, we interchangeably use tokens and WME’s throughout the paper. \( T \) denotes an abstract time to demonstrate parallel matching performed in condition-level and distinguish various events occurred in a high level execution sequence. Assuming WME1, WME4, and WME7 are simultaneously injected into the network at time \( T_0 \), the following steps will take place:

1) At time \( T_0 \), WME1 \([\{p\ 1\} \ (q\ 2) \ (r\ *)\] comes into the network as a data token and is distributed to 2 PE’s in GROUP 3 since it has 3 Attribute-Value Pairs. Let us consider an execution sequence in PE3,1.

a) Loop 0 in the PE3,1 for intraelement feature test:

i) At time \( t_0 \), the switch in PE0 forwards the token to the Matching/Store Unit (MSU) where the token is identified as a monadic comparison actor. Indeed, the other term of the comparison is “built-in” the comparison actor, therefore no matching is necessary.

ii) At time \( t_1 \), the token can be sent to the Instruction Fetch Unit (IFU), where a built-in operand (two one-input nodes and one binding node labeled 11, 12, and 13, depicted in Fig. 8) and opcode (comparison function) are fetched from a program memory (PM) and sent to the ALU.

iii) At time \( t_2 \), receiving two operands and opcode by the ALU, five comparison operations are simultaneously performed on five pairs, i.e., on \( 'p' \ and \ 'p' \, ' 1' and ' 1' \, 'q' and 'q' \, ' 2' and ' 2' \, and 'r' and 'r' \) in five functional units. As pointed out in Section II-C, we assume that each ALU has several simple functional units to support a parallel execution of subcondition level. Note that a variable \( X \) is automatically bound to * when the comparisons are successful.

iv) At time \( t_3 \), after two one-input nodes are successfully compared in ALU of PE3, 1 the data token \([\{p\ 1\} \ (q\ 2) \ (r\ *)\] is sent to the Token Formatting Unit (TFU), where the necessary tagging operation is done (since the architecture model adopted is a dynamic data-flow architecture). The output module (not shown in Fig. 2) routes the token back to PE3,1 for two-input node operations as it receives it from TFU.

b) Loop 1 in the PE3,1 for array operation:

i) At \( t_4 \), the switch in PE3,1 sends the token to MSU.

ii) At \( t_5 \), the IFU fetches an append opcode.

iii) At \( t_6 \), the WME1 is sent back to the switch in PE3,1.

c) Loop 2 in the PE3,1 for saving WME1.

i) At \( t_7 \), the switch in PE3,1 sends the token to MSU.

ii) At \( t_8 \), the I-Structure Controller (ISC) copies LM14 and appends WME1 to it (shown in Fig. 8).

d) Loop 3–4 in the PE3,1 for interelement feature test: PE3,1 checks the MSU to see if any WME arrived from PE2,2, in which RM17 is allocated. Assuming that step 1 finishes before step 2, there is no WME arrived at MSU of PE3,1. It sends out WME1 to PE2,2.

2) At \( T_0 \), WME7 \([\{c\ *\} \ (d\ 6)\] is distributed to 4 PE’s in GROUP2 since it has 2 AVP’s.

a) Loop 0–2 in PE2,2 for intraelement feature test and saving WME7 in RM17 (shown in Fig. 8).

b) Loop 3–5 in PE2,2 for interelement feature test about X: PE2,2 checks the MSU to see if any WME arrived from PE3,1. As assumed in step 1-d the matching operation on WME1 is performed before step 2, so WME1 has been stored in LM14 and sent to MSU of PE2,2. To check the consistency in variable instantiations, the values of attribute \( r \) in WME1 of LM14 and \( c \) in WME7 of RM17 are compared and found equal. Two
3) At $T_0$, $WME_4 \{(15)(m \, n \, (o \, n))\}$ is distributed to 2 PE's in GROUP4 since it has 4 AVP's.
   a) Loop 0–2 in PE4,0 for intraelement feature test and saving $WME_4$ in RM23 (shown in Fig. 8).
   b) Loop 3 and 4 in PE4,0 for interelement feature test: PE4,0 checks its MSU but no WME has arrived from LM18 of PE0,1 since the step 4 is not yet completed. It therefore routes $WME_4$ to PE0,1.

4) At $T_1$, $WME_1,7$ is received by PE0,1.
   a) Loop 0 and 1 in PE0,1 for saving $WME_1,7$ in LM18 (shown in Fig. 9).
   b) Loop 2–4 in PE0,1 for a second intraelement feature test: PE0,1 checks the MSU and finds $WME_4$, which has been sent from step 3. The values of attribute ‘$d$’ in $WME_1,7$ of LM18 and ‘$n$’ in $WME_4$ of MSU are compared. The test fails due to the inconsistent variable instantiations. The values of W in $WME_4$ and $WME_1,7$ are respectively ‘+’ and ‘6’ and certainly different. It then sends out $WME_1,7$ to PE04,0. See step 7.

5) At $T_1$, $WME_{10} \{(a \, +) \,(b \, 6)\}$ is distributed to 4 PE's in GROUP2.
   a) Loop 0–2 in PE2,0 for intraelement feature test and saving $WME_{10}$ in RM3 (shown in Fig. 9).
   b) Loop 3–5 in PE2,0 for interelement feature test about $W$: PE2,0 checks the MSU and finds $WME_7$, which has been received from step 2. The values of attribute ‘$b$’ in $WME_{10}$ of LM3 and ‘$d’ in $WME_7$ in MSU are compared and found successful. $WME_{10}$ and $WME_7$ are put together to $WME_7,10$, which is sent to LM7 of PE0,0. See step 8 for next sequence.

6) At $T_1$, $WME_3 \{(p \, 1) \,(q \, +) \,(r \, 3)\}$ is distributed to the 2 PE's in GROUP 3. No PE succeeds in matching since built-in operand $[(p \, 1) \,(q \, 2) \,(r \, X)]$ and $WME_3$ are different.

7) At $T_1$, $WME_5 \{(15) \,(m \, 6) \,(n \, 6) \,(o \, 2)\}$ is distributed to 3 PE's in GROUP4.
   a) Loop 0–2 in PE4,0 for intraelement feature test and saving $WME_5$ in RM23.
   b) Loop 3 and 5 in PE4,0 for the second interelement feature test about $W$: PE4,0 checks the MSU and finds $WME_7$, which has been sent from step 4. The values of attribute ‘$d’ in $WME_{1,7}$ of MSU and ‘$n’ in $WME_5$ of RM23 are compared and found equal.
   c) Loop 6 in PE4,0 for rule instantiation: $WME_{1,7}$ and $WME_5$ are put together into $WME_{1,5,7}$, which is sent to terminal node for selection step. At this time, Rule 2 is said to be satisfied with $X$, $W$, and $Z$ instantiated respectively to ‘*’, ‘6’, and ‘2’.

8) At $T_2$, $WME_7,10$ is received by PE0,0.
   a) Loop 0 and 1 in PE0,0 for saving $WME_7,10$ in LM7 (shown in Fig. 10).
   b) Loop 2 and 3 in PE0,0 for a second interelement feature test: It checks the MSU to see if any WME arrived from PE3,0. Assuming step 8 completes before step 9, no WME is found in MSU of PE0,0. $WME_7,10$ is then routed to PE3,0. See step 9 for next sequence.

9) At $T_2$, assume that the conflict is resolved. Rule 2 fires and -$WME_1$ is distributed to PE3,0 and PE3,1 through RN3 since -$WME_1$ has 3 AVP's.
   a) Loop 0 in PE3,0 for intraelement feature test: Nodes 11–13 are executed on -$WME_1$.
   b) Loop 1–3 in PE3,0 for memory examination: Recall that PE3,0 contains a negated pattern. PE3,0 checks if $WME_1$ exists in RM8 and selects $WME_1$ from the RM8.
   c) Loop 4 and 5 in PE3,0 for counter manipulation: Assume that there is only one $WME_1$ in RM8, as is the case, and that the counter attached to $WME_1$ is 1. Here, again the counter on WME is treated in a similar fashion as other tags attached to a data token. The counter tag is decremented by one and...
found zero. Of course, at the same time PE3.1 deletes WME1 from the RM14 at T2 and in turn WME1.7 from LM18 at T3 by the same manner.

d) Loop 6 and 7 in PE3.0 for second interelement feature test about variable X. It checks the MSU and determines if any WME arrived from PE0.0. As we assumed in step 8-b, there is WME7.10 in MSU. The values of attribute c in WME7.10 of MSU and r in -WME1 or RM8 are compared and found equal. Now, Rule 1 is satisfied with X, Y, and Z instantiated respectively to *, 6, and +. Any conflict resolution strategy will proceed.

10) At T2, WME11.1 [(a 2) (b 6)] is distributed to 4 PE’s in GROUP2 and goes through the intraelement feature test in PE2.0. Upon matching, WME11 is stored in LM3. PE2.0 checks its MSU to determine whether any WME arrived from PE2.1. It finds WME7 in it. T0 checks an interelement feature test about Y, the values of attribute b in WME11 of LM3, and d in WME7 of RM6 are compared. The test succeeds. WME7 and WME11 are put together into WME7.11, which is sent to LM7 of PE0.0 (shown in Fig. 10).

11) T2, WME6 [(a *) (c 5) (e 7) (f *)] is distributed to 3 PE’s in GROUP4 and goes through an intraelement feature test in PE4.1. Upon matching, WME6 is stored in RM27. The counter on WME6 is examined and found nonzero. Recall that PE4.1 contains negated pattern. No interelement feature test about W is necessary.

B. Analysis of the Example

The condition-level parallel matching is demonstrated in steps 1, 2, and 3, where the dynamic parallel distribution of WME’s is made. Steps 5, 6, 7, and 8 as well as steps 9, 10, 11, and 12 also shows the condition-level parallel matching. Negated-pattern handling and deleting WME’s are detailed in steps 9 and 11. The advantage of the allocation policy we adopted is apparent in the above example. By allocating memories to different PE’s, all the data-flow processing environment. The Tagged-Token Data-flow Architecture of Arvind has been chosen as a simulation model.

A. Simulation

In this simulation, a set of 12 WME’s and three rules (shown in Example 2) are used. Rule 2 is converted into a data-flow graph. Fig. 11 shows the condition pattern of Rule 2, i.e., nodes 11–14 of Fig. 4. One-input nodes 11–13 of Fig. 4 for intraelement feature tests are implemented through decision functions labeled 0, 2, 3, and 39 in Fig. 11. All others in Fig. 11 are for two-input node 14 of Fig. 4.

One-Input Nodes and Array Operations: Only one PE is used in this set of experiments. First, one-input nodes are tested and then those successful in tests are appended and copied to another array. The results of these simulation runs are displayed in Table II and Fig. 12(a) depicts the results of trial 1.

Note that the memories are allocated to the same PE where the OIN’s are. Table II shows a sequence of one-
input nodes takes about 15 time units, or 15\(r\), using one PE. Each additional matching takes 13\(r\).

**Independent Two-Input Nodes:** Three conditions are tested separately one at a time. The condition pattern 1, \([(p_1)(q_2)(rX)]\), is matched to a set of WME’s with variations in the order [see Fig. 12(b)-(d)]. WME’s 1 and 2 are injected into the left sequence of Rule 2 assuming that the RM17 is filled with WME’s 7, 8, 9, and 12 that have been matched with the middle sequence, \([cXdW]\), of the same rule. Table III summarizes the simulation time. Trial 1, shown in Fig. 12(b), indicates WME1 matches against WME7 of RM17 which results in 76\(r\). Notice in trial 5 that when no match occurs, i.e., when WME2 is placed into the network, the simulation time becomes 286\(r\) due to an exhaustive search in the RM17.

**Parallel Execution of Two-Input Nodes:** In this experiment, two patterns are executed in parallel, as depicted in Fig. 12(c)-(g). It takes about 200-500\(r\) depending upon the number of WME’s that have reached either LM14 or RM17 of the two-input node in Fig. 4. Table IV summarizes the results with various WME’s coming into the network.

The first two columns in Table IV show the WME’s randomly coming into the network without any order and go to either left or middle sequence of Rule 2. X’s in the table represent WME’s that will never match WME’s that come from the other sequence whereas O’s represent WME’s that will match those from the other sequence. For example, the first row with X and X shows that 1 WME is distributed to each pattern, depicted in Fig. 12(e) and there is no match. The 7th row with X O O and X X O shows that there are three WME’s distributed to each pattern and that there are two matches.

### B. Performance Evaluation

Besides the simulator specifications described in Section II, several assumptions are made, which are considered reasonable in this data-flow processor. Each PE has several simple functional units which can enable the parallel matching of attribute-value pairs. The number of simple functional units in the PE would range from 1 to 10 due to the fact that there are no more than five attribute-value pairs in any condition of the left-hand side of the rules. Furthermore, the following assumptions are made in the simulation for the sake of evaluation:

- A simulation time unit \(\tau\) is set to 1 \(\mu s\).
- Each PE runs at 3 MHz clock = 1 \(\mu s/instruction\).
The routing time for a token to reach any PE is set to 1.5, and the time taken to process one pattern is 21 loops and this is validated by a given configuration of our data-flow architecture. Simultaneous distribution of many WME's to many PE's and allocation of conditions and O(n) iterations to different PE's have proven effective in delivering the parallelism inherent to the RETE algorithm.

The Tagged Token Data-flow Machine has been chosen for our simulation model. The allocation and distribution schemes we developed are exercised in our simulation. The RETE algorithm has been successfully implemented into a data-flow processing environment. The complete graph for a rule has been created to execute in a data-flow multiprocessor.

To detect and estimate the different levels of parallelism in the production matching step, various simulations have been undertaken. Conditions in the rule are executed in parallel. Our simulation results show that our data-flow multiprocessor can fire at a rate of 1000 rules per second in the absence of conflict resolution implementation. Although a conflict resolution is not taken into account in implementing a production system here, the results we obtained reveal that symbolic computations on a data-flow multiprocessor computer can indeed be processed efficiently. Comparison with conventional computers has shown that a high speed-up could be obtained from this approach.

However, some problems in applying data-flow principles of execution remain unsolved. One of the problems
is the programmability in high-level language. Also, a complete implementation of conflict resolution algorithm will be next undertaken. In conclusion, it appears that the data-flow principles of execution are not limited to numerical processing but will also find applications in some AI problems.

REFERENCES