Parallel Processing of Production Systems on Data-Flow Multiprocessors

Andrew Sohn

CENG Technical Report: 91-24

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2563
(213)740-4484

Acknowledgments

I would like to express my profound gratitude to Professor Jean-Luc Gaudiot, the chairman of my dissertation committee, for his valuable guidance, for his constant encouragement, and for his having enough faith and trust in me. I have benefited greatly from his extensive technical knowledge and experience during the course of this research. I would like to continue to value his insights and encouragement.

I am grateful to the other members of the dissertation committee: Professor V. K. Prasanna Kumar for his helpful comments on parallel algorithms and architectures, and Professor Christoph von der Malsburg for his valuable suggestions on connectionist and neural network aspect of production systems. Their suggestions and constructive criticism are of particular help in refining this thesis. I also want to thank Professor Paul S. Rosenbloom for his invaluable criticism on production system processing and Professor Dan I. Moldovan for showing me his view on parallel knowledge processing.

The Macro Data-flow Multiprocessor Project is an ongoing project at the University of Southern California. Thanks are due to my fellow members of the Data-flow Research Group at USC, Dr. Robert Lin, Namhoon Yoo, Chinhyun Kim, and Daekyun Yoon. Thanks also go to Mary Zittercob for her administrative help.

I would like to acknowledge the financial support received from the National Science Foundation under grant No. CCR-9013965.

My most sincere thanks go to my best friend, my wife *Sun Kyung*, for her moral support and endless encouragement. Without her support, this thesis would not have been possible. Finally, I would like to dedicate this work to the Lord *Jesus Christ* who has given me wisdom and courage through countless incidents, and will be with me for the rest of my life and *forever*.

Contents

List of Figures

List of Tables

Abstract

The importance of production systems in artificial intelligence has been repeatedly demonstrated by a large number of rule-based expert systems developed and used for the last decade. As the number and size of expert systems grow, there has however been an emerging obstacle in the processing of such an important application: the large match time. Much effort has been therefore expended on developing special algorithms and architectures dedicated to the efficient execution of production systems.

This thesis presents methods to improve the processing time of production systems. In particular, two approaches are undertaken to reduce the matching time of an inference cycle: software approach in *algorithm* level and hardware approach in *implementation* level. Bottlenecks of the most widely used match algorithm have been identified, based on which a new algorithm, the MRN match algorithm, is developed in algorithm level. Experimental results of benchmark production systems indicate that the MRN match algorithm can give 4-6 fold speedup over the best match algorithm, regardless of the type of a computer used. The MRN match algorithm can give a multiplicative effect when coupled with any match algorithm used for production systems.

In implementation level, data-flow principles of execution have been employed as architectural models. A generic production system is implemented in *micro* actor data-flow principle to explore the parallelism in *fine* grain processing. Parallelism in the *medium* grain processing of production systems is explicated through the implementation of a production system in *macro* actor data-flow principle. Simulation results indicate that the data-flow multiprocessors can give 17 fold speedup out of 32 processing elements when coupled with the MRN match algorithm. The approaches taken in this thesis demonstrate that the data-flow principles of execution are not limited to numerical computation but also find applications in non-numeric computation.

Chapter 1

Introduction

The importance of production systems in artificial intelligence (AI) has been repeatedly demonstrated by a large number of expert systems. Those earlier rule-based expert systems such as Prospector [13], R1 [42], and Mycin [10] have had a major impact on the development of expert systems. This rapid development of production systems are due mostly to the fact that the way that the productions are structured is very similar to the way that the people talk about how they solve problems. Among other characteristics the production system paradigm offers modularity, uniformity, and naturalness.

Modularity refers to the fact that individual production systems in the rule base can be added, deleted, or changed independently. They behave much like independent pieces of knowledge. Changing one rule, although it may affect the performance of the system, can be accomplished without having to worry about direct effects on the other rules, since rules communicate only by means of the context data structure; they do not call each other directly. This relative modularity of the rules is important in building the large rule bases of current AI systems.

Another general attribute of production systems is the uniform structure imposed on the knowledge in the rule-base. Since all information must be encoded within the rigid structure of production rules, it can often be more easily understood, by another person, or by another part of the system itself, than would be possible in the relatively free form of semantic net or procedural representation scheme, for example.

A further advantage of the production system paradigm is the ease with which one can express certain important kinds of knowledge. In particular, statements about what to do in predetermined situations are naturally encoded into production rules. Furthermore, it is these kinds of statements that are most frequently used by human experts to explain how they do their jobs.

As the number and size of expert systems grow, there has however been an emerging obstacle in the processing of such an important AI application: the large match time. In rule-based production systems, for example, it is often the case that the rules and the knowledge base needed to represent a particular production system would be on the order of hundreds to thousands. 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 rules can reach 90% of the total computation time spent in production systems [16]. The need for faster execution of production systems has spurred research in both the software and hardware domains, including connectionist architectures.

1.1 Parallel Processing of Production Systems

In the software domain, the Rete state-saving match algorithm has been developed for fast pattern matching in production systems [16]. The motivation behind developing the Rete algorithm was based on the observation, called *temporal redundancy* which saves in memory the information concerning the changes in the working memory between production cycles, and then utilizes them at a later time [9]. The algorithm has been put in practice for numerous production systems and is known to be the most efficient algorithm for pattern matching in production system interpreters.

Inefficiencies in the state-saving Rete algorithm were identified, based on which the non-state-saving Treat match algorithm was developed [45]. The improvements made over the Rete algorithm were motivated by the McDermott's conjectures which states that the retesting cost will be less than the cost of maintaining the network of sufficient tests [43]. Experimental results of the Treat algorithm on various benchmark production system programs demonstrated that this conjecture can be substantiated to a limited extent.

However, the Rete match algorithm and its improved pattern matcher are based on a *sequential* processing environment and running them on parallel machines requires much effort to yield a higher performance. The bottlenecks of the matching algorithm in *parallel* environments were identified, based on which the MRN approach has been introduced to parallelize the matching step [21]. When the MRN approach was initially developed, it was intended to run on parallel environments. However, its running on a sequential machine also gave a significant speedup over the sequential Rete algorithm and can give a multiplicative effect on any pattern matching algorithm [59].

For further speeding up the processing of production systems, another approach has been taken in the rule firing stage of the production systems, called multiple rule firing [35,40,46]. Firing many rules that do not cause any conflict in the database, this approach attempts to manifest the potential parallelism in the rule firing stage, if any. Furthermore, the removal of a selection stage from the production system paradigm would completely parallelize the entire production cycle, thereby resulting in true parallel production systems. To do this, however, a careful analysis of the rule dependencies has to be performed to ensure the correctness of the endeavor. In other words, the solution resulting from the multiple rule firing must be equivalent to the one resulting from a single rule firing.

In the hardware domain, requirements of special architectures for production systems have been identified and various machines have been designed and/or built along the conventional von Neumann model of execution [1, 25, 26, 27, 37, 38, 47]. Among the architectures, a special architecture called DADO, dedicated to production system processing, has been built and operational since 1985 [64]. Performance evaluation of that machine with 1023 8-bit processors has been reported in [63].

Production systems have also been implemented on a general purpose computer, Encore Multimax, a tightly-coupled shared memory multiprocessor with 2-20 processing elements [25]. An objective behind this study is to investigate the very fine grain parallelism in production systems. Experimental results indicated that a shared memory multiprocessor can give a significant speedup

when the fine grain parallelism is utilized.

The suitability of message passing computers for production system processing has been investigated and the evaluation of their performance has been made on the Nectar simulator, a message passing multicomputer with low message overhead [1,27]. Simulation results of the benchmark production systems on Nectar indicated that the loosely coupled message passing multicomputer can be effective in processing of production systems, provided that tasks can be evenly distributed.

The conventional control-flow model of execution described above, however, is limited by the "von Neumann bottleneck" [7]. Architectures based on this model cannot easily deliver large amounts of parallelism [3]. The potential parallelism in the problem will not be fully manifested due to the von Neumann bottleneck. Furthermore, the lack of programmability in the von Neumann model of execution would limit the potential parallelism in the given problem. The data-driven model of execution has therefore been proposed as an alternative to solve these problems. These principles have been surveyed [62,68]. Various research efforts on the development of data-flow multiprocessors and languages have been ongoing for the last two decades [20]. Prototype data-flow multiprocessors including the Epsilon project of the Sandia National Laboratory [20] and the Sigma-1 data-flow supercomputer of the Electro-Technical Laboratory in Japan have been built and currently operational [31,32,].

The applicability of data-flow principles of execution to various numeric computations such as Partial Differential Equations [41], Digital Signal Processing [19], etc. has been investigated, where issues related to how to map the numeric algorithms onto data-flow multiprocessors are discussed. Simulation results have indicated that the data-flow principles of execution can indeed extract from the given problems as much the potential parallelism as the given problems have.

The applicability of data-flow principles of execution to symbolic computations (or AI processing) has been investigated [21]. The pattern matching operation of production systems has been successfully mapped onto a fine grain dynamic data-flow multiprocessor and its success has demonstrated that data-flow principles of execution can also find some applications in symbolic computation [22].

To the characteristics of AI problems, the medium grain processing approach, called *multilevel macro actor/token* has been applied to implement a subset of symbolic computation. A generic production system has been mapped onto a macro data-flow multiprocessor simulator [59]. This medium grain approach utilizes the distinctive characteristics that AI problems exhibit and has given a significant performance improvement over the fine-grain processing.

1.2 Adaptive Processing of Production Systems

In an effort to find a completely *new* way of processing production systems, several attempts have been recently made along the connectionist networks [17,54,55,66,67]. The underlying argument in these various attempt is: The ultimate goal of developing AI production systems would be the modeling or at its best the simulating of human intelligence. At the same time, connectionist architectures or neural networks (NN) try to model or simulate the way in which the human brain behaves. Both AI and NN appear to have been going towards an ultimate common goal, achieving the *human intelligence*. However, these two fields have been coming from two completely different paths to achieve this common goal. If these two different approaches are to achieve the human-like intelligence, one should come to believe that there must be a common ground.

If the above argument is put into a different perspective, we would come to conclude that (1) a set of efficient techniques developed in AI may be able to help solve the problems that NN have difficulties in solving, and (2) by the same token, a set of efficient techniques developed in neural networks may be able to help solve the problems AI finds difficult. For example, the conventional AI search problems such as Traveling Salesman Problem, N-Queen Puzzle, etc. are found to have difficulties for large problems size due mostly to the fact that the processing time to search the state space is exponential. However, neural networks are known effective in solving such AI search problems [33].

The argument just described above is where the connectionist production systems come in to believe that production systems can utilize the techniques developed in neural networks. Employ-

ing techniques developed in connectionist architectures, the production system paradigm is believed to be equipped with the following capabilities which otherwise would be difficult to achieve: massive parallelism, quick search, and fault tolerance.

The first advantage of connectionist production systems would be in achieving massive parallelism. A set of simple neuron-like processing elements may hold the key to some important aspects of intelligence [29]. The second advantage, quick search, refers to the fact that neural networks can retrieve relevant items very quickly and without apparent effort. Even when incomplete information is present, neural networks can sometimes interpolate across the missing data [30]. The third advantage, fault-tolerance, lies in the fact that in many domains, the recognition abilities of neural networks far exceed that of other systems. When some of the neurons malfunction, the results produced by the neural networks would be imperfect but still usable since each macroscopically important behavior of the network is implemented by many different microscopic units.

However, there are problems that neural networks have to overcome. Among these problems, neural networks find it difficult to implement the variable binding problem which AI systems commonly use. Another disadvantage stems from the fact that neural networks have difficulties in solving problems associated with a temporal sequence such as production systems, planning problems, etc. Furthermore, neural networks have to develop good learning algorithms in order to be able to handle many different types of problems in the real world.

Multiple layers of feedforward networks have been proposed to suit the production system paradigm [17]. In order to represent the search space in neural networks, local representation is used in, where a concept is directly assigned to a particular neuron [55]. A particular implementation of this local representation for production systems has been reported, where the three layers of the ring-structured network was used [56]. The main objective behind using such layers is in reducing the pattern matching time. However, the local representation has been found difficult to implement the variable binding. Another important drawback for the local representation is in the lack of scalability, i.e., it would be impractical to use the local representation when the problem size is large and varying.

To avoid such drawbacks in local representation, a distributed representation has been developed, where a concept is assigned to a pattern of activity of many or all neurons [67]. While it partially permits the variable binding, the distributed representation requires a sophisticated learning algorithm in order to be practical [14]. To avoid this impracticality for large problem size, the hierarchical representation has been introduced, based on which production systems are transformed and represented in n-dimensional feature space [54]. This thesis limits the scope to the parallel processing perspective and leave the adaptive processing perspective for future research. However, the new representation technique has been given to guide the development of production systems in connectionist architectures.

1.3 Organization of the Thesis

This thesis attacks the problem of parallel processing of production systems in two phases: (1) the development of a parallel match algorithm for production systems to suit parallel processors, (2) the implementation of the parallel algorithm on a data-flow multiprocessor and on a conventional sequential processor.

Chapter 2 gives a brief introduction to production systems and the Rete match algorithm. A brief introduction to data-flow principles of execution is also presented in the chapter as well. Three different data-flow machine models, a static machine, a dynamic micro machine model, and a new machine model, our macro data-flow multiprocessor, are to follow the basic principles. Problems of processing production systems in general are identified and approaches to the problems are described from two different levels, sequential and parallel algorithm level, and parallel machine implementation level.

Chapter 3 identifies the suitability of the data-flow principles to the implementation of the production system paradigm. Issues related to mapping production systems to data-flow architectures are presented along the suitability. The inefficiencies of the Rete algorithm in multiprocessor environments are identified, based on which a new algorithm, called *a Multiple Root-Node* (MRN)

Chapter 2

Background

Necessary background is briefly presented in this chapter. A brief introduction to production systems is given, followed by a description of the Rete match algorithm for production systems. An example is also given to illustrate the underlying concepts. The three data-flow principles of execution are presented: static, dynamic, and macro actor. Various machine architectures developed or under development are presented to contrast the principles. Problems associated with production systems are discussed. Possible solutions to the problems which have been reported or currently undertaken are summarized from the parallel processing perspective and the adaptive processing perspective.

2.1 Production systems

The production system paradigm is described in this section, followed by the Rete algorithm, the most widely used match algorithm for production systems.

2.1.1 The production system paradigm

A production system paradigm shown in Figure 2.1 consists of a *production memory* (PM), a *working memory* (WM), and an *inference engine* (IE). PM (or rulebase) is composed entirely of conditional statements called productions (or rules). These productions perform some predefined

actions when 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. LHS consists of one to many elements, called *condition elements* (CEs) while RHS consists of one to many actions.

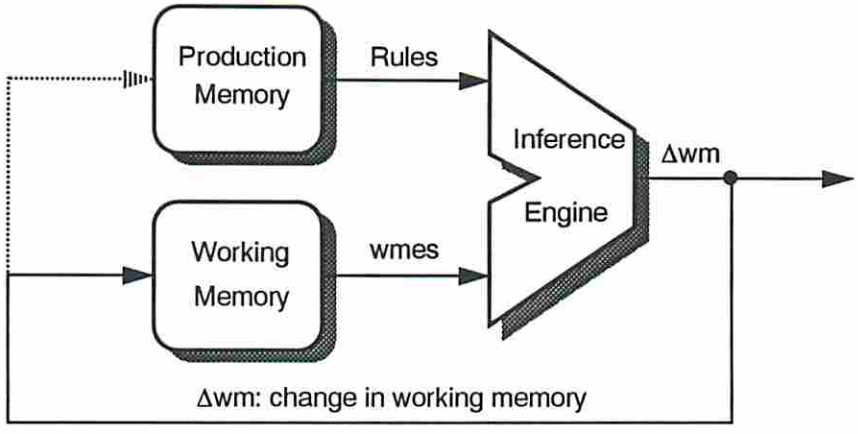


Figure 2.1: An architecture of production systems

The productions operate on WM which is a database of assertions called *working memory elements* (wmes). Both condition elements s and wmes have a list of elements, called *attribute-value pairs* (AVPs). The value to an attribute can be either *constant* or *variable*: the former in lower case and the latter in upper case (which are similar to the *Prolog* representation). To further illustrate, consider the following simple production system:

Production Memory

```
Rule1: [(c X) (d Y)]          ;CE1
       [(b Y)]                ;CE2
       [(p 1) (q 2) (r X)]    ;CE3
       →
       [Remove (b Y)]         ;Action 1
       [Make (c 1) (d Y)]     ;Action 2
```

Working Memory

```
wme1: [(p 1) (q 2) (r *)]
wme2: [(r =) (d +)]
wme3: [(c *) (d +)]
wme4: [(b 3)]
wme5: [(b +)]
wme6: [(p 1) (q 3) (r 7)]
```

The rule above will perform action 1 and action 2 when all the conditions are verified in the working memory. The inference engine executes an inference cycle which consists of three steps: *pattern matching*, *conflict resolution*, followed by *rule firing*:

- *Pattern Matching*: The LHSs of all the production rules are matched against the current wmes to determine the set of satisfied productions. wme1 in the above example can satisfy CE3 with the variable X instantiated to * whereas wme6 cannot. This step will eventually identify three wmes, 1, 3, and 5 which all satisfy the above rule with X and Y instantiated respectively to * and +.

- *Conflict Resolution*: If the set of satisfied productions is non-empty, one rule is selected for execution in the next step. Otherwise, the execution cycle halts because there are no satisfied productions. In the above example, 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 above example, 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.

2.1.2 The Rete match algorithm

The Rete match algorithm is a highly efficient approach used in the matching of objects in production systems [16]. The simplest possible matching algorithm would consist in going through all the rules and WMEs one by one until one (or several) match(es) has (have) been found. The Rete algorithm, however, does not iterate over the WMEs 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*, that there is little change in the working memory between production cycles [9]. 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 useful for sequential uniprocessor environments. When structural similarity is implemented in a multiprocessor

environment, it would cause a substantial amount of communication overhead [22,24].

Therefore, given a set of rules a network is built which contains information extracted from the LHSs of the rules. Figure 2.2 shows a network built for Rule 1 shown earlier, which consists of several types of nodes:

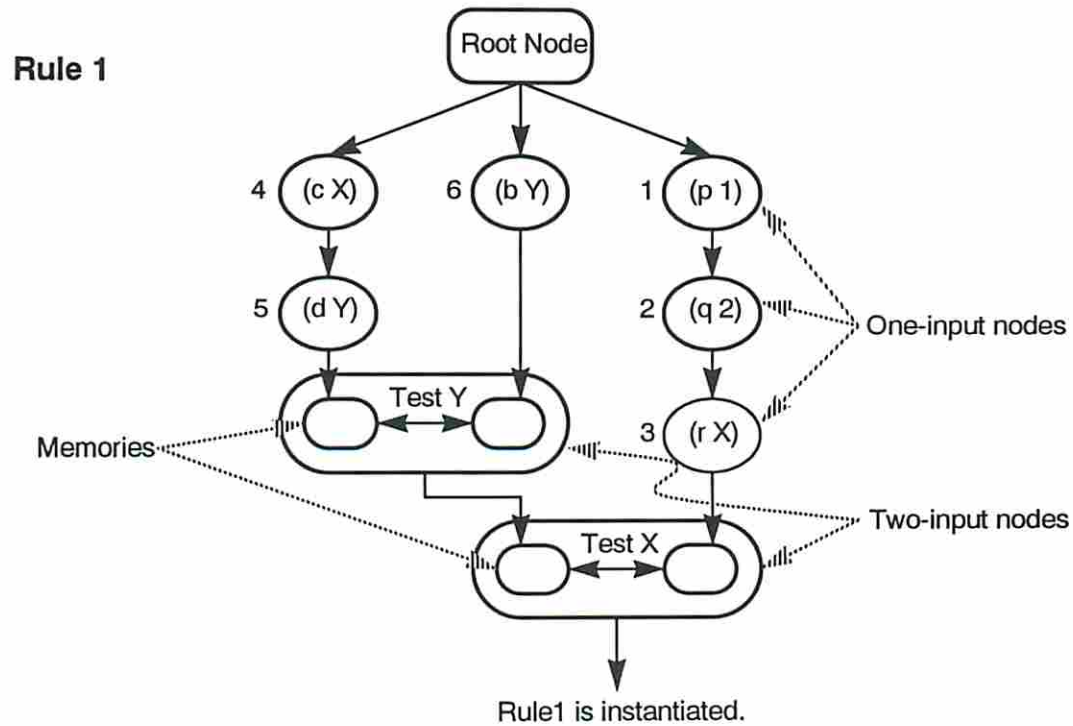


Figure 2.2: A Rete network for Rule 1.

- The *Root Node* (RN) distributes incoming data-tokens (or wmes) to sequences of children nodes, called one-input nodes.

- *One-Input Nodes* (OIN) test intra-element features contained in a condition element, i.e., compare the value of the incoming wmes to some preset value in the condition element. For example, the first condition element of Rule1 contains 3 intra-element features and therefore 3 OINs 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 inter-condition features contained in two or more

condition elements. The variable *X*, which appeared in both CE1 and CE3, must be bound to the same value for rule instantiation. Attached to the two-input nodes are left- and right memories in which wmes matched through one-input nodes are saved. Another type of two-input node is called *Negated Two-Input Nodes* (NTIN) which are designed to process condition elements preceded by – (*not*). NTIN tests to determine whether no wme satisfies it. The result from two-input nodes, when successful, are passed to nodes, called terminal nodes.

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

2.2 Data-flow Principles and Machines

The three data-flow principles of execution are now presented: static, dynamic, and macro actor. Various machine architectures developed or under development are presented to contrast the principles.

2.2.1 Basic data-flow principles of execution

Developing and using a large scale MIMD (Multiple Instruction Multiple Data) multiprocessor involves much effort both at the software level and at the hardware level [3]. Among many issues at the software level is *programmability*. 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 current state of the art in programming MIMD machines relies on the programmer to express all parallelism using low-level synchronization and communication constructs.

Another important issue that must be addressed when developing and using a large multiprocessor is *synchronization*, which refers to the ordering of instruction execution according to their data dependencies. Cooperating processes in a multiprocessor environment must often communicate and synchronize. Typical synchronization methods exercised in the conventional von

Neumann model of execution are use of shared variables and message passing. When using shared variables, either mutual exclusion or conditional synchronization is commonly employed to ensure that no other process enters the critical section and modify the shared variables. Problems associated with these synchronization methods are that if the critical section is busy, the process attempting to enter the critical section must wait, resulting in wasting of precious computing resources if the wait is long. Furthermore, if members of a group of processes which hold resources are prevented indefinitely to access resources held by other processes within the group, deadlocks will immediately arise in this otherwise highly concurrent multiprocessor system.

The data-flow model of computation, proposed in the early 1970s as an alternative to the conventional control-driven model of computation, explicitly addresses the issue of programmability and the synchronization. The first problem, programmability, is resolved by use of functional languages, where any variable can be assigned a data value only once, called the *single-assignment* principle. Programs are compiled into data-flow graphs, which represent the data dependencies among instructions. An instruction executes as soon as all the required operands are available. Since instruction execution is triggered by the availability of operands, the computation is capable of tolerating arbitrary memory latencies and allows data to arrive in an arbitrary order.

Data-flow principles of execution offer the runtime synchronization of operations, which the conventional multiprocessor system has difficulties in overcoming. Synchronization is enforced at the instruction level because every instruction waits for all its operands to be produced before executing. Data-flow principles are parallel in nature and thus would allow a very large number of different tasks to be efficiently and safely allocated to the entire machine. A comprehensive surveys on data-flow machines can be found in [62,68].

2.2.2 The static data-flow principle

The fundamental principles of data-flow developed by Jack Dennis in the early 1970s [11] form the foundations of what became known as the *static* data-flow architecture and also served as the basis for subsequent developments, including *dynamic* data-flow systems. A data-flow program is

represented as a directed graph consisting of *actors*, which represent instructions, and *arcs*, which represent data dependencies among actors. Operands are propagated along the arcs in the form of data packets, called *tokens*.

The execution of an instruction is called the firing of an actor. The basic instruction firing rule common to all data-flow systems is that an actor is fired as soon as tokens are present at all its input arcs. When an actor fires, a token from each input arc is removed and a result token is placed on each of its output arcs. Figure 2.3 shows an example of a data-flow graph to evaluate the expression, $x=a*b - c+d$.

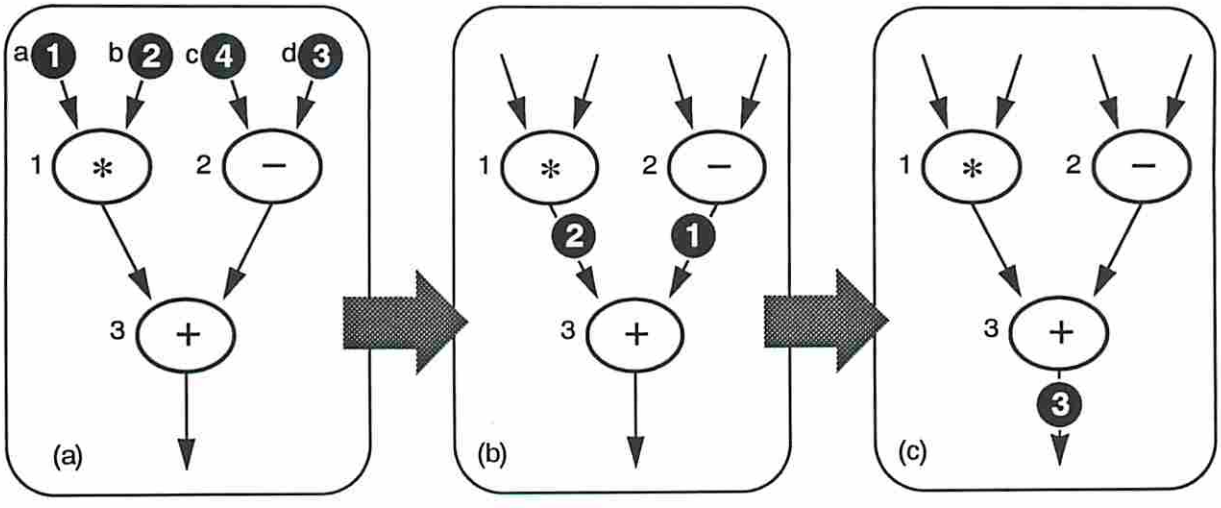


Figure 2.3: Snap shot of the data-flow graph for an expression $x = a*b - c+d$.

In the above example, the actor labeled 1 is enabled as two tokens, 1 for a and 2 for b, have arrived on input arcs of actor 1. The firing of this actor produces a new token, 2, which is received by actor 3. Similarly, actor 2 will fire when the two tokens, 4 for c and 3 for d, arrive at the input arcs of actor 2, resulting in a new token 1. These two new tokens, 2 and 1, in turn, enable the firing of the actors 3, producing another new token, 3.

For a simple graph, such as the one just discussed, the basic firing rule is sufficient to guarantee proper execution. Problems arise when a graph is reentrant, as for example, in the case of a loop body. In this case it could happen that an actor is enabled while a token is still present on one of its output arcs. If it fires, more than one token could be present on a single arc. Several schemes have

been proposed to deal with this problem. The static data-flow approach simply disallows more than one token to reside on any one arc. The single-token-per-arc constraint is enforced through *acknowledgment signals* generated by additional destination address fields in an activity template. In the data-flow graph, acknowledgment signals are shown by additional arcs leading from consuming to producing actors. A token on an acknowledgment arc indicates that the corresponding data arc is empty, i.e., is ready to receive a token. When an actor fires, it sends one signal along each acknowledgment arc.

A small model consisting of four micro-coded processors was built; however, it suffered from a number of serious drawbacks [12]. Among them, the feedback interpretation of data-flow programs using acknowledgment arcs allows only a limited amount of parallelism to be exploited because consecutive iterations of a loop can only partially overlap in time but not concurrently. Thus, with acknowledgment signals, only a pipelining effect may be achieved, where the maximum length of the pipeline is bounded by the critical path through the loop body. Another undesirable effect of the feedback interpretation, in addition to making the machine more complex to design and operate, is the fact that token traffic is doubled. Since communication is a serious problem in any multiprocessor system, keeping the number of tokens to a minimum is an important requirement.

2.2.3 The dynamic data-flow principle

The performance of a parallel machine increases significantly when loop iterations and subprogram invocations can proceed in parallel [2]. To achieve this, each iteration or subprogram invocation should be able to execute as a separate instance of a reentrant subgraph. To distinguish between activities belonging to different instances, the basic activity names are extended: each carries an iteration count and also the procedure context within which it executes. Thus, each actor is replicated for every loop iteration and every procedure invocation. This replication, however, is only conceptual. In the implementation, only one copy of any data-flow graph is actually kept in memory but each instruction is fetched and executed independently for any iteration number and

procedure context.

With this approach, each arc can be viewed as a bag that may contain an arbitrary number of tokens with different tags (colors). Each *activity* is uniquely identified by a tag of the form c.s.i, where *c* is the *context*, s is the *activity name* (instruction number), and *i* is the *iteration number*. The context uniquely identifies the current procedure invocation while the iteration number uniquely identifies the current iteration instance. Actors in the tagged-token data-flow principle will fire when tokens carrying identical tags are present at each of its input arcs. This eliminates the need for any feedback signals, thus increasing parallelism and decreasing token traffic. Data-flow machines that employ this method are called *tagged-token* or *dynamic* data-flow machines. Figure 2.4 illustrates the tagged-token principles. On the left arc there are four tokens each of which carries a different tag while on the right arc five tokens present again with different tags. The only identical pair found on both arcs is the one with filled which enables the actor labeled 1. When the actor fires, When the actor 1 fires, it consumes the two filled tokens and produces the new result token, of which the color (or tag) is again filled, as depicted in Figure 2.4.

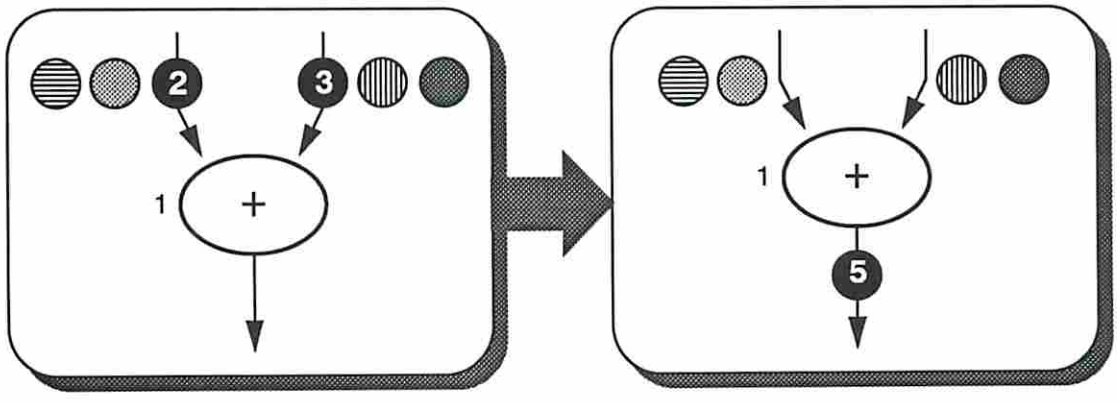


Figure 2.4: Snap shot of a data-flow graph for the dynamic tagged-token principle.

2.2.4 A dynamic data-flow machine

The MIT Tagged-Token Data-flow Machine is based on the principles just outlined [5]. It consists of a number of identical PEs connected through an *n*-dimension hypercube packet-switching net-

work. Figure 2.5 shows the organization of a single PE of the Tagged-Token Data-flow Machine. It consists of a Matching Store Unit (MSU), an Instruction Fetch Unit (IFU) with access to a Program and Constant Memory, an Arithmetic Logic Unit (ALU), a Token Formatting Unit (TFU), and a Token Queue (TQ).

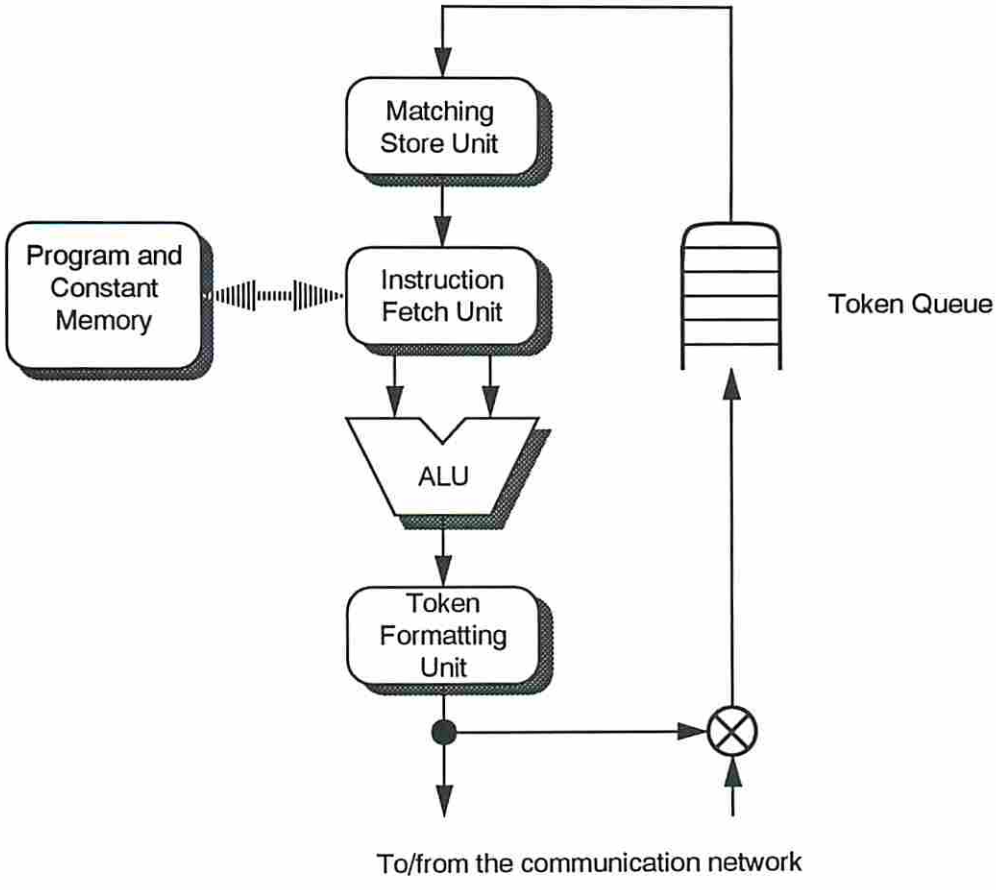


Figure 2.5: Organization of a single processing element

A program under execution is distributed over the program memories of the different PEs. The Token Queue contains data tokens produced by previous firings of actors in the current PE or other PEs. In the latter case, they have arrived in the Token Queue through the communication network. Tokens are removed from the queue by the MSU. If a token is destined for a monadic operator, it is passed directly to the IFU. Otherwise, it is for a dyadic operation; a matching phase is initiated. This involves comparing the tag of a token to the tags of all tokens currently in the MSU. If a match is found, the token pair is forwarded to the IFU; otherwise, the token is stored in the MSU to await

the arrival of its partner.

The IFU retrieves an instruction from the Program Memory according to the tag carried by the operand token(s). The fetched instruction may also include a literal or a reference to a constant to be used as an operand. In the latter case, the constant is fetched immediately from the Constant Memory. When a complete executable packet is assembled, consisting of an opcode, operand values (literals, constants, or values carried on tokens), destination references, and a tag, it is passed to the ALU for execution.

The ALU computes a new value according to the opcode and, in parallel, the new tags are derived. The result and the new tags are sent to the TFU, where new tokens are assembled and forwarded to the local Token Queue, to another PE if the destination address is non-local, or to an I-Structure Storage, if the operation is an access to a structure.

To solve the problem of large data structures, the concept of I-structures has been proposed by [6]. An I-structure may be viewed as a repository for data values, which obey the single-assignment principle. That is, each element of the I-structure may be written into only once. After it has been filled, it may be read any number of times.

The MIT Tagged-Token Data-flow Machine was the first to use the dynamic data-flow principles and the concept of I-structure. It has been simulated on a Multiprocessor Emulation Facility consisting of 32 TI Explorer Lisp Machines interconnected by a high-speed network, which has been operational since January 1986. The experiences gained by the MIT Tagged-Token Data-flow Machine are being used to build a prototype of its successor, the Monsoon Architecture.

2.2.5 The Manchester data-flow machine

The same principles of tagged-token data-flow were also developed *independently* at the University of Manchester. The resulting Manchester Data-flow Computer is the *first actual* hardware implementation of a dynamic data-flow computer ever built [28]. The prototype, consisting of a single execution ring, became operational in October 1981. A block structure of this ring is shown by Figure 2.6.

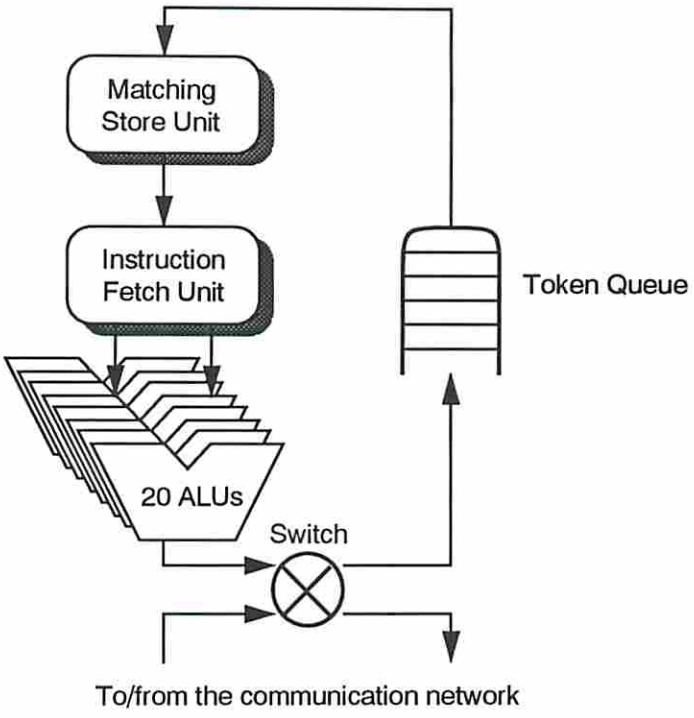


Figure 2.6: Organization of the Manchester data-flow computer.

The inadequacy of handling complex data structures has been perhaps the most serious drawback of the Manchester Data-flow Computer. Another problem was the implementation of an efficient Matching Store Unit. First, the pipelined hashing strategy used for the Matching Store Unit requires a rather long pipeline and thus degrades performance under low parallelism. Second, a separate Overflow Unit was necessary to prevent menacing overflows of the Matching Store Unit. This has later been enhanced by dynamic resource scheduling in the form of a hardware throttle. Finally, the ability to feed 20 concurrently working ALUs by just one pipeline did not prove feasible due to the Matching Store Unit bottleneck. Though the Manchester Data-flow Computer was too small to run any significant applications, it was able to demonstrate that pipelines in a data-flow computer can be kept busy almost effortlessly [5].

2.2.6 The SIGMA-1 data-flow machine

The SIGMA-1, built at the Electro-Technical Laboratory in Japan, is the most ambitious tagged-token data-flow computer to date [32,49]. It is a supercomputer for large-scale numerical compu-

tations and has been operational since early 1988. It consists of 128 Processing Elements and 128 Structure Elements interconnected by 32 Local Networks (10 by 10 crossbar packet switches) and one global two-stage Omega Network. Sixteen Maintenance Processors are also connected with the Structure Elements and with a Host Computer for I/O operations, system monitoring, and maintenance operations.

Figure 2.7 shows the structure of one processing element, implemented with several gate array chips and a large memory. It operates as a synchronous two-stage pipeline. The first stage is the firing stage, consisting of a FIFO Input Buffer, an Instruction-Fetch Unit accessing a Program Memory, and a Waiting-Matching Unit supported by chained hashing hardware. The Input Buffer receives data tokens generated locally or by other PEs. The processing element number is stripped before a token enters the Input Buffer.

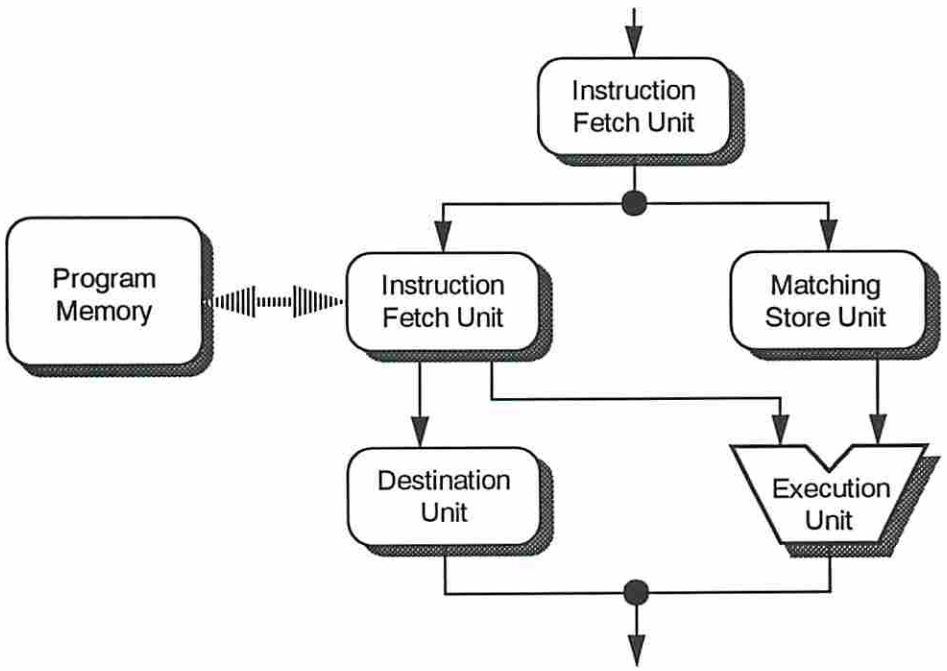


Figure 2.7: Organization of a processing element of the SIGMA-1

The Instruction-Fetch Unit and Waiting-Matching Unit work simultaneously on the same token, transmitted from the Input Buffer. If no match is detected, the fetched instruction is discarded. Otherwise, it is passed to the second stage for execution. This stage consists of an Execution Unit and a Destination Unit. The former includes a Floating-Point Arithmetic Unit, a Multiplier, and a

Structure Address Generator. The Destination Unit produces the destination addresses for the result tokens, which are routed to the local Input Buffer or to a remote PE.

The SIGMA-1 also contains the first implementation of I-Structure Storage for handling large structures and it provides hardware support for local memory management. This includes buddy allocation and several waiting-matching functions, such as "sticky/non-sticky" read operations. Unlike the Manchester machine, however, these functions are used only for maintaining loop constants, rather than handling general data structures.

The implementation of each PE as a synchronous two-stage pipeline keeps the total size of a PE small and suitable for scaling-up. The short pipeline is especially advantageous when the parallelism in a program is small. It demonstrated a performance of 170 MFLOPS on a small program while the peak performance reaches 427 MFLOPS [31].

2.2.7 The Macro data-flow principle

One of the main problems of tagged-token data-flow machines has always been the implementation of a Matching Store Unit. For reasons of performance, an associative memory would be ideal. Unfortunately, the amount of memory needed to store tokens waiting for a match tends to be very large, which renders this approach impractical or, at least, not cost-effective. As a result, all existing data-flow machines use some form of hashing, sometimes supported by hardware hash tables. However, hashing techniques are typically not fast enough to be used effectively as a single stage in the instruction processing pipeline. This often results in a long hashing pipeline, like that of the Manchester Data-flow Computer, thus degrading performance of sequential applications.

The *macro* data-flow principle has therefore been proposed by Gaudiot and Ercegovac to alleviate the problems associated with the number of tokens to be matched at the Matching Store Unit [18]. A macro actor is a collection of scalar instructions. The objective behind lumping instructions into one larger unit is to improve performance by exploiting locality within these larger units. In fine grain data-flow computation, high overhead needed to respect the functionality in execution will result in poor performance at low levels of parallelism. Indeed, to execute fine grain graphs

(where each actor represents a single instruction), execution, communication, computation overhead must be enforced to associate with actual computation actors to insure the correct execution. Therefore, overhead problem will be inevitably created and will degrade performance.

When the grain size of a graph is increased, in other words, an actor now represents several operations instead of one single operation in graphs, the overhead problem can be easily alleviated. Indeed, with actors of various sizes, the amount of non-compute operations and the cost of communication can be significantly reduced. However, one should also note that the increasing size of actors (with larger granularity) may reduce the available parallelism in programs and increase memory and communication latency. Hence, forming macro-actors from fine grain micro actors is a trade-off between latency and parallelism.

The Macro Data-flow Project currently underway at USC is based on the macro principle. The machine architecture is organized in two level: the dynamic data-flow model at the high level and the conventional von Neumann model at the low level. The dynamic data-flow principle employed at the high level exploits the programmability and the inherent synchronization methods the dynamic data-flow principle offers. The conventional von Neumann model at the low level exploits the program locality to reduce the overhead incurred in the token traffic and tag matching. The machine consists of 64 PEs, connected through a six-dimensional hypercube interconnection network. In between two neighboring PEs are *three* facilities: two communication nodes and a link. Each PE has *four* facilities connected in a pipe, as is done in the MIT Tagged-token Data-flow Computer, shown in Figure 2.5. An initial assessment on this hybrid machine indicates that the Macro data-flow indeed alleviates the problems associated with the dynamic principles, where the tag matching time is of major concern in developing data-flow multiprocessor [41].

2.3 Processing Production Systems

Problems associated with the Production system paradigm are identified and approaches to the problems are iterated from two different perspectives: parallel processing and adaptive processing perspectives. The approaches taken in this thesis are then presented from the two perspectives.

2.3.1 Inefficiencies in the production system paradigm

The production system paradigm described earlier presents two inefficiencies. First, the processing mechanism itself is inherently sequential, i.e., the three steps must be performed in sequence by an inference engine. The definition embedded in the production system prevents its efficient evaluation on a parallel machine. Second, there are heavy memory dependencies in the matching step. All the condition elements and wmes to be matched must be repeatedly stored and recalled whenever a new inference cycle is started. Indeed, the time taken to match conditions over rules can reach 80%-90% of the total computation time spent in production systems [16].

The production system paradigm can be viewed as a state space search, consisting of *local* and *global latencies*. The search space with both latencies is depicted in Figure 2.8, where PM, CR, and RF stand respectively for pattern matching, conflict resolution, and rule firing.

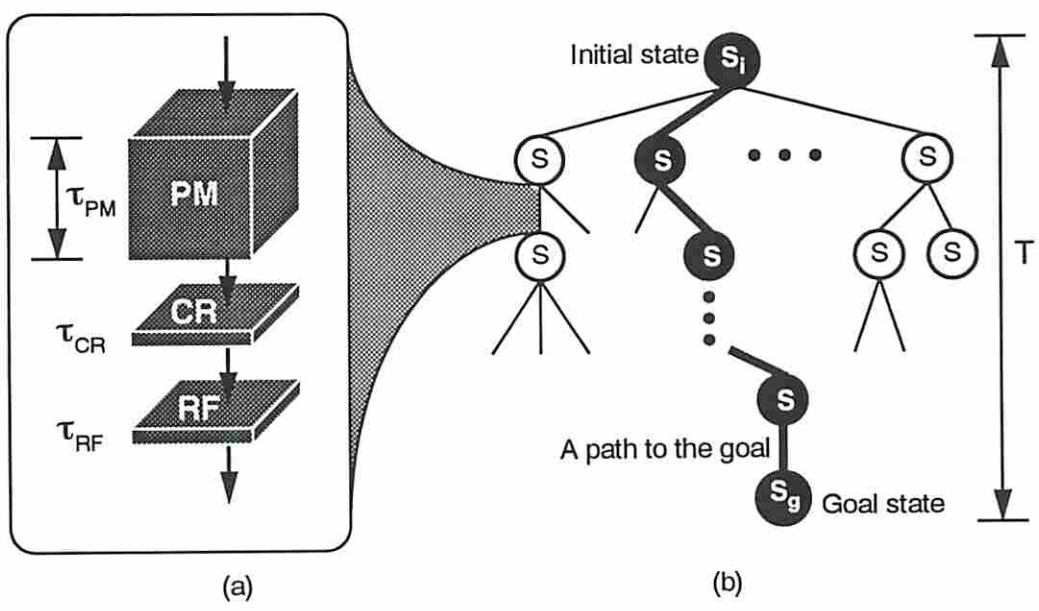


Figure 2.8: Production systems as a search. (a) Local latencies. (b) Global latency.

The local latency, τ, refers to the processing time of a *particular* inference cycle in the production system paradigm. Each step in the production cycle such as matching, conflict resolution, or rule firing is considered a local latency, as shown in Figure 2.8(a). The global latency, T, depicted in Figure 2.8(b), is the processing time of a nondeterministic number of inference cycles in the

search tree of the state space. Given the initial state, the inference engine finds the next state by executing the inference cycle. Based on the control strategy or heuristics, the system will select the state in the search tree it will explore. The global latency, T, is thus linearly proportional to the number of states, n, to be explored in the search tree, i.e., $T \propto n\tau$, when no backtracking is made.

2.3.2 Parallel processing approaches to solving the inefficiencies

A set of techniques that have been identified to reduce the latencies in the PS paradigm can be classified basically into two categories: (1) parallel hardware/software processing and (2) adaptive/ heuristic processing [69]. The first approach, parallel processing of production systems, attempts to reduce the total processing time by using many processing elements (PEs). From the hardware perspective, applying many processing elements to pattern matching will reduce the local latency in an inference cycle, thereby reducing the total processing time, as depicted in Figure 2.9(a).

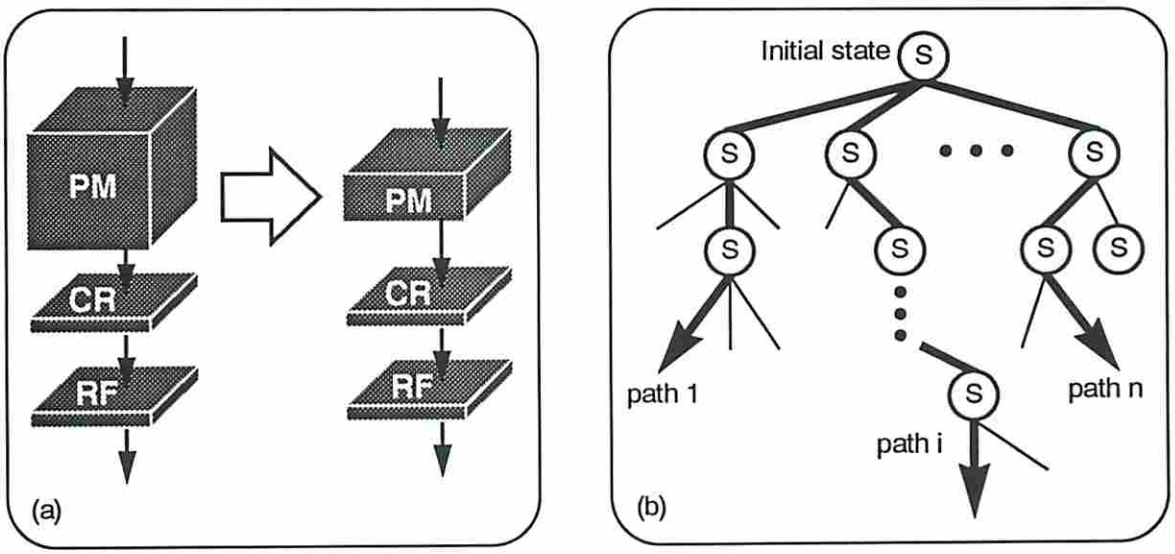


Figure 2.9: Parallel processing of production systems. (a) Reduction in processing time of the matching step, (b) Simultaneous exploration of the search tree by having many PEs follow multiple paths, path 1,..., path n.

Exploring the search tree simultaneously by many PEs would eventually reduce the total processing time, i.e., reduction in the global latency. Figure 2.9(b) illustrates this parallel processing

of search tree. This simple hardware approach with infinitely many PEs can eliminate problems associated with backtracking and can hopefully find a desired solution in a finite amount of time. However, this technique is impractical and too costly since for most practical AI problems the number of possible states in the search tree would be exponential. This leads to the development of parallel *software* to help parallel machines work efficiently.

2.3.3 Adaptive processing approaches to solving the inefficiencies

The second approach, adaptive/heuristic processing, aims at reducing the global latency, T, i.e., the total number of inference cycles. In this approach, production systems can be viewed as searching the state space to find a desired goal. The search can be performed to prune nonpromising branches in the search tree by use of an additional information such as heuristics and/or new rules learned either at compile time or at runtime. Putting together this additional information, the conventional production system architecture shown in Figure 2.1 can transformed into an adaptive production system depicted in Figure 2.10.

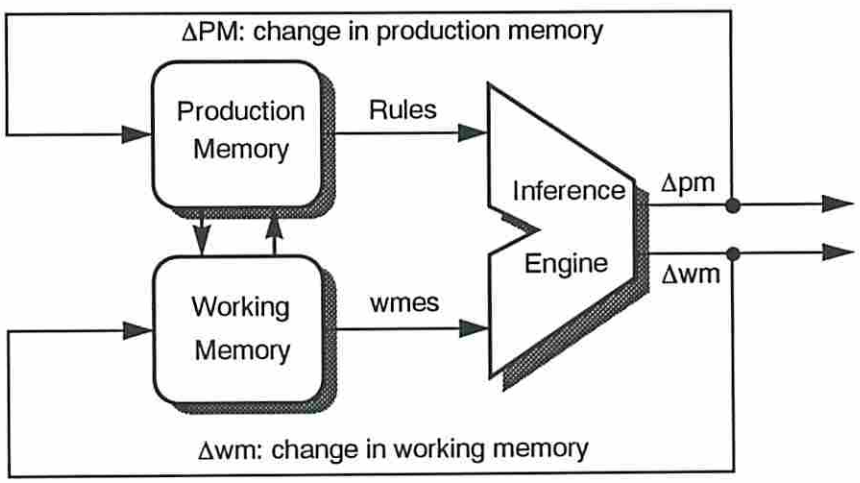


Figure 2.10: Adaptive processing of production systems, where the inference engine learns new rules or heuristics at either compile time or runtime.

There the inference engine generates not only the change in the working memory, ΔWM, but also the change in the production memory, ΔPM. A similar approach we have attempted for the

planning problem along the line of neural networks stems from the fact that by learning knowledge at run time the system will take less processing time to solve a similar problem at a later time [58].

Another important approach to this adaptive processing can be made by firing multiple rules in parallel. When many rules are fired in parallel, many paths in the search tree can be simultaneously explored, thereby reducing the total number of inference cycles executed in productions systems [39,40,46]. Figure 2.11 shows an example on the effect of reduction in the search space when multiple rules are fired in parallel [47].

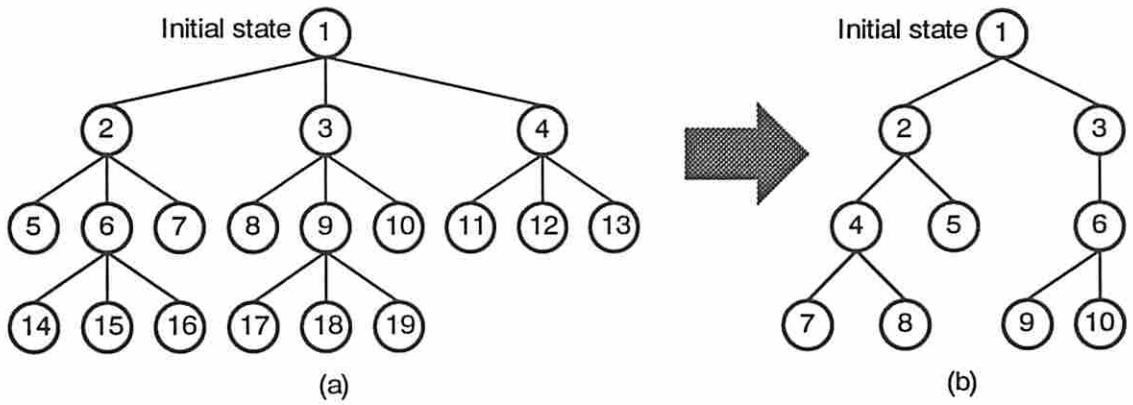


Figure 2.11: Reduction of global latency in the search space using adaptive processing techniques. (a)original search space with 19 states and 13 paths, (b) reduced search space with 10 states and 5 paths.

The original search space depicted in Figure 2.11(a) has 19 states and 13 paths whereas the reduced search space shown in Figure 2.11(b) now has 10 states and only 5 paths. The reduction in paths is 8, which is of significant. When the problem size is substantial, the parallel firing of multiple rules would significantly reduce the search space, thereby rendering intractable problems to tractable. However, there are many problems to be resolved for this to be practical, including rule dependency analysis. Much effort is currently being expended along this direction throughout the research community.

Chapter 3

Production System Processing in Data-flow

As we have discussed earlier, the production system paradigm has been implemented in various computer systems, ranging from conventional sequential uniprocessors, to general purpose multiprocessors, as well as special purpose multiprocessors dedicated to production systems. The study in this thesis has focused on the data-flow principles of execution. Since the underlying architectural model employed in this study is drastically different from the architectural model investigated thus far, many issues such as mapping production systems to data-flow principles, allocation of resources, etc. have to be clarified before production systems are implemented on the target data-flow multiprocessor.

We shall first identify in this chapter the suitability of the data-flow principles of execution to processing production systems, followed by a discussion of the issues related to mapping production systems onto data-flow multiprocessor. In the course of mapping, any necessary modifications to the chosen algorithm will be identified and resolved to suit the target multiprocessor. Bottlenecks of the Rete algorithm in data-flow multiprocessor environment are identified and solutions to them are presented, resulting in the MRN network. This new match algorithm invalidates the conventional policies on resource allocations. A new policy for the allocation of productions has therefore been developed and presented in this chapter, followed by a dynamic wme distribution policy.

3.1 Suitabilities

Based on the background information discussed in the previous chapter, we present the suitability of data-flow processors for production system processing. The necessary mapping schemes to fit the Rete match algorithm and the data-flow multiprocessor are identified in this section. Bottlenecks in the Rete algorithm are identified, which will result in the development of a new match algorithm.

3.1.1 Suitability of data-flow principles to production systems

The applications of data-flow computers studied thus far fall basically into the area of numerical computations such as signal processing [19], partial differential equation solvers [41], 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 nondeterministic 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 (list) 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, say, the basic Tagged Token Data-flow Architecture.

- Fewer primitive functions are needed in symbolic computations than are required for numer-

ical 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 shall discuss later in Chapter 5, the above observations lead to the development of a macro actor/token approach among data-flow principles of execution. Details on necessary modifications as well as an objective behind using data-flow principles of execution for the implementation of production systems are iterated in [22].

As we have demonstrated in the previous chapter, data-flow principles of execution and the Rete match algorithm present a 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 the following 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 actors in data-flow and comparison tests in the Rete algorithm are possible unless PEs 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 wmes. The counter for negated-pattern processing can be treated the same as other tags in the dynamic architectures.

3.1.2 Mapping production systems onto a multiprocessor

Mapping production systems onto multiprocessor systems has been investigated in several ways in the recent literature. Direct mapping employed in the DADO project uses "full distribution," which allocates a production to an available PE [63]. In this approach, production-level parallelism can be achieved by having several PEs operate simultaneously on wmes to match productions. In [23] a relevancy between the rules and the wmes is identified and used to directly allocate rules to PEs. The relevancy is defined as "A working memory element is relevant to a production if it matches at least one of its condition elements." However, the direct mapping of a rule to a PE is not likely to yield a good performance as Gupta has reported in his thesis [24].

It has been suggested by Bic that the semantic network can be directly viewed as a data-flow graph [8]. 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 Tenorio and Moldovan may be considered an indirect mapping [65]. In this approach, all productions are analyzed and grouped according to the dependency between productions to enable firing of multiple rules.

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:

3.1.3 The Rete algorithm in a multiprocessor environment

The Rete algorithm is a highly efficient matching algorithm for production system in a sequential processing environment [16]. When used in a parallel machine environment, it, however, presents

two apparent bottlenecks: one in the root node and the other in two-input nodes. Figure 3.1 illustrates the two bottlenecks. Assuming that each condition element (CE) is ideally distributed to a different processing element (PE), tokens coming into the root node will immediately pile up on the input arc of the root node since there is one and only one root node which can distribute tokens one at a time to all CEs. For the network shown in Figure 3.1 where there are n condition elements, the root node will have to make nm distributions to the network when m wmes are present on the input arc of the root node.

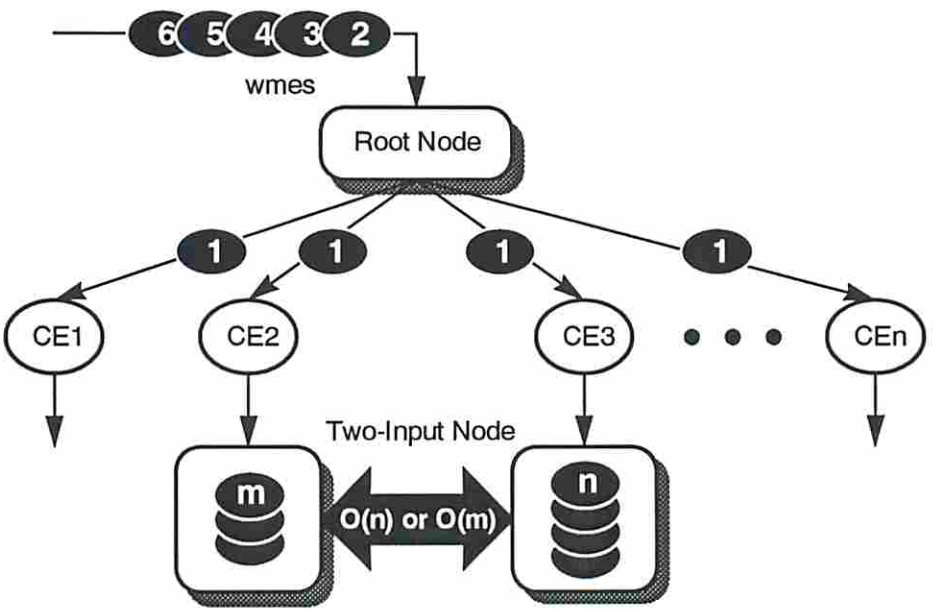


Figure 3.1: Two bottlenecks of the Rete algorithm in a multiprocessor environment. (1) piling up of wmes on an arc of the root node, which results in a sequential distribution of wmes to all CEs one at a time. (2) $O(n)$ or $O(m)$ comparisons in two-input nodes.

The second inefficiency can also be seen on Figure 3.1. 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 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, 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 wasting

time in searching through the entire memory, an effective allocation of two-input nodes and one-input nodes should be devised.

3.2 The MRN-based Match Algorithm

A new match algorithm is introduced to overcome the bottlenecks of the Rete match algorithm described above. Policies on allocation of productions and distribution of multiple wmes at runtime are introduced to support the new match algorithm.

3.2.1 The MRN network

The first bottleneck described above can be resolved by introducing *multiple* root nodes (MRN) in the network, as depicted in Figure 3.2. This introduction of multiple root node is based on the observation that a wme that has n AVPs never matches a condition element (CE) that has m AVPs where $n<m$. For example, a wme, [(a 1) (b 2)], cannot match a CE, [(a X) (b Y) (c Z)], where X, Y, Z are variable, since the wme is missing the third attribute-value pair (AVP) (c Z). However, a wme [(a 1) (b 2) (c 3) (d 4)] can match the CE.

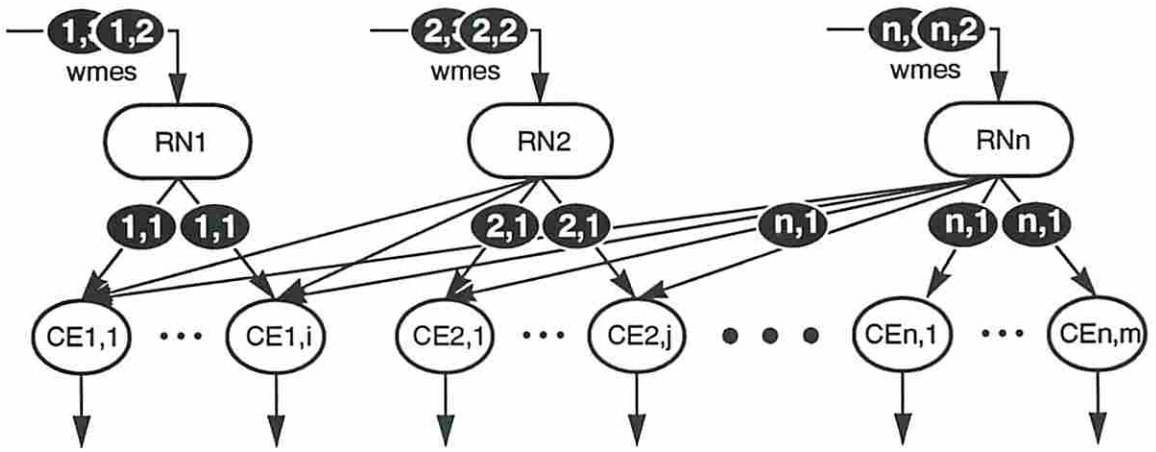


Figure 3.2: An MRN network. RNn distributes wmes to CEs under RN1 through RNn. A wme (i,j) refers to a wme with i AVPs, where j signifies its arrival order. The MRN network also demonstrates a parallel distribution of wmes, where n RNs can simultaneously distribute n different wmes to the network.

Constructing an MRN network is straightforward. All LHSs are split into condition elements (CEs). All CEs are grouped based on the number of AVPs in a CE, i.e., a CE with n AVPs belongs to a group n. Associated with each group is a root node which distributes a set of wmes to a particular group of CEs of the MRN network. For example, RN2 of Figure 3.2 distributes wmes with 2 AVPs to those CEs, where each CE has not more than 2 AVPs. A simple algebra would clarify the obvious advantage of using the MRN network.

Suppose that the network has n groups, each of which has equally m CEs, i.e., the total number of CEs is nm. Assuming that the number of wmes generated due to a rule firing in each production cycle is constant, i.e, k, then the original Rete network will need nmk distribution. Assuming that k wmes are equally distributed over the n groups, i.e., k/n wmes per each group, the MRN network will only need $(1+2+...+n)mk/n$ distributions. For an equal distribution of wmes over the groups, the MRN approach is guaranteed to yield at least a 2-fold speedup over the conventional Rete network. In the remainder of this chapter, this claim will be substantiated using various production system programs.

3.2.2 Allocation of productions

Productions are partitioned into LHSs and RHSs. LHSs are further partitioned into patterns. All patterns are logically grouped together according to the number of attribute-value pairs (AVPs) in the patterns. There are two ways of allocating productions onto the PEs: redundant- and minimum allocations. The first one, redundant allocation, does not follow the structural similarity of the Rete algorithm. 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 PEs. 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 PEs busy. At the same time storage usage can be substan-

tially reduced. However, this will increase overhead in inter-processor communication.

In the data-flow multiprocessor environment, the redundant allocation strategy is chosen as a production allocation policy. A major reason for adopting this policy is in the fact that the runtime communication overhead is much more expensive than simply using more memory space.

Depending on the number of groups in all condition elements, the processor space is partitioned logically into a two-dimensional array PE[i,j], where *i* is a group number and *j* is a PE identification number within the group. Allocating one-input nodes is straightforward. A condition that has *i* AVPs(or OINs) is allocated to PE[i,j], where $j \geq 0$. There are however many factors affecting the allocation 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 at condition level. In what follows, we informally describe our allocation algorithm:

```
Procedure Allocate_Condition_Elements_to_PEs
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 AVPs in CONDITION[j] of RULE[i];
6.          n ← USED[NAVP[i,j]]              ;A number of PEs used in GROUP[i]
7.          For k=1 to NAVP[i,j] do
8.              PE[NAVP[i,j],n] ← OIN[i,j,k]  ;one-input node allocation
9.          PE[NAVP[i,j],n] ← MEM[i,j]        ;memory allocation
10.         USED[NAVP[i,j]] ← n+1;
```

Terminal nodes are not explicitly allocated to PEs 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. To illustrate the above allocation policy, consider

the following production memory of three rules and the MRN network constructed for the three rules (Figure 3.3):

Rule 1

[(a Z) (b Y)]
[(c X) (d Y)]
–[(p 1) (q 2) (r X)]
→
[Modify (c Y) (d X)]

Rule 2

[(p 1) (q 2) (r X)]
[(c X) (d W)]
[(l 5) (m 6) (n W) (o Z)]
→
[Remove 1]

Rule 3

[(c X) (d W)]
–[(a *) (c 5) (e 7) (f W)]
→
[Make (p 1) (q 2) (r X)]

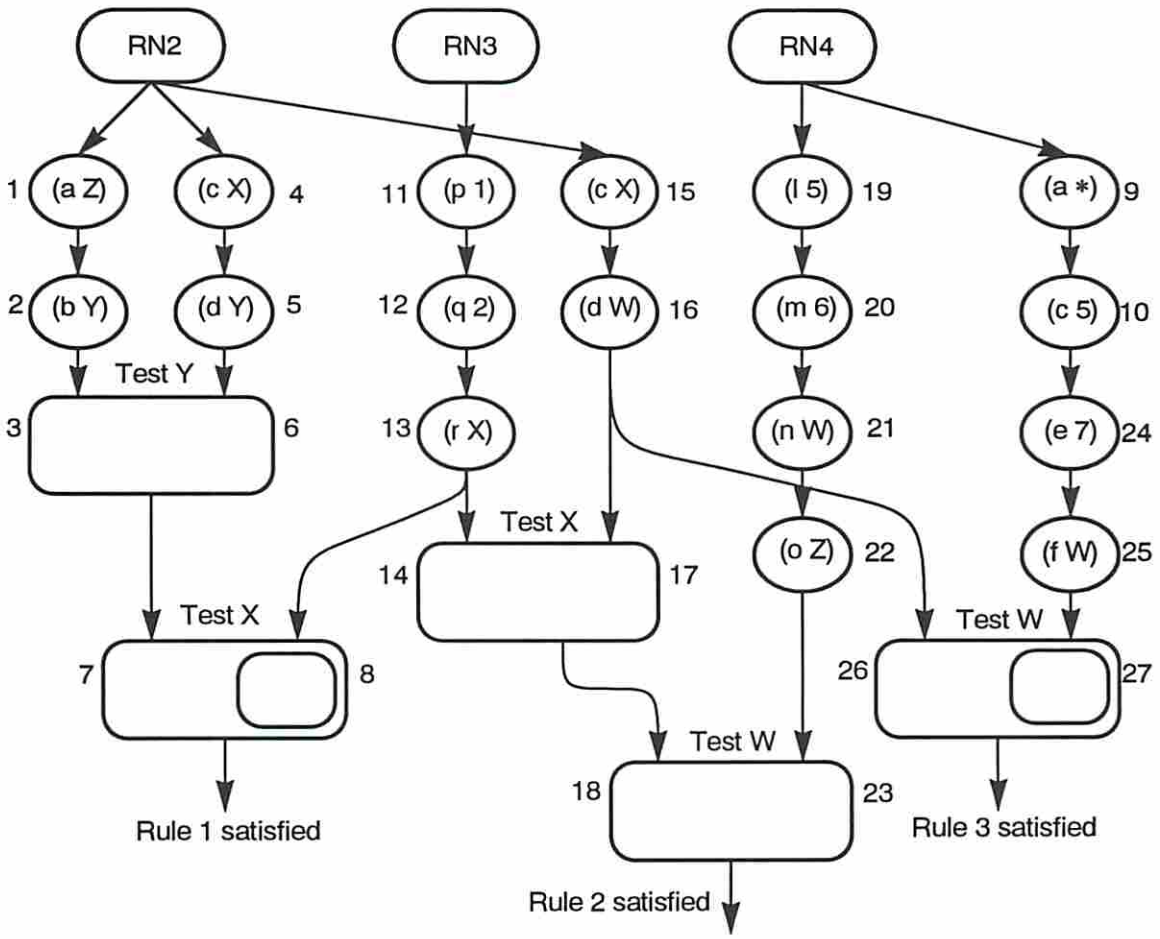


Figure 3.3: An MRN network for the three rules.

Note that Rules 1 and 3 are designed to demonstrate the performance of the negated two-input node. Ellipses in Figure 3.3 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. Note also that for the simplicity of presentation, OINs 11-13 are shared by NTIN8 & TIN14, and OINs 15 &16 by TINs 17 & 26. The actual implementation does *not* share the similar nodes to avoid the overhead in inter-processor communication. Based on the above allocation policy, the network is allocated to PEs, shown in Figure 3.4.

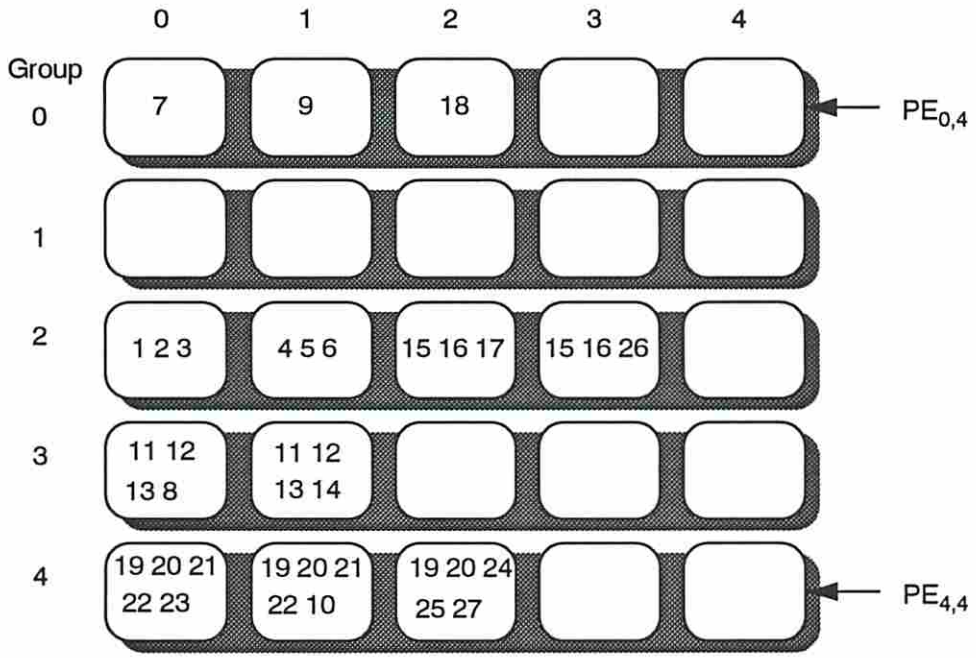


Figure 3.4: A redundant allocation policy. Twelve PEs are used to allocate the three productions. CEs with *n* AVPs are allocated to PEs of Group-*n*.

PEs are partitioned into 5 different groups, where PEs in Group *n* contains patterns having n AVPs. Group 1 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 Figure 3.4 (11 through 13 are one-input nodes). Since the pattern has 3 AVPs, it is classified into Group 3 and allocated to PE1 of Group 3, denoted by PE3,1. The second pattern of Rule 2 in Example 2 has 2 AVPs and right mem-

ory, labeled 15 through 17. It is classified into Group 2 and allocated to PE2 of Group 2, denoted by PE2,2.

In the above allocation policy, we observe that the number of PEs needed to allocate productions is proportional to the number of inter-element feature tests in the productions. For example, suppose that a certain system has n productions and that there are on the average m inter-element feature tests per production. For each inter-element feature test, two memories are needed. The number of PEs needed to allocate n productions would then be $2mn$. For the three rules shown above, there are 3 rules and on the average 2 inter-element feature tests. In total, 12 PEs are used, as depicted in Figure 3.4.

3.2.3 Distribution of multiple working memory elements

Although the Rete algorithm is designed to save computation time in matching patterns over wmes, 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 PEs at a time, provided that many wmes are available at the same time for distribution. It is based on the fact that certain wmes eventually fall into PEs in certain group, where they may be matched. wmes that have i AVPs never match patterns that have j AVPs such that $i<j$.

Whenever the new wmes that are generated due to the rule firings become ready for distribution to the network, the PEs perform the following operations:

Procedure *Distribute_wmes_to_PEs*
1. NAVP[n] ← A number of AVPs 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=NAVP[n].

Assume that the three rules listed in the previous section are compiled and allocated to the PEs according to the allocation policy described in section 4.2. Suppose further that a set of wmes shown below is available and is about to be distributed into the network in Figure 3.4 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 6) (o 2)]
wme6: [(a *) (c 5) (e 7) (f 6)]
wme7: [(c *) (d 6)]
wme8: [(c 3) (d +)]
wme9: [(c 3) (d 6)]
wme10: [(a +) (b 6)]
wme11: [(a 2) (b 6)]
wme12: [(c 2) (d =)]

If the Rete algorithm distributes one wme at a time to the network through the root node, it would take 12 time units to distribute them. This is depicted in Figure 3.5, where one wme at a time is sequentially distributed to all PEs.

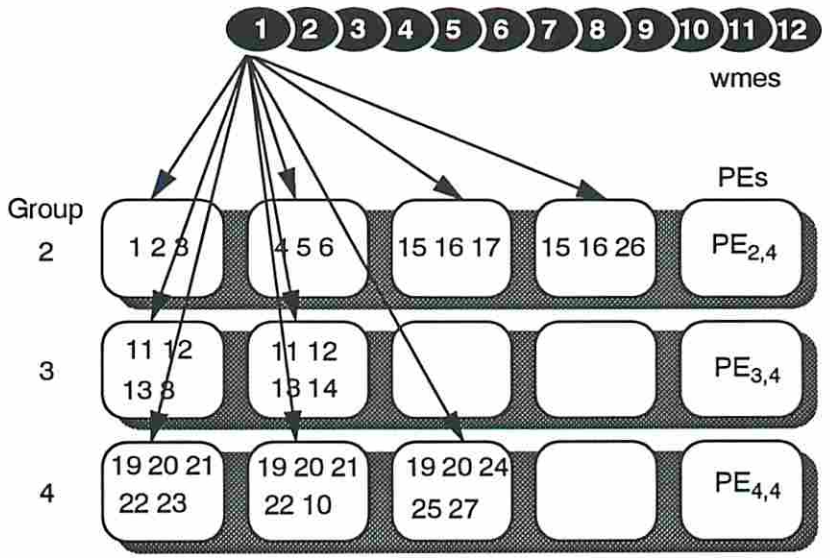


Figure 3.5: Sequential distribution of wmes. Only one wme can be distributed to *all* PEs at a time. To distribute 12 wmes, it takes at least 12 steps (or time units).

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 (= 9PEs x 12wmes). For example, when wme1 is distributed, all 9 PEs to which patterns are allocated make a comparison test in parallel. Only two PEs, 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 wmes are partitioned into 3 groups and the group numbers are assigned to wmes. Wmes 7 through 12 get group #2 while 1 through 3 get #3 and 4 through 6 get #4. The total number of comparison tests performed at the very first one-input nodes in three sequences reduces to 36 (= 6x4 + 3x2 + 3x2), as shown in Figure 3.6. There are three bins in Figure 3.6, where each bin corresponds to a certain group. In each group, wmes are sequentially distributed to PEs belonging to the corresponding group in the PE space. However, between groups wmes are simultaneously distributed.

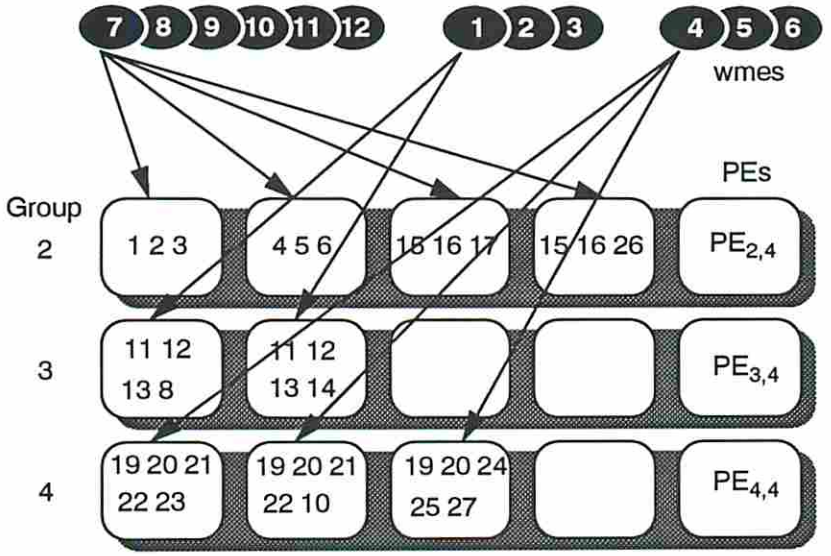


Figure 3.6: Parallel distribution of wmes. Three wmes can be distributed to PEs at a time. To distribute 12 wmes, it takes max{number of wmes in each group}.

If we define 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 the speedup, S=108/36=3 for the given set of wmes. The number of groups in working memory determines the speed-up S. In the worst case, only one wme can be distributed to all PEs at a time as shown in Figure 3.5. 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 wmes will be evenly classified to all groups.

A detailed analysis of the MRN approach will be given in Chapter 6, where several benchmark production systems are implemented on the MRN-based production system interpreter, which was written for the purpose of verifying the performance of the MRN approach. In the mean time, we shall come back to data-flow, and discuss issues related to the implementations of production systems on data-flow multiprocessors.

Chapter 4

Parallel Implementation on a Micro Data-flow Multiprocessor

The MRN-based production system interpreter is implemented on a Micro data-flow multiprocessor. The dynamic data-flow principles are chosen and the MIT Tagged-Token Data-flow Computer is employed as a simulator model. Several assumptions at the hardware level are made to suit the processing of production systems in the simulator. To give a better understanding of the execution of production systems on the target machine, the simple production system with 3 rules given in the previous chapter is used along with a set of 12 wmes. A complete execution sequence is presented in detail. The allocation and distribution policies described in the previous chapter are exercised in the execution. Various runtime statistics are measured to identify the behavior of production systems on the micro data-flow machine. Experiment results are analyzed to compare with experimental results drawn from the conventional von Neumann computers.

4.1 A Simulator Model

A simulation approach has been taken to investigate the performance of the MRN-based match algorithm. A dynamic data-flow multiprocessor, described in Chapter 2, is chosen as a simulator model. Each PE has several simple functional units which can enable the parallel matching of Attribute-Value Pairs (AVPs). 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. Furthermore, the following assumptions are made in the simulation for the sake of evaluation:

- A simulation time unit, τ, is set to 1 μsec.
- Each PE runs at 3 MHz clock $\approx$ 1 μsec/instruction.
- A wme can match a condition element in 1τ.
- The routing time for a token to reach any PE is set to 1τ.
- Each unit in the PE shown in Figure 2.5 takes 1τ.
- Each PE can execute 10 comparison tests at a time.
- The time taken for the I-Structure Controller (ISC) is the same as other units in the PE.
- On the average, there are 3 elements (1 TIN and 1 NTIN) per rule.

Note above that the simulation time units taken for the ISC and other units are equally set to 1. In fact the ISC takes longer than the other units. However, there are other units that take relatively shorter than the ISC, this therefore offsets our original assumption.

Besides the assumptions listed above, three time units are defined which are used in the following example. They are a unit time t, a loop, and an abstract time T. 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 wmes throughout the paper. T denotes an abstract time to demonstrate parallel matching performed at the condition-level and distinguish various events occurring in a high level execution sequence.

4.2 Example

The execution sequence of the MRN-based production system interpreter on a data-flow multiprocessor system is presented. We use the three rules and the 12 wmes shown in Chapter 3, as well as the corresponding network shown in Figure 3.3. 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.

4.2.1 Activities at time T_0: Steps 1-3

Assuming wme1, wme4, and wme7 are simultaneously injected into the network at time T_0, the following steps will take place:

Step 1: wme1 [(p 1) (q 2) (r *)] comes into the network as a data token and is distributed to 2 PEs in GROUP 3 since it has 3 Attribute-Value Pairs. Let us consider an execution sequence in PE3,1.

1. Loop 0 in the PE3,1 for intra-element feature test:
   a) 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.
   b) 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 Figure 3.3 and also Figure 4.1) and opcode (comparison function) are fetched from a program memory (PM) and sent to the ALU.
   c) 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 2.3, we assume that each ALU has several simple functional units to support a parallel execution at the sub-condition level. Note that a variable X is automatically bound to * when the comparisons are successful.
   d) At time t_3, after two one-input nodes are successfully compared in the 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 Figure 2.5) routes the token back to PE3,1 for two-input node operations as it receives it from the TFU.

This top portion is omitted for clarity. See Figure 3.3 for this part.

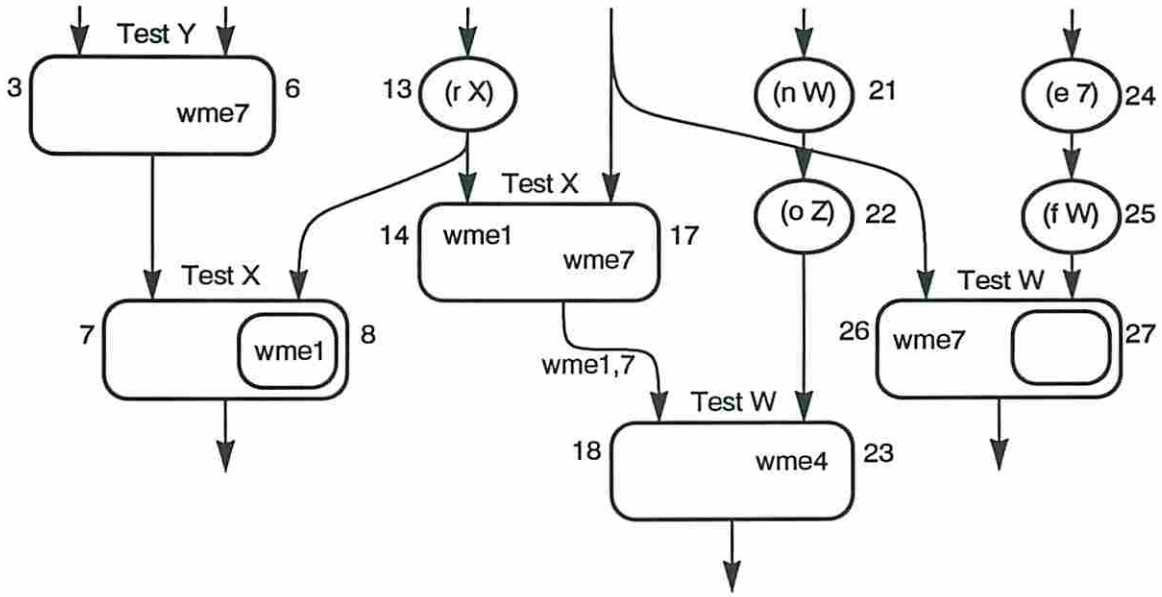


Figure 4.1: Snapshot of the MRN-network after the first match cycle T_0.

2. Loop 1 in the PE3,1 for array operation:
   a) At t_4, the switch in PE3,1 sends the token to MSU.
   b) At t_5, the IFU fetches an append opcode.
   c) At t_6, the wme1 is sent back to the switch in PE3,1.
3. Loop 2 in the PE3,1 for saving wme1.
   a) At t_8, the switch in PE3,1 sends the token to the MSU.
   b) At t_9, the I-Structure Controller (ISC) copies LM14 and appends wme1 to it (shown in Figure 4.1).
4. Loop 3 through 4 in the PE3,1 for inter-element 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, no wme arrived at the MSU of PE3,1. It sends out wme1 to PE2,2.

Step 2: wme7 [(c *) (d 6)] is distributed to 4 PEs in GROUP2 since it has 2 AVPs.

1. Loop 0 through 2 in PE2,2 for intra-element feature test and saving wme7 in RM17 (shown in Figure 4.1).
2. Loop 3 through 5 in PE2,2 for inter-element feature test about X: PE2,2 checks the MSU to see if any wme arrived from PE3,1. As assumed in the step 1.4 the matching operation on wme1 is performed before step 2, so wme1 has been stored in LM14 and sent to the 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 wmes are put together with wme1,7, which is sent to LM18 of PE0,1. See step 4 for next sequence.

Step 3: wme4 [(l 5) (m 6) (n +) (o *)] is distributed to 2 PEs in GROUP4 since it has 4 AVPs.

1. Loop 0 through 2 in PE4,0 for intra-element feature test and saving wme4 in RM23 (shown in Figure 4.1).
2. Loop 3 and 4 in PE4,0 for inter-element 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 wme4 to PE0,1.

4.2.2 Activities at time T_1: Steps 4-7

Step 4: wme1,7 is received by PE0,1.

1. Loop 0 and 1 in PE0,1 for saving wme1,7 in LM18 (shown in Figure 4.2).
2. Loop 2 through 4 in PE0,1 for a second inter-element feature test: PE0,1 checks the MSU and finds wme4, which has been sent from step 3. The values of attribute 'd' in wme1,7 of LM18 and 'n' in wme4 of MSU are compared. The test fails due to the inconsistent variable instantiations. The values of W in wme4 and wme1,7 are respectively '+' and '6' and certainly different. It then sends out wme1,7 to PE4,0. See step 7.

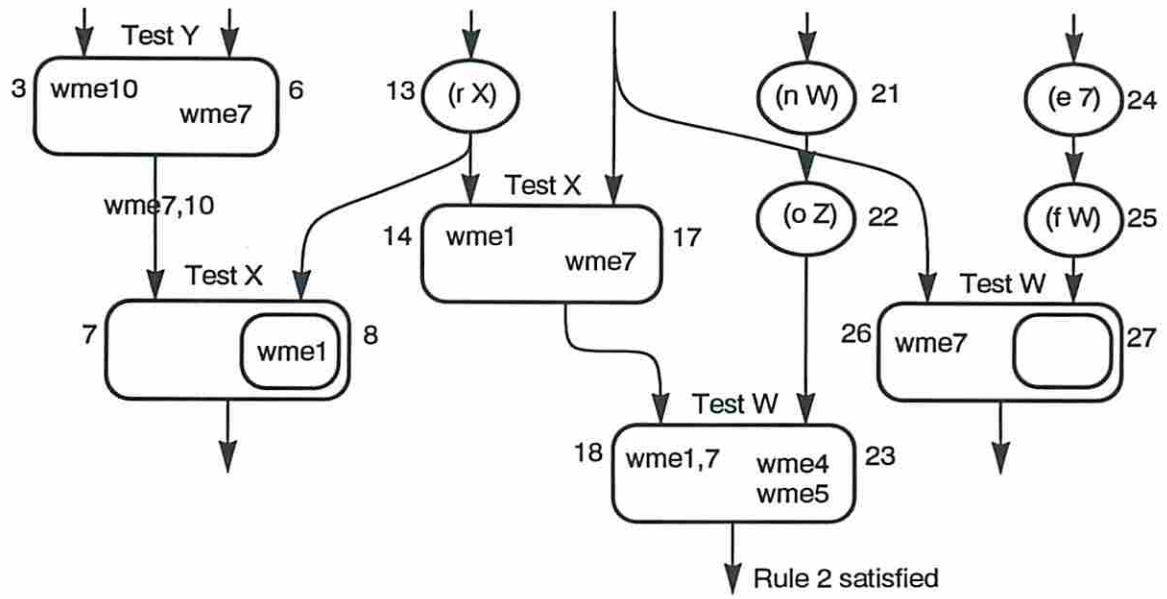


Figure 4.2: Snapshot of the MRN-network after the second match cycle T_1.

Step 5: wme10 [(a +) (b 6)] is distributed to 4 PEs in GROUP2.

1. Loop 0 through 2 in PE2,0 for intra-element feature test and saving wme10 in RM3 (shown in Figure 4.2).
2. Loop 3 through 5 in PE2,0 for inter-element feature test about Y: PE2,0 checks the MSU and finds wme7, which has been received from step 2. The values of attribute 'b' in wme10 of LM3 and d in wme7 in MSU are compared and found successful. wme10 and wme7 are put together to wme7,10, which is sent to LM7 of PE0,0. See step 8 for next sequence.

Step 6: wme3 [(p 1) (q +) (r 3)] is distributed to the 2 PEs in GROUP 3. No PE succeeds in matching since built-in operand [(p 1) (q 2) (r X)] and wme3 are different.

Step 7: wme5 [(l 5) (m 6) (n 6) (o 2)] is distributed to 3 PEs in GROUP4.

1. Loop 0 through 2 in PE4,0 for intra-element feature test and saving wme5 in LM23.
2. Loop 3 and 5 in PE4,0 for the second inter-element feature test about W: PE4,0 checks the MSU and finds wme1,7, which has been sent from step 4. The values of attribute 'd' in wme1,7 of MSU and 'n' in wme5 of RM23 are compared and found equal.

3. Loop 6 in PE4,0 for rule instantiation: wme1,7 and wme5 are put together into wme1,5,7, 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'.

4.2.3 Activities at time T_2: Steps 8-11

Step 8: wme7,10 is received by PE0,0.

1. Loop 0 and 1 in PE0,0 for saving wme7,10 in LM7 (shown in Figure 4.3).
2. Loop 2 and 3 in PE0,0 for a second inter-element 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. wme7,10 is then routed to PE3,0. See step 9 for next sequence.

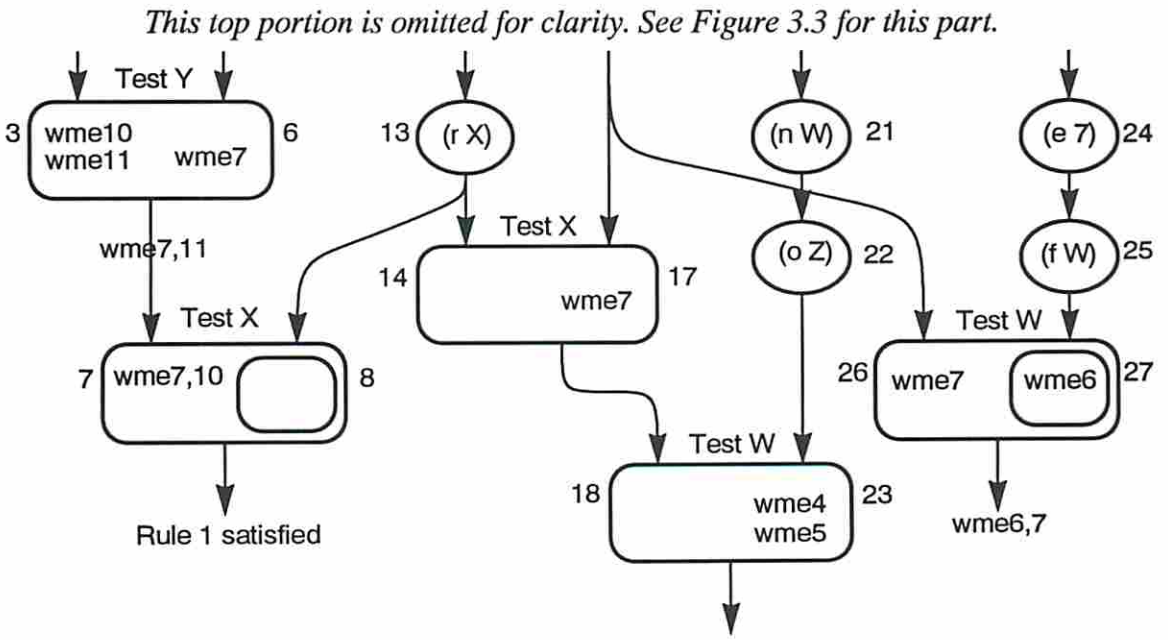


Figure 4.3: Snapshot of the MRN-network after the third match cycle T_2.

Step 9: Assume that the conflict is resolved. Rule 2 fires and -wme1 is distributed to PE3,0 and PE3,1 through RN3 since -wme1 has 3 AVPs.

1. Loop 0 in PE3,0 for intra-element feature test: Nodes 11 through 13 are executed on -wme1.

2. Loop 1 through 3 in PE3,0 for memory examination: Recall that PE3,0 contains a negated element. PE3,0 checks if wme1 exists in RM8 and selects wme1 from the RM8.
3. Loop 4 and 5 in PE3,0 for counter manipulation: Assume that there is only one wme1 in RM8, as is the case, and that the counter attached to wme1 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.
4. Loop 6 and 7 in PE3,0 for second inter-element feature test about variable X: It checks the MSU and determines if any wme arrived from PE0,0. As we assumed in Step 8.2, 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.

Step 10: wme11, [(a 2) (b 6)] is distributed to 4 PEs in GROUP2 and goes through the intra-element 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. T_o check an inter-element 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 Figure 4.3).

Step 11: wme6 [(a *) (c 5) (e 7) (f *)] is distributed to 3 PEs in GROUP4 and goes through an intra-element 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 element. No inter-element feature test about W is necessary.

4.3 Analysis of the Example

The condition-level parallel matching was demonstrated in step 1, 2, and 3, where the dynamic parallel distribution of wmes is made. Steps 5, 6, 7, and 8 as well as steps 9, 10, 11, and 12 also show

the condition-level parallel matching. Negated-element handling and deleting wmes 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 PEs, all the cross-checking activities for the inter-element features are distributed throughout the system so that memory operations are performed asynchronously.

Table 4.1 summarizes the above 11 steps. T, A, C, and R in Table 4.1 stand respectively for a test of equivalence, an array operation, a check of arrival, and routing. Each operation takes 1 loop except A, which takes 2 loops.

Step	Executed at	No. of loops	Type of operations	PEs involved
1	T_0	5	T, A, C, R	pe3,0, pe3,1
2	T_0	6	T, A, C, T, R	pe2,0 thru pe2,3
3	T_0	5	T, A, C, R	pe4,0 thru pe4,2
4	T_1	4	A, C, T, R	pe0,1
5	T_1	6	T, A, C, T, R	pe2,0 thru pe2,3
6	T_1	1	T	pe3,0, pe3,1
7	T_1	7	T, A, C, T, R	pe4,0 thru pe4,2
8	T_2	4	A, C, R	pe0,0
9	T_2	8	T, T, A, T, T, T, R	pe3,0, pe3,1
10	T_2	6	T, A, C, T, R	pe2,0 thru pe2,3
11	T_2	6	T, A, T, T	pe4,0 thru pe4,2

Table 4.1: Summary of the 11 steps in the example

From the above example, we identify the following six observations, each of which performs the typical operation in the Rete algorithm. Each operation is expressed in terms of number of loops. For observations 2 and 3, we assume that there is only one wme in any memory.

1. T_o, intra-element feature test (a set of OINs) by 1 PE, = 1 (T from any step)

2. T_t, inter-element feature test (TIN) by 2 PEs, = 4 (A + C + T from step 2)

3. T_n, negated-element test (NTIN) by 2 PEs, = 6 (C + A + T + T + C from step 9)

4. T_a, adding a wme to a memory, = 4 (T + A + C from step 1)

5. T_d, deleting a wme from a memory, = 6 (T + C + A + T + T from step 9)

6. T_r, routing a token from a PE to a PE, = 1 (R from any step)

Furthermore, T_{r2}, a number of loops executed to instantiate Rule 2, can be approximated as a summation of max{step1, step2, step3} and max{step4, step6, step7}. Using Table 4.1, we find T_{r2} = 6 + 7 = 13. By the same token, $T_{r1} = T_{r2}$ + max{step8, step9, step10} = 13 + 8 = 21. Based on the above observations, we identify below the results that are to be compared with the simulation results in the following section:

1. R_1, ratio of processing time of TIN to OIN, = T_t/T_o = 4/1 = 4

2. R_2, ratio of processing time of NTIN to TIN, = T_n/T_t = 6/4 = 1.5

3. R_3, ratio of processing time of Routing to min{T_t, T_n}, = $T_r/T_t \leq 1/4 = 0.25$

4. R_{r1}, ratio of instantiating Rule 1 to OIN, = T_{r1}/T_o = 21/1 = 21

5. R_{r2}, ratio of instantiating Rule 2 to OIN, = T_{r2}/T_o = 13/1 = 13

4.4 Simulation and Performance Evaluation

A simulation approach has been taken to investigate the performance of the MRN-based match algorithm in a data-flow processing environment.

4.4.1 One-input nodes and array operations

In this simulation, a set of 12 wmes and three rules (shown in Chapter 3) are used. Rule 2 has been converted into a data-flow graph. Figure 4.4 shows the first condition element of

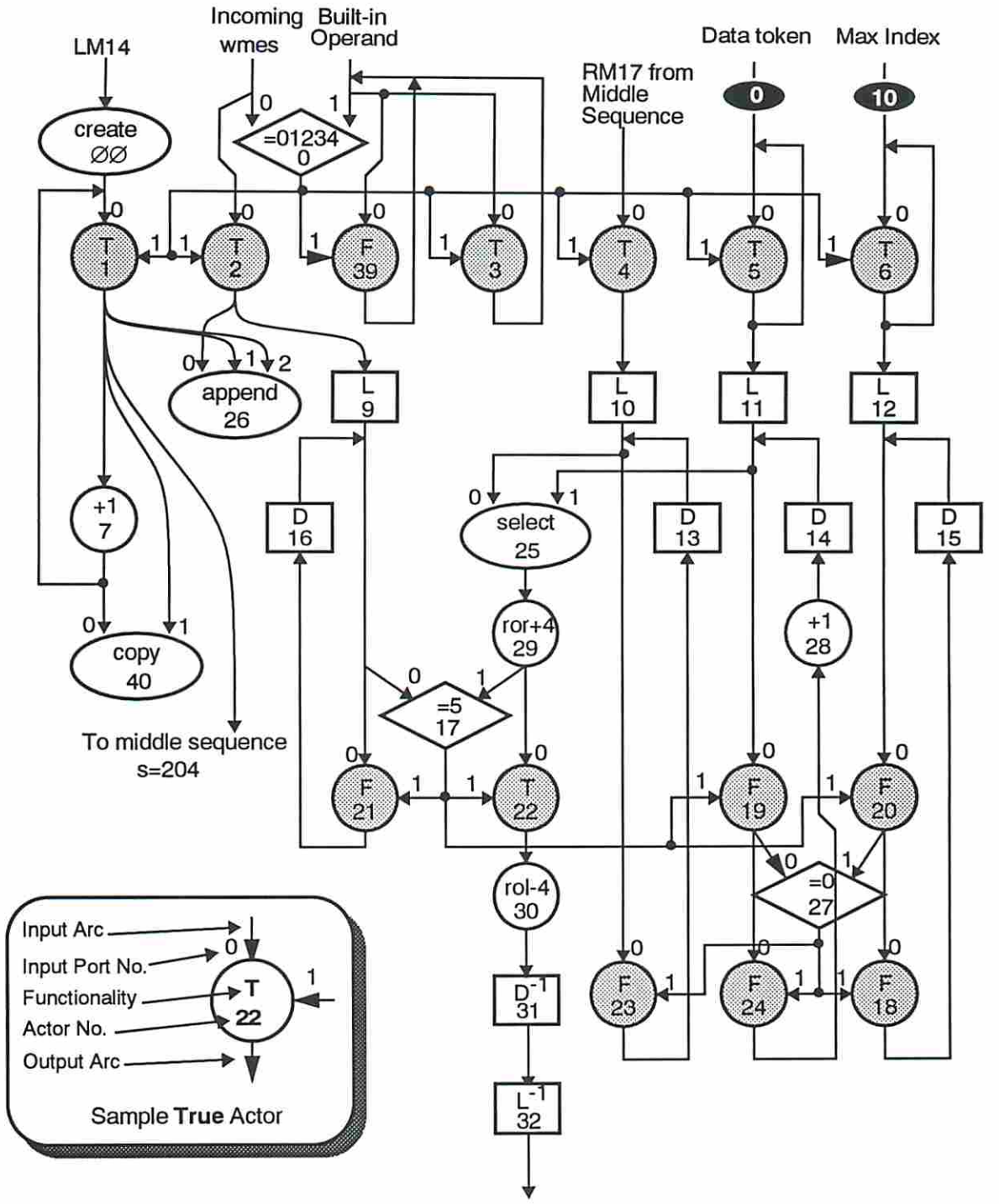


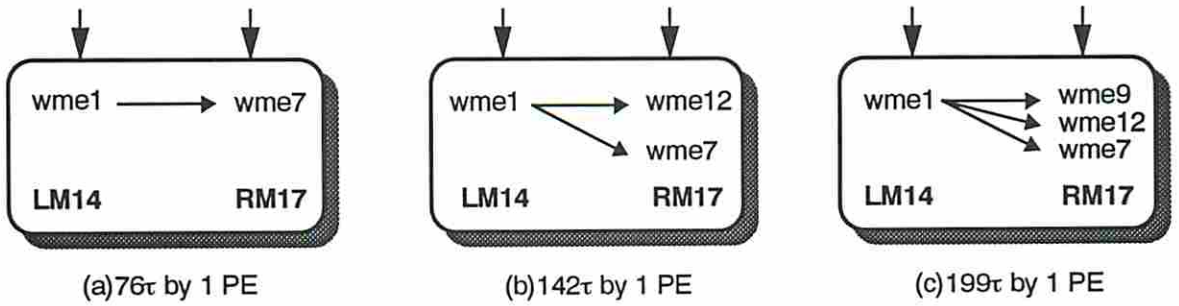
Figure 4.4: A data-flow graph for nodes 11 through 14 of Rule 2. 'ror+4' is to rotate right 4 times. 'rol-4' is to rotate left 4 times.

Rule 2, i.e., nodes 11 through 14 of Figure 3.3. One-input nodes 11 through 13 of Figure 3.3 for intra-element feature tests are implemented through decision actors labeled 0, 2, 3, and 39 in Figure 4.4. All others in Figure 4.4 are for the two-input node 14 of Figure 3.3.

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 4.2. Note that the memories are allocated to the same PE where the OINs are. Table 4.2 shows a sequence of one-input nodes takes about 15 time units, or 15τ, using one PE. Each additional matching takes 13τ.

Trial	No. of wmes	OINs only	Append & Select
1	1	17	29
2	2	0	47
3	3	43	61

Table 4.2: Simulation time units for one-input nodes and array operations



(a)76τ by 1 PE (b)142τ by 1 PE (c)199τ by 1 PE

Figure 4.5: Simulation results by 1 PE for independent two-input nodes processing.

4.4.2 Independent execution of two-input nodes

Three conditions are tested separately one at a time. The condition element1, [(p 1) (q 2) (r X)], is matched to a set of wmes with variations in the order (see Figure 4.5). wmes 1 and 2 are injected

into the left sequence of Rule 2 assuming that the RM17 is filled with wmes 7, 8, 9, and 12 that have been matched with the middle sequence, [(c X) (d W)], of the same rule. Table 4.3 summarizes the simulation time. Trial 1, shown in Figure 4.5(b), indicates wme1 matches against wme7 of RM17 which results in 76τ. Notice in trial 5 that when no match occurs, i.e., when wme2 is placed into the network, the simulation time becomes 286τ due to an exhaustive search in the RM17.

Trial	Input wme	Order of RM17	Simulation Time
1	wme1	7 8 9 12	76
2	wme1	12 7 8 9	142
3	wme1	9 12 7 8	199
4	wme1	8 9 12 7	260
5	wme2	8 9 12 7	286

Table 4.3: Matching CE1 with RM17 of Rule 2 by 1 PE

4.4.3 Parallel execution of two-input nodes

To identify the behavior of the parallel execution, two condition elements are executed in parallel, as depicted in Figure 4.6. It takes about 200-500τ depending upon the number of wmes that have reached either LM14 or RM17 of the two-input node in Figure 4.6. Table 4.4 summarizes the results with various wmes coming into the network.

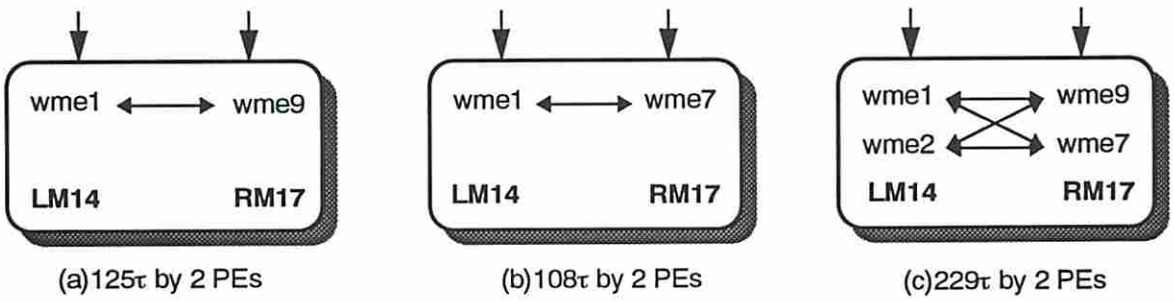


Figure 4.6: Simulation results by 2 PEs for parallel two-input node processing.

The first two columns in Table 4.4 show the wmes 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 wmes that will never match wmes that come from the other sequence whereas O's represent wmes that will match those from the other sequence. For example, the 1st row with X and X shows that 1 wme is distributed to each element, depicted in Figure 4.6(a) and there is no match. The 7th row with X O O and X X O shows that there are 3 wmes distributed to each element and that there are 2 matches.

Trial	Incoming wmes falling into		Simulation Time	
Number	CE1	CE2	1 PE	2 PEs
1	X	X	166	125
2	O	O	130	108
3	O O	O O	207	146
4	X X	X X	379	243
5	O X	X O	327	229
6	O X O	X O	396	256
7	X O O	X X O	521	337
8	X O X	X O X	585	374

Table 4.4: Parallel execution of CE1 and CE2 of Rule 2

4.4.4 Performance evaluation

With the simulation results and assumptions listed above the following results are identified:

1. T_o, the simulation time units for a PE to process one-input nodes and variable bindings with one wme, is 17τ and 13τ for an additional wme (see Table 4.2).
2. T_t, the time units for a PE to process a two-input node with one wme, is 76τ and 50τ for an additional wme (see Table 4.3 and Figure 4.5). This fact validates our approximation made earlier in section 4.2, where R_1, the ratio of T_t to T_o is 4 since $T_t/T_o = 76/17 \approx 4$.
3. Executing a two-input node with various wmes takes 125τ, as shown in Table 4.4 and Fig-

ure 4.6. The 2 PEs to which 2 elements are allocated simultaneously match wmes that are randomly coming into the 2 elements. This fact again validates R_1 being 4. Since this test is done by 2 PEs, $R_1 = (125/2)/17 \approx 4$.

We now calculate the time units for negated-element processing as follows: Given $R_2 = T_n/T_t = 1.5$, $R_3 = T_r/T_t = 0.25$, $R_{r1} = T_{r1}/T_o = 21$, $R_{r2} = T_{r2}/T_o = 13$, $T_o = 17\tau$ (simulation result shown in Table 4.2), and $T_t = 125\tau$ (simulation result shown in Table 4.4 and Figure 4.6(a)). We find $T_n = R_2 T_t = 1.5 \times 125 = 188 \approx 200\tau$. When the routing time T_r is considered, we now find $T_t = T_t(1 + R_3) = 125(1 + 0.25) = 156$ and $T_n = T_n(1 + R_3) = 200(1 + 0.25) = 250$. The time units to process either NTIN or TIN by 2 PEs is, therefore, not more than 300. Note that the approximation for T_n is based on the simulation result in Figure 4.6(a), where there is only two wmes, one in each memory of the two-input node.

T_{r1}, the time taken to process Rule 1 that has one regular TIN and one NTIN, is therefore approximately $T_t + T_n = 156 + 250 \approx 400\tau$. In fact, in our earlier discussion in section 4.2, we approximated $T_{r1} = 21$ loops and this approximation is validated as follows: Given $R_{r1} = 21$, $T_o = 17\tau$, $T_{r1} = R_{r1} T_o = 17 \times 21 \approx 400\tau$. For the Rule 2 that has 2 TINs, $T_{r2} = 2 T_t = 2 \times 156 \approx 300\tau$.

Suppose that a certain production system has rules with average number of two inter-element features (1 two-input node and 1 negated two-input node) per rule and that there is only one wme matched through the one-input nodes and stored in each memory. The data-flow model would instantiate a rule in 400τ, which is equivalent to 0.4 msec. If there are more than 1 wme matched through one-input nodes and stored in each memory, T_r, the time taken to fire a rule will be proportional to the number of wmes stored in each memory [45], as verified by our simulation results shown in Table 4.4 and Figure 4.6(a) and (c). When there are on the average n wmes in each memory, $T_r \approx 400n = 0.4n$ msec in the absence of conflict resolution.

When the conflict resolution step (about 10% of total computation time [16]) is taken into ac-

count, $T_r = 0.4(1 + 10/90)n \approx 0.5n$ msec, where n is an average number of wmes stored in a memory. This T_r in turn gives $1000/0.5n = 2000/n$ rule firings/second. Compared to the analysis of the implementation of OPS5 onto DADO [23], the choice of a data-flow multiprocessor gives a $2000/100n = 20/n$ fold in speed-up since DADO is estimated to be able to fire below 100 rules/second.

4.5 Summary

The potential of data-flow multiprocessor systems for the efficient implementation of non-numeric computations has been investigated in this chapter. Among the various data-flow architectures proposed, the dynamic principles have been chosen since they provide maximum parallelism in the problem domain. We have identified modifications to the implementation of the basic data-flow principles of execution before these can be used in an AI computation environment.

The Rete algorithm has been chosen as a benchmark of symbolic computations on a data-flow multiprocessor because it is the most widely used match algorithm. Inefficiencies in the implementation of the Rete algorithm on parallel machines have been identified and possible solutions to the problems have been worked out in our data-flow environment. Simultaneous distribution of many wmes to many PEs and allocation of conditions and $O(n)$ iterations to different PEs have proven effective in delivering the parallelism inherent to the Rete algorithm and allowed by a given configuration of our data-flow architecture.

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 for execution 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 algorithms 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.

Chapter 5

Parallel Implementation on a Macro Data-flow Multiprocessor

The applicability of data-flow principles of execution to matching operations for production systems has been presented in the previous chapter. In this chapter, we further explore the applicability of data-flow principles of execution to production systems. It has been our observation that AI problems exhibit a behavior characteristically different from conventional numeric computations. We demonstrate in this paper that a macro actor/token approach will best match these characteristics. Section 3 describes those characteristics of production systems from the parallel processing perspective, which we optimize by the utilization of macro data-flow principles. Characteristics of the production system paradigm are identified, based on which we introduce the concept of *macro tokens* as a companion to macro actors. A brief analysis along with simulation results is presented to show *why* medium grain macro actors are preferred to fine grain micro actors. Section 4 discusses several strategies about *how* to derive well-formed macro actors from micro actors for production systems. A set of guidelines is identified in the context of production systems to derive well-formed macro actors from primitive micro actors.Parallel pattern matching is written in macro actors/tokens to be executed on our Macro Data-flow simulator. Section 5 gives simulation results based on our execution model, the macro data-flow simulator, as well as performance evaluation. Simulation results demonstrate that the macro approach can be an efficient implementation of production systems.

5.1 Production System Processing in Macro Data-flow

The basic macro data-flow principle is presented and its implications on production system processing are identified from the macro perspective.

5.1.1 The macro data-flow principle

A macro actor is a collection of scalar instructions. The objective behind lumping instructions into one larger unit is to improve performance by exploiting locality within these larger units. Figure 5.3(b) shows a typical macro actor with a number of micro actors within it. In fine grain data-flow computation, high overhead needed to respect the functionality in execution will result in poor performance at low levels of parallelism. Indeed, to execute fine grain graphs (where each actor represents a single instruction), execution, communication, and computation overhead must be enforced to associate with actual computation actors to insure correct execution. Therefore, overhead problems will be inevitably created and will degrade performance.

When the grain size of a graph is increased, an actor now represents several operations instead of one single operation. This can allow the overhead problem to be alleviated. The concept of macro-actors (several operations are grouped into a single actor) described by Gaudiot and Ercegovac [18], has been shown to bring a solution to the problems of a fine grain computation model. Indeed, with actors of various sizes, the amount of non-compute operations and the cost of communication can be significantly reduced. However, one should also note that the increasing size of actors (with larger granularity) may reduce the available parallelism in programs and increase memory and communication latency. Hence, forming macro-actors from fine grain micro actors is a trade-off between latency and parallelism.

5.1.2 Macro from an AI processing perspective

The macros, when viewed from the AI processing perspective, preserve the AI paradigm at the high level. The basic data object in AI is a list which is a set of elements. When the semantics of

facts or rules under question are concerned, operations on facts or rules are preferred in the form of lists rather than in the individual elements of the lists since each individual element does not carry useful information.

To see the difference between m*a*cros and m*i*cros, consider for example an assertion such as BELIEVE(X Y), which means X believes the fact Y. This assertion, when implemented for processing, can be represented as a list of three elements (BELIEVE X Y). If we break it into three elements and form three data tokens (BELIEVE), (X), and (Y) as a basic element to operate on, each of these three tokens does not carry useful information. Each data token by itself semantically stands for little but a data token. For the above assertion, all three elements must be merged to give some useful information, i.e., they *collectively* operate to make an assertion.

Furthermore, breaking the assertion into many elements and collecting them back to restore it will unnecessarily complicate processing. It is therefore indispensable for the data-flow principles of execution to process AI problems in the form of a list rather than in an individual token. We would like to have a primitive data token as a collection or list of simple elements to carry more information than the single scalar element allowed in the generic data-flow architecture.

Similarly, a macro token is a collection of primitive data tokens. Consider an assertion IS(X Y). This assertion, when implemented, can be represented as a list of three elements (IS X Y). If we break it into three elements and form three data tokens (IS), (X), and (Y) as basic elements to operate on, each of these three tokens carry little useful information.

When viewed from the architectural perspective, macro actors will substantially reduce the overhead in matching tags of data tokens. When using dynamic data-flow principles [2], tokens carry tags which consist of the context, code block, or instance of a loop to which the token belongs. If the fact (IS X Y) is split into three data tokens and is compared with another three data tokens (IS), (X), and (Z), the tag matching time for three pairs of six data tokens is no less than three time units. However, when the two facts are compared in two lists, the tag matching time is *only* 1!

There are, however, drawbacks in using macros. If the primitive actors are grouped and formed

into macros, the parallelism in fine grain processing will be lost. The grouping must be carefully made so as to avoid inefficient macros. Putting too many micros into a macro will apparently decrease the parallelism in fine grain processing, resulting in degradation of the performance. Forming a macro with too few micros will not give a noticeable improvement in performance.

The production system paradigm which we are considering for our application domain is known to have data parallelism existing in many patterns and wmes. By putting too many micros into a macro, the data parallelism will likely diminish. The formation of macros is heavily dependent on problem domain. There must be a set of guidance criteria for the formation of macros. In section 3 we shall identify several heuristics from the production system paradigm and establish criteria to guide the grouping process, thereby producing efficient and well-formed macros. In the mean time, we will discuss in detail the effectiveness of the macros for pattern matching operations.

5.2 List Comparison Operations on Macro and Micro

To give an intuitive overview of the macro over micro, a simple list of comparisons is given to illustrate the performance of the macro. A complete analysis will be given shortly.

5.2.1 A micro-actor perspective

Consider a typical match operation in OPS5 which compares the following two condition elements: p_1=[P (a X) (b Y)] and p_2=[Q (a 1) (b 2)], where X and Y are variables and all others constants. In order to achieve the *maximum* parallelism in fine grain micro-actor operations, a data-flow graph for the match operation can be drawn much like the one in Figure 5.1. The two lists are split into ten primitive data tokens: P, a, X, b, Y, Q, a, 1, b, and 2. The ten data tokens form five pairs: (P Q), (a a), (X 1), (b b), and (Y 2), each of which would then be allocated to a different PE. Five comparison operations are executed in parallel by five PEs to achieve maximum parallelism existing in the fine grain processing. The results of all three comparison operations are reported to

actors allocated to PE0. As shown in Figure 5.1, a final result of the matching operation is obtained from actor 9 of PE0 after three levels of AND operations.

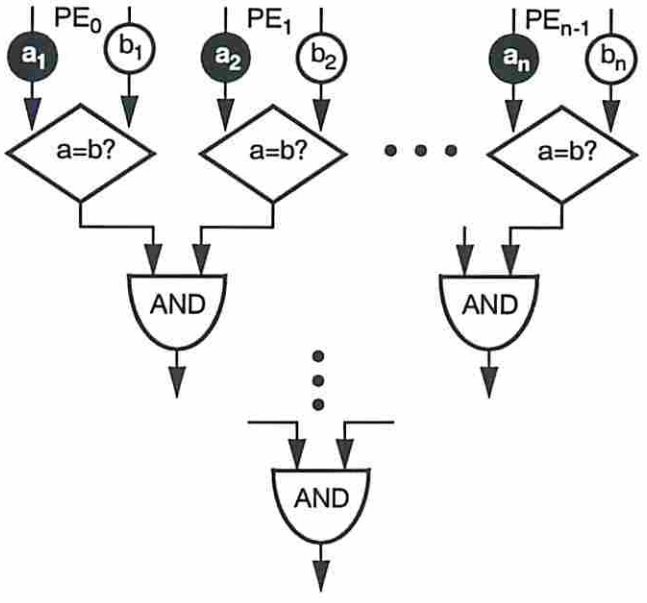


Figure 5.1: A data-flow graph in *micro*-actors for the comparison of two lists.

To further examine the number of operations and the time units for this matching operation, consider a hypothetical data-flow machine consisting of 64 processing elements (PE). Between two neighboring PEs are *three* facilities: two communication nodes and a link. Assume that each PE has *four* facilities connected in a pipe (we shall come back to this in the following section).

Let us briefly define various timing units which we are going to use throughout this paper.

- τ = Processing time for each processing facility in the system,
- t_r = Token routing time to a neighboring PE,
- t_c = Time taken for a *comparison* operation,
- t_a = Time taken for an *and* operation,
- t_{pe} = Time taken for a PE to produce a result token after receiving token(s),

Assuming for a moment that there is no token waiting for dyadic (2 input) operations, we have

$t_c=t_a=t_{pe}=4\tau$ and $t_r=3\tau$. The total time taken to process the matching operation by 5 PEs would then be as follows:

$$t_{micro} = t_c+t_r+t_a+t_r+t_a+t_r+t_a = 4t_{pe} + 3t_r = 4(4\tau) + 3(3\tau) = 25\tau.$$

Now, consider a typical match operation, shown in Figure 5.1, which compares two lists, $(a_1,...,a_n)$ and $(b_1,...,b_n)$. To achieve the maximum parallelism existing in the fine grain micro approach, the n-pairs can be simultaneously compared in n PEs, each of which is connected through a (log n)-dim hypercube. Assume that two neighboring PEs must communicate through three facilities (two communication nodes and a link) and that each PE consists of four facilities connected in a pipeline fashion. If each facility take τ to execute, the total time to process n comparisons on n PEs would be

$$\begin{aligned} t_{micro,n} &= t_c + (\alpha + \lceil \log_2 n \rceil) t_a + (\beta + \lceil \log_2 n \rceil)\, t_r + \gamma \\ &= 4(1+\lceil \log_2 n\tau \rceil)\tau + 3\lceil \log_2 n \rceil \tau \\ &= (4+7\lceil \log_2 n \rceil)\tau \end{aligned} \qquad (5\text{-}1)$$

For the matching operation shown earlier in the section with two lists p_1=[P (a X) (b Y)] and p_2=[Q (a 1) (b 2)], we verify from (Eq. 5-1) that $t_{micro}=25\tau$ by substituting n for 3. Note that in this simple calculation, it is assumed that no token waits in the matching/store unit of each PE to simply show how the list matching can proceed in a data-flow environment. Furthermore, all the comparison actors are *ideally* allocated to neighboring PEs such that a token routing can be done in a step, i.e., 3τ (which may not be realizable).

If any one of the six tokens happened to arrive at the designated PE late, then the results of other two comparison operations would have become useless. AI processing requires five condition-wme pairs to be executed in parallel such that the current state is correctly reflected to knowledge-base. If they were executed at different time, the current state of the knowledge is not correctly reflected and the result of the matching operation is not guaranteed to be correct unless some sort of

synchronization mechanism for the knowledge-base is provided. In such a case the AI problem solving system under consideration will not yield desirable solutions.

5.2.2 A macro-actor/token perspective

The macro approach provides a solution to the above problem of comparison operation. In the macro approach, the basic unit of the data-token operating on data-flow graph can be made in the form of list. It preserves the semantics of wmes and correctly reflects the change of wmes to knowledge-base since both the wme and the condition element are compared as a whole. The number of PEs used in this operation is one and therefore there is *no* communication overhead. The total time taken to process the matching operation in the macro approach is simply proportional to the number of elements in the condition element.

In general, the total time to compare two lists with n elements each on a PE, would be:

$$t_{\text{macro},1} = nt_c+(n-1)t_a = 4(2n-1)\tau \tag{5-2}$$

The ratio of the time taken for macro actors with 1 PE to micro actors with n PEs is

$$R = t_{\text{macro},1}/t_{\text{micro,n}} = 4(2n-1)/(4+7\lceil\log_2 n\rceil) = O(n/\log_2 n). \tag{5-3}$$

Note that in the micro-actor approach, we assumed that the token routing is done in one step, i.e., 3τ. In general, such one-step routing for this kind of matching operation is impractical for a 6-dimension hypercube topology. Because 6-dimensional hypercube interconnection network provides not more than 6 PEs in one step routing distance, to have more than 6 comparisons and routing performed in parallel requires *very* sophisticated allocation and routing policies.

For the match operation in Figure 5.1, where n=5, we obtain $t_{\text{macro}}=36\tau$ from (Eq.5-2). Comparing with $t_{\text{micro}}=25\tau$ the micro actor approach, we lose 11τ. Considering that the average number of elements in a list is five for production systems, the ratio becomes $R \approx 5/(\log 5) \approx 1.7$, as can be verified from (Eq.5-3). The macro actors will outperform since there is *no* communication overhead involved in macro actors.

The analysis described above is intended to show the macro approach and is by no means a complete analysis. More detailed simulation results and their performance evaluation are presented shortly.

5.3 Formation of Macro Actors

In the previous section, we discussed *why* macros are preferred over micros for processing AI problems. We shall now discuss in this section *how* we can obtain well-formed macros from micros for the PS paradigm. More precisely, we shall construct well-formed macros to implement rules.

5.3.1 Guidelines for well-formed macro actors

A macro-actor is a collection of scalar instructions. The objective behind putting many instructions into one is to improve performance by exploiting locality in the instructions. As seen from Figure 5.2, the set of ordered micro actors $A=\{a_1,...,a_n\}$, is converted to a set of ordered macro actors $B=\{b_1,...,b_m\}$, where b_i is a partially ordered set of micro actors $\{a_i,...,a_j\}$ and $m<n$. Well-formed macros (*wfms*) would give a substantial performance improvement as we have seen in the previous section. However, a question immediately arises as to *how to form macros* from a set of micros. Ill-formed macros will not simply give a desired performance enhancement but would potentially degrade the performance.

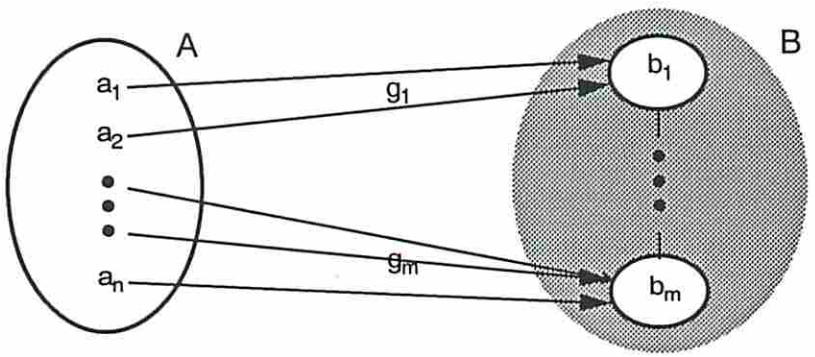


Figure 5.2: Mapping micro actors into macro actors. A set of macro actors, $B=\{b_1,...,b_m\}$, is derived from a set of micro actors, $A=\{a_1,...,a_n\}$, based on the guidance criteria, $g_1,...,g_m$, where $m<n$.

A criterion must be carefully identified to obtain wfms. There can be many criteria we can possibly apply to a set of micro actors to form wfms. Yet, the parallelism existing in the problem domain should be preserved after the formation of macros. Before we discuss the formation of macros, let us give definitions we will use throughout this discussion.

Let A be a set of micro (or primitive) actors, $\{a_1,...,a_n\}$, and T_A be a set of data tokens, $\{t_1,...,t_m\}$, manipulated by A. Let B be a macro actor derived from A such that $B \subseteq A$. Let t_1 be the time taken to process a micro actor on a PE. Let t_n be the time taken to process A on n PEs. Let T_1 be the time taken to process a macro actor on a PE. Let r be a ratio of T_1 to t_n, i.e., $r=T_1/t_n$. A macro actor B is said to be *well-formed* if $r<\varepsilon$, $\varepsilon \leq 2$.

An objective behind setting such a ratio is in the fact that if the processing time of the macro actor is not more than twice the processing time of the corresponding micro actors in an ideal environment, we shall form a macro actor from micro actors. The word "ideal environment" refers to an ideal allocation of micro actors on n PEs and an ideal routing policy of the data tokens. As we discussed earlier in section 3, achieving such an ideal environment would be impossible. The macro actors would be preferred over micro actors because of no data token routing, no waiting for the mating data token (for two operand instructions), etc. In this study, we simply set ε to 2 for macro actor formation. We now briefly describe the formation of well-formed macro actors.

Let I_i be a set of tokens input to a_i and O_i be a set of tokens output from actor a_i. We denote the dependence relation for $a_i,a_j \in A$ as follows: If $O_i \subseteq I_j$ such that $i \neq j$, $a_i \angle a_j$ for all i and j, where $\angle$ is a dependence operator which implies that a_i must be executed before a_j. By applying the dependence relation $a_i \angle a_j$ to A, we obtain an ordered set of actors, $B=\{b_1,...,b_m\}$, where $b_i=\{a \mid a_i \angle a_j$ is not true$\}$. The dependence distance for $b_i,b_j \in B$ is defined as $d(b_i,b_j)=d_{i,j}=i-j$. The maximum dependency distance d_{max} for B is $m-1$.

We list below five guidelines for the formation of *wfms*:

1. (*Flow Dependency*) Let a_i and a_j be two actors. A macro actor $M=\{a_i,a_j\}$ can be defined

if $O_i \subseteq I_j$ and $d_{i,j}=1$, where O_i is an output of a_i, I_i is an input to a_j, and $d_{i,j}$ is a dependence distance between a_i and a_j.

2. (*Encapsulation Effect*) Let a be a comparison actor and b be a set of true/false actors $\{b_1,...,b_n\}$. A macro actor $M=\{a,b\}$ can be defined if $O_a \subseteq I_b$ and $d_{a,b}=1$. This guideline eliminates unnecessary true/false actors.

3. (*List Processing*) Let $A=\{a_1,...,a_n\}$, and L be a list of m data tokens $\{t_1,...,t_m\}$. Let $I=I_1 \cup ... \cup I_n$ and $O = O_1 \cup ... \cup O_n$. A macro actor $M=\{a_1,...,a_n\}$ can be defined if $I_i \subseteq I \cup - O \cup L$ for $1 \le i \le n$ and $O_k \notin I$ for $k=d_{max}$. This guide preserves the semantics of a list.

4. (*Array Operations*) Let $A=\{a_1,...,a_n\}$ and B = {Append, Select, Create, Copy}. If $A \cap B \neq \varnothing$, a macro actor M can be formed on $A-B$. Separating array operations from the macro actors removes the potential bottleneck in array operations.

5. (*Interconnection Topology*) Let $A=\{a_1,...,a_n\}$ such that $d_{max} = \text{Max}\{d_{i,j}\} = 1$ for all $a_i, a_j \in A$. Let m be a dimension of hypercube interconnection network. If $m \le n$, a set of macro actors $\{M_1,...,M_k\}$ is defined where $M_i=\{a_1,...,a_m\}$, $k= \lceil n/m \rceil$, and $1 \le i \le k$.

5.3.2 An example on the conversion process

Using these five guidelines, we shall now write several macro actors to implement a simple rule. The functionality of the rule that are important to implement production systems will be taken into account. Consider the following OPS5-like rule:

```
Rule:   [A (Y Z)]                ;;CE 1
        [B (c X) (d Y)]          ;;CE 2
        [C (p 1) (q 2) (r X)]    ;;CE 3
        →
        [Modify B (c Y) (d X)]   ;;Action 1
```

Suppose that we have a Rete condition-dependency network constructed for the above rule

[22]. Figure 5.3 shows a conversion process for the *first* condition element. A micro data-flow graph for the comparison operations on two elements, [A (Y Z)] and [A (B C)], is depicted in Figure 5.3(a) and the corresponding macro actor in Figure 5.3(b).

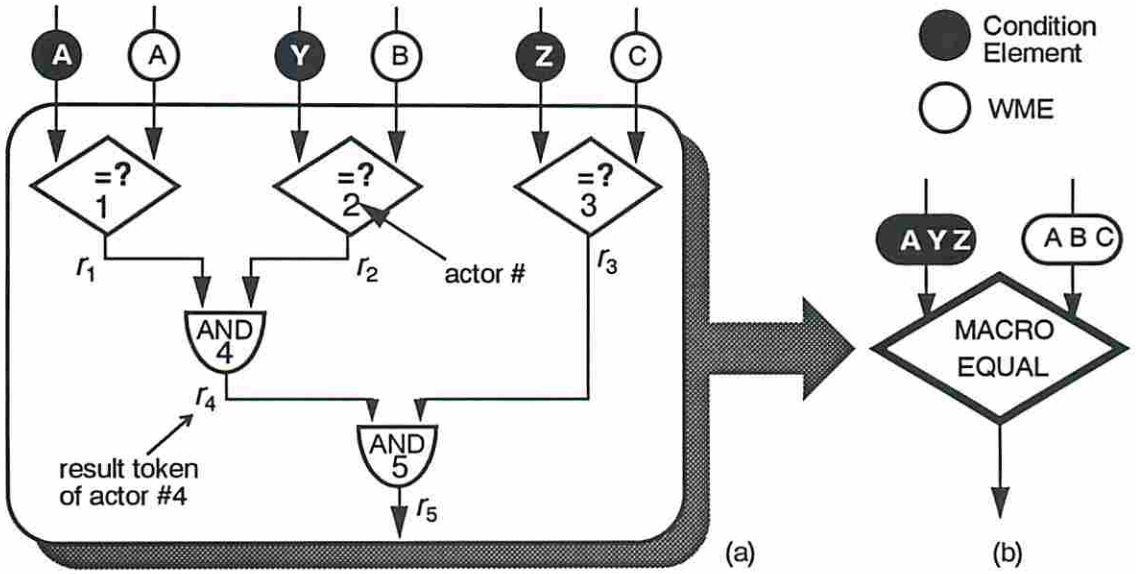


Figure 5.3: A conversion process for the comparison operation on two lists. (a) a micro actor data-flow graph, (b) a macro actor.

Rule *Guide3* is applied to this conversion process as follows: Let A be a set of five actors $\{a_1,...,a_5\}$ (three comparison actors and two AND actors), and L be a list of six data tokens $\{A,Y,Z,-A,B,C\}$. Let r_i be the output token of an actor a_i. Applying a dependency distance to the set A, we partition A into three sets A_1, A_2, and A_3, where $A_1=\{a_1,a_2,a_3\}$, $A_2=\{a_4\}$, and $A_3=\{a_5\}$. We then find $d_{max}=2$ because $\text{Max}\{d_{A1,A2}, d_{A1,A3}, d_{A2,A3}\} = \text{Max}\{1,2,1\} = 2$. We also observe that

$$I = I_1 \cup ... \cup I_5 = \{L,r_1,...,r_5\},\ O=O_1 \cup ... \cup O_5 = \{r_1,...,r_5\} \tag{5-4}$$

From (Eq.5-4), we have $I_i \subseteq I \cup O$ for $1 \le i \le 5$, and $O_{A3}=r_5 \notin I$. Therefore, *Guide3* is satisfied and a macro actor $M = \{a_1,...,a_5\}$ can be formed. After *Guide5* is applied to the data-flow graph for the first condition element of the above rule, we obtain a graph shown in Figure 5.4. Note that those

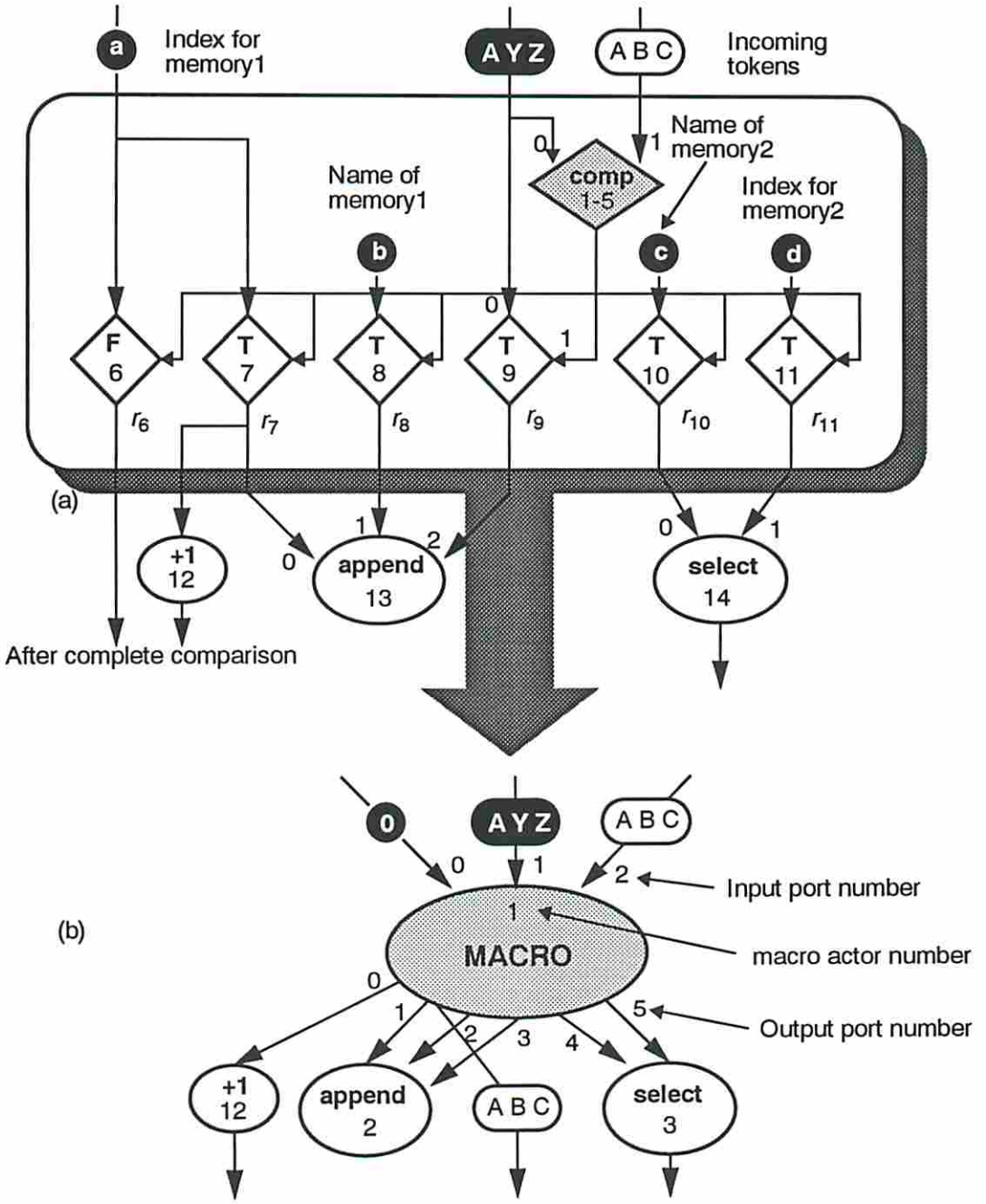


Figure 5.4: Converting a micro-actor data-flow graph to macro actors: (a) micro-actors (b) macro-actors.

five comparison micro actors of Figure 5.3(a) have been replaced by a single actor #1, *comp*, in Figure 5.4(a) for the simplicity of presentation.

In Figure 5.4(a), there are 2 actors related to array operations: 'append' and 'select.' *Guide5* states that if there exists an actor a_i in A such that $a_i \in$ {append,select,...}, then a macro actor M is

considered on the set $A-a_i$. Applying the *Guide5* to the graph partitions it into 3 sets of micro actors A_1, A_2, and A_3, where $A_1=\{a_1,...,a_{12}\}$, $A_2=\{a_{13}\}$, and $A_3=\{a_{14}\}$. In the first step of the conversion process, the set of actors $\{a_1,...,a_5\}$ however is converted to a macro actor M_1. We therefore treat M_1 as a micro actor in the following discussion. Partitioning the graph into three graphs is shown in Figure 5.4(a).

The last rule stems from the fact that there are six true/false actors in A_1 (Figure 5.4(a)). *Guide1* states that if there is a comparison actor A which immediately affects a set of true/false actors B such that $O_A \subseteq I_B$ and $d_{A,B}=1$, then we should form a macro actor $M=\{A,B\}$.

The macro comparison actor M just described above is for the first condition element of the rule shown earlier. The argument discussed above applies to other condition elements and shall not be discussed further. The set of guidelines described above is by no means a complete set. It can, however, serve as a starting point for the formation of *wfms*

5.4 Simulation and Performance Evaluation

A simulation approach has been taken to investigate the performance of the MRN-based match algorithm in a data-flow processing environment.

5.4.1 Simulation results

A simulation approach has been taken to identify the performance of production system processing on the Macro Data-Flow Multiprocessor (MDFM) simulator [70]. The machine contains 64 PEs interconnected by a 6-dim hypercube network. The target production system, which we call a 'generic production system,' has 15 rules, all of which are written in micro actors based on the parallel version of the Rete algorithm [59].

A typical OPS5-like rule was shown in the previous section. Each rule has on the average 5 condition elements, 2 action elements, and 3 two-input nodes. Each condition element has on the average 3 one-input nodes and at least one variable in the value-part (see [16] for details). With the

guidance criteria we developed, the micro actors for the rules are written in macro actors, each of which contains on the average 50 micro actors.

Tables 1 through 3 show simulation time, network load, and speedup. Table 1 lists simulation time units and network load for sequential and parallel distribution of wmes with various number of PEs. Table 2 derives the ratio of sequential distribution (SD) to parallel distribution (PD). Parallel distribution of wmes yields a maximum of 4.4 speedup and reduces a maximum of 2.5 times the network load over sequential distribution of the original Rete algorithm. Regardless of the number of PEs used, parallel distribution provides an average of 2.5 speedup and reduces the network load on the average 2.4 times. Table 3 shows simulation results on speedup of using different number of PEs. Various curves for the simulations results are depicted in Figure 5.5.

Number	SD1		PD1		SD2		PD2	
of PEs	T	L	T	L	T	L	T	L
1	23619	0	8955	0	23544	0	8912	0
2	13269	9510	4589	4017	12161	9432	4840	3792
4	7239	15738	2614	6901	5873	15078	2895	6406
8	5047	22491	1701	9643	3296	21389	1545	8935
16	3519	26568	1423	11841	2101	26822	1038	11101
32	3336	33364	1314	14157	1434	31991	763	12874

Table 5.1: Simulation time, *T*, and network load, *L*, for a production system with 15 rules executed on MDFM. SD = Sequential Distribution, PD= Parallel Distribution.

Number	SD1/PD1		SD1/PD2		SD2/PD1		SD2/PD2	
of PEs	T	L	T	L	T	L	T	L
1	2.6	N/A	2.7	N/A	2.6	N/A	2.6	N/A
2	2.9	2.4	2.7	2.5	2.7	2.3	2.5	2.5
4	2.8	2.3	2.5	2.5	2.2	2.2	2.0	2.4
8	3.0	2.3	3.3	2.5	1.9	2.2	2.1	2.4
16	2.5	2.2	3.4	2.4	1.5	2.3	2.0	2.4
32	2.5	2.4	4.4	2.6	1.1	2.3	1.9	2.5

Table 5.2: Ratio of sequential distribution to parallel distribution.

No of PEs	SD1	PD1	SD2	PD2
1	1.00	1.00	1.00	1.00
2	1.78	1.95	1.94	1.84
4	3.26	3.43	4.00	3.08
8	4.68	5.26	7.14	5.77
16	6.71	6.29	11.21	8.59
32	7.08	6.82	16.42	11.68

Table 5.3: Speedup of a generic production system executed on MDFM.

5.4.2 Performance evaluation

From the simulation results, we verify that: *First*, our parallel network with multiple root nodes reported in [21] gives an impressive improvement over the original sequential Rete network: the number of groups we had among condition elements of our generic production system is 3. The simulation time of the sequential Rete network, regardless of the number of PEs used, is almost always three times that of our parallel network, as seen from Table 1 and Figure 5.6.

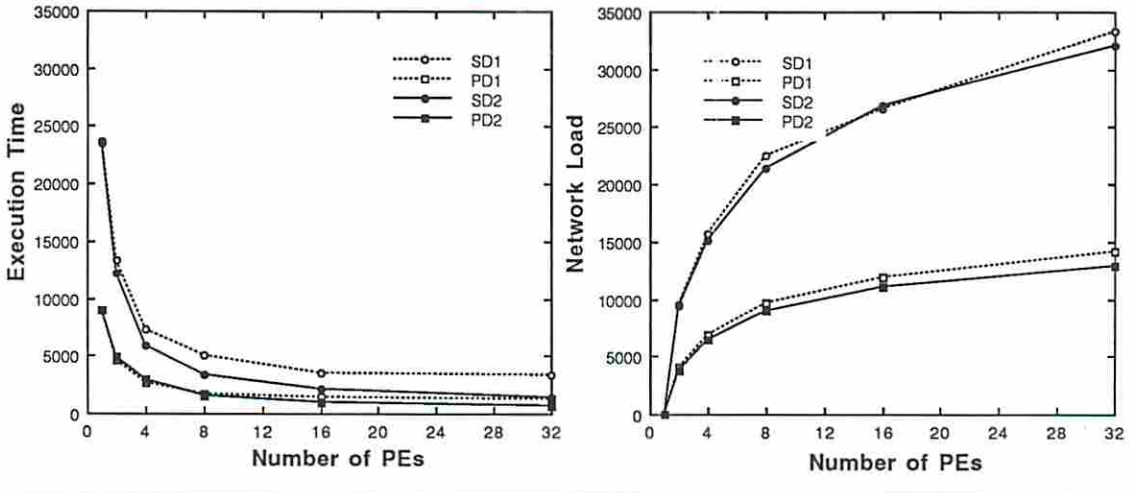


Figure 5.5: Simulation results. (a) execution time, (b) network load.

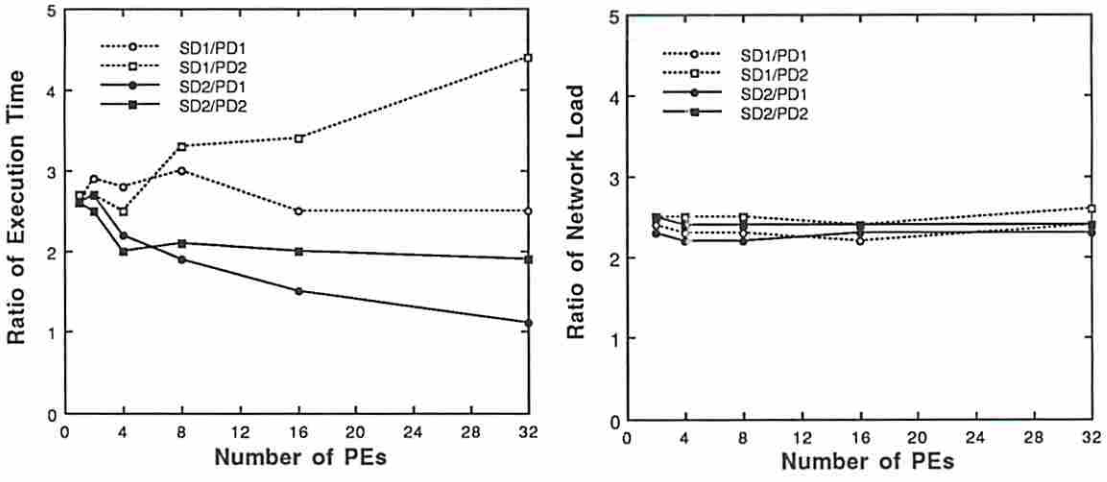


Figure 5.6: Ratio of sequential distribution to parallel distribution. (a) execution time, (b) net work load

Second, the data-flow principles of execution can, not only efficiently perform nonnumeric computation, but also yield an impressive performance over the conventional von Neumann model of execution for production system processing. From the speedup curve of Figure 5.7, we find that data-flow principles of execution can indeed yield a 17-fold speedup when 32 PEs are used, regardless of the type of matching algorithms.

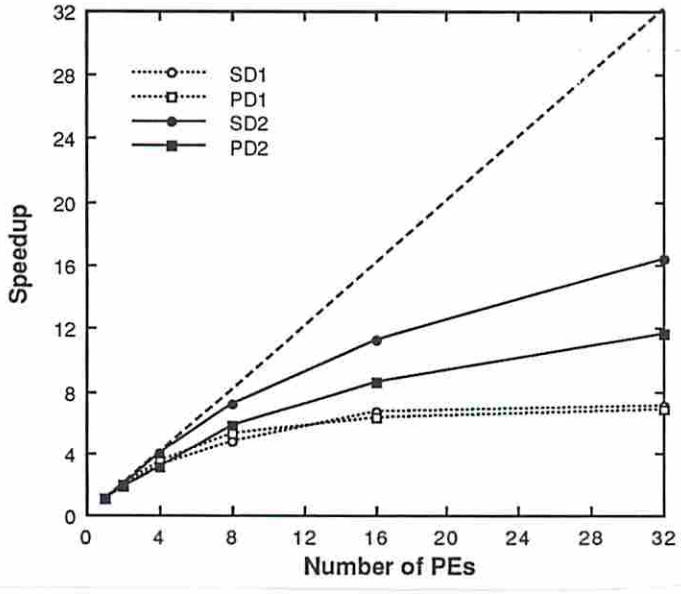


Figure 5.7: Speedup of sequential distribution vs. parallel distribution.

5.5 Summary

A macro actor approach for AI problems, specifically production systems, has been demonstrated as an efficient implementation tool. Characteristics of production systems from parallel processing have been discussed to suit the macro data-flow multiprocessor environment. A simple example on comparison operations has been explained in detail from the macro perspective. Several guidelines have been demonstrated to form *wfms*. A condition element of a rule in PS is converted to macro actors. The results of a deterministic simulation with 15 rules and more than 100 condition and action elements on the macro data-flow simulator have revealed that the macro approach is an efficient implementation for the AI production systems. Indeed, the macro approach gives a 17-fold speedup on 32 PEs. Furthermore, our parallel matching algorithm with multiple root nodes gives an additional speedup of 3, regardless of the machine used. Assessments of the data-flow systems on productions systems have proven effective and we are currently investigating issues related to parallel firing of multiple rules toward true parallel production systems.

Chapter 6

Performance Evaluation of the Multiple Root Node (MRN) Approach

An implementation of the MRN-based production system in Common Lisp is presented and several distinctive features of the MRN-based production system are listed in the first section. Organization of the program as well as data structures used in the program are briefly presented to give a feeling of how the implementation is carried out. Those five benchmark production system programs chosen for this study are introduced in Section 2 along with various statistics gathered at compile time on the five programs. Among the statistics, it is the grouping information that makes the MRN-based production system interpreter different from other production system interpreters.

Section 3 presents various data points obtained at runtime on the benchmark programs. All the statistics are gathered in terms of production cycle numbers. Among the data, an execution time profile of matching one-input nodes is measured in real time in terms of production cycle numbers. To ensure the correctness of the measurement, another important criterion is measured, the number of comparison operations, again in terms of production cycle numbers.

Performance evaluation on the two approaches, the MRN approach and the Rete-based OPS5, are made in Section 4 in terms of number of comparison operations and execution time for one-input match step. The last section, Section 5, summarizes this chapter.

6.1 Implementation of the MRN-based Interpreter

Implementation details of the MRN-based production system are presented in this section. Several distinctive features embedded in the implementation are iterated to show how Lisp systems can be constructed. Data structures as well as the organization of the program are briefly explained to help understand the implementation.

6.1.1 Characteristics of the implementation

The MRN-based production system has been completely implemented in Common Lisp from scratch. It is listed in the Appendix and is currently operational. Its functionality is 100% up to the Rete-based OPS5 production system interpreter. The size of the program is approximately 3,000 lines in Common Lisp. The main features of the implementation are:

- It is free of global variables except a single one which traces the number of wmes generated during the lifetime of a particular production system program,
- Over 90% of the functions are written in tail-recursion,
- A simple data structure using defstruct of Lisp is used.

A major reason to avoid using global variables is in that the program should be easily ported to various multiprocessor environments without having to change much of its source codes. By not using global variables, the potential communication and synchronization overhead between processes would be reduced when ported to a multiprocessor environment. Furthermore, encapsulating the scope of variables within a function would allow us to analyze the data dependency, if any, between functions, thereby resulting in easy program partitioning. The ultimate goal of parallel processing, extracting and exploiting more potential parallelism from given codes, would then become within a reachable distance. This claim yet has to be substantiated and will be left as a future research topic.

Much effort has been spent on writing the program in tail recursion. One reason to do so was also due partly to the portability to various multiprocessor environments. When functions are writ-

ten in tail recursion, it can be much easier to understand its behavior since the program tracing is automatic. This easiness in understanding of the behavior of a program will directly translate into an easy conversion to iterations. Those vectorizing compilers or parallelizing compilers can be readily used to convert the Lisp programs into a language suitable for vector or multiprocessors.

The third feature, a simple data structure defstruct, would not necessarily be considered a good feature for parallel environments. The main employing defstruct is that it will simplify the implementation process due to its structuredness. This structured approach will shield the data dependency between data, i.e., dynamically changing memories in the network. However, this dynamic data structure consumes more memory space than other data structures such as lists. There is certainly a trade-off between the runtime memory space and the easiness in programming and debugging.

6.1.2 Descriptions on data structures

The data structures used in the implementation are illustrated in Figure 6.1. An array is used to implement production memory, where each element is a pointer to a production. Each production is implemented in a structure using the Lisp construct defstruct. There are seven entries in the structure of each production (the last one is used for debugging purpose):

- name is a name of the production specified in the production system program.
- no is a unique number assigned to a production. It is for internal use only and is not related to the actual production number specified in the production system program.
- cond is a list of condition elements, each of which is implemented in a structure. More details are presented below.
- act is a list of actions, each of which is implemented in a structure. More details are presented below.
- vars is a list of variables appearing in the LHS of the production.
- negated is a list of flags, where each flag indicates the appearance of a negated condition element.

- tt-last-fired is a time tag to indicate the time, i.e., the production cycle number at which the production is fired,
- last-instantiation is a list of instantiations, where each instantiation is a list of wmes based on which the production fired (not shown in Figure 6.1).

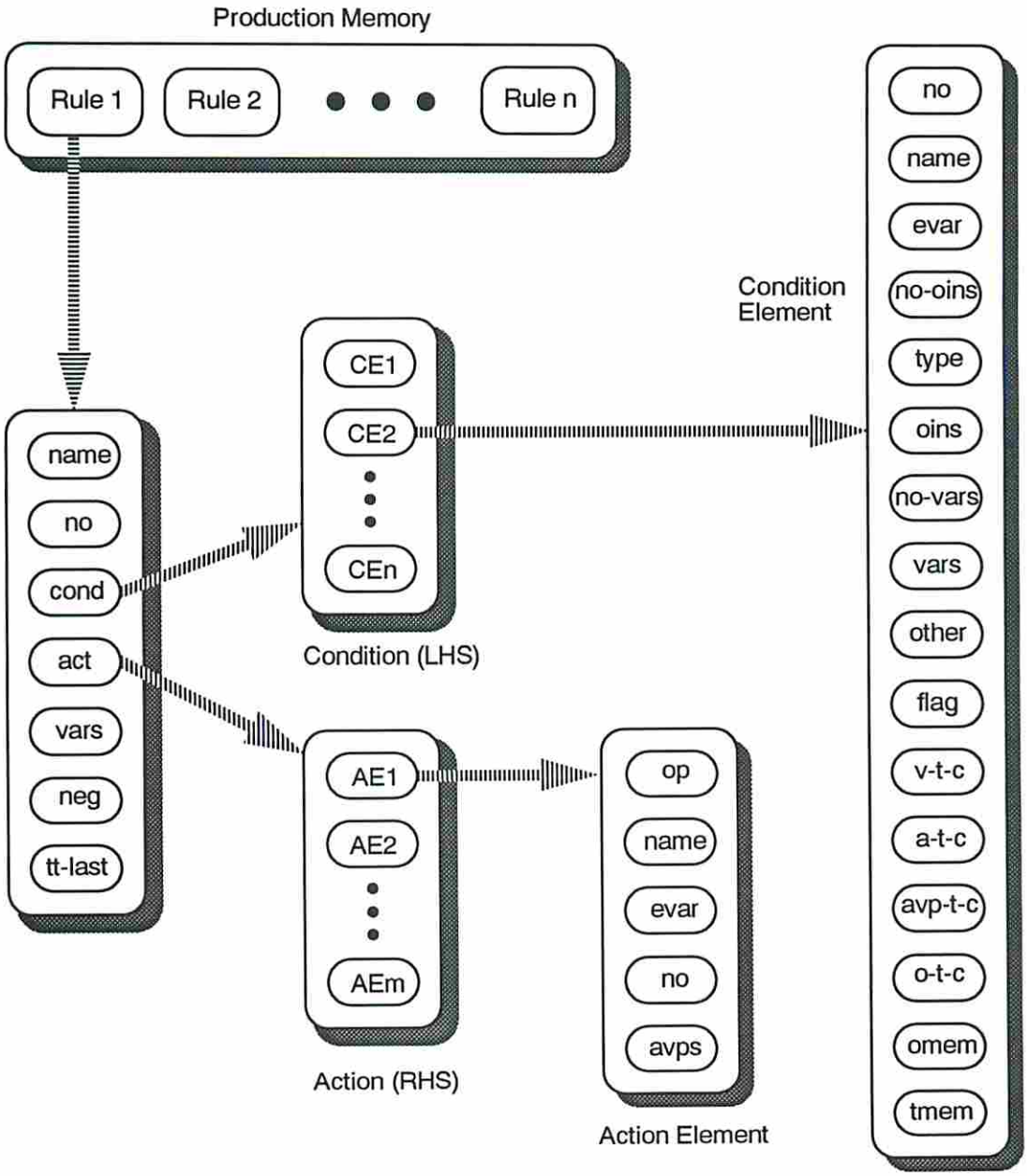


Figure 6.1: Data structure used in the MRN implementation.

Each condition element is implemented in a structure with 16 entries. The first entry, no-in-rule (or no in Figure 6.1), is an ordinal number (or position) of the condition element within the LHS. For example, if a LHS has 3 CEs, the first CE is assigned to 1, the second CE to 2, and so on. This information is useful in deciding which condition element to access instead of searching through the whole list of CEs. It is found that this information helps save the processing time at runtime.

The second entry, name, is simply a name of the CE, if there is any. The third entry, evar-name (or evar in Figure 6.1), is a name of the element variable assigned to the CE, if any. If no element variable name is assigned to the CE, the ordinal number, no-in-rule, will be inserted into the slot. The fourth entry, no-oins, indicates the number of attribute Value Pairs in the CE. The information, no-oins, might be considered redundant since it can be readily obtained by using the Lisp function length. However, in order to reduce the execution time, as much information is extracted at compile time as possible.

The fifth entry, type, indicates whether the CE is a positive or negated condition element. The sixth entry, oins, is a list of one-input nodes. The seventh entry, no-vars, shows how many variables appear in the CE. The eighth entry, vars, is a list of variables appeared in the CE.

The ninth entry, other-atr (or other in Figure 6.1), is a list of attributes extracted from other condition elements. This information is designed exclusively for two-input nodes and the length of this entry indicates the number of tests required when checking the consistency of variable binding is initiated in the two-input node. For example, suppose that the current CE is the second one in LHS and has a variable X. Suppose further that the variable X also appears in the first CE of the LHS. The entry other-atr will have an attribute of the variable X of the first CE. Extracting this kind of information at compile time will reduce a substantial amount of runtime since it does not require any search time.

The tenth entry, mem-changed (or flag in Figure 6.1), is a flag to indicate whether there is any change in the memory attached to the CE. The main purpose of doing so is again to reduce runtime. The memory will be accessed only when this flag is set to true. The value of the flag is also propagated up to the production level to signal that whether this rule should initiate the two-input nodes assigned to the production, if any.

The eleventh entry, var-to-cmp (or v-t-c in Figure 6.1), is a list of variables to be tested when the two-input node ever initiates consistency checking. The length of this list is a number of tests for

the two-input node. The twelfth and thirteen entries serve the same purpose. The twelfth entry, atr-to-cmp (or a-t-c in Figure 6.1), contains a list of attributes, each of which corresponds to a variable in the eleventh entry, var-to-cmp. The thirteen entry, avp-to-cmp (or avp-t-c in Figure 6.1), is nothing but a list of Attribute Value Pairs, where each pair is the one which contains a variable.

The fourteenth entry, op-to-cmp, is a list of operators. When a condition element contains an Attribute Value Pair such as (attr <> <*X*>), the not-equal operator, <>, is extracted from the pair and inserted in the op-to-cmp list.

The last two entries are two memories, omem and tmem. The entry, omem, is a list of wmes, each of which matches the current CE. The last entry, tmem, contains a list of merged wmes. Each merged wme is a list of wmes. If the current CE is the very first one in the LHS, tmem will contain a list merged wmes, where a merged wme is a list of one wme. If the current CE is the second one in the LHS, tmem will contain a list merged wmes, where a merged wme is a list of two wmes. One is the one from omem of the current CE and the other is the one propagated down from tmem of the first CE. For the third CE, tmem will have a list of merged wmes, where a merged will be a list of three wmes. One is from omem of the current CE while the other two wmes are from tmem of the second CE.

The data structure used for actions is quite simple compared to the one used for condition elements. An action is implemented in a structure with five entries, as illustrated in Figure 6.1. The first entry, op, indicates whether the action is make, modify, or remove. The second entry, name, is a name of the action if there is any. The third entry, evar-name, is the element variable name which connects an action of RHS to a condition element of LHS. If there is no element variable name assigned to an action, an ordinal number of the corresponding CE is inserted at compile time. The fourth entry, no-in-rule (or no in Figure 6.1), is an ordinal number of the corresponding CE. The last entry, avps, is a list of Attribute Value Pairs.

The data structure used for wmes is a structure which contains five entries. The first entry, tag, is a time tag to indicate at which production cycle it is generated. The second entry, no, is a number in which it is created throughout the lifetime of production cycles. This reflects the recency of a wme and is used in rule selection step. The third entry, type, indicates it is to be added to or deleted from working memory. The fourth entry, name, is the working memory element name. The last one, avps, is a list of attribute value pairs.

6.1.3 An example on data structures

To give a better understanding of how data structures organized, an example is presented below.

A rule shown below is taken from Monkey and Banana:

```
(p mb.11
    ((goal ^ status active ^ type holds ^ object-name <o>) <goal>)
    ((phys-object ^ name <o> ^ weight light ^ at <p> ^ on <> ceiling) <object>)
    ((monkey ^ at <p> ^ on floor ^ holds null) <monkey>)
    - (phys-object ^ on <o>)
    →
    (modify <object> ^ on null)
    (modify <monkey> ^ holds <o>)
    (modify <goal> ^ status satisfied))
```

After the above rule is compiled, we have the following structure with all the slots filled with information we mentioned earlier:

```
#s(rule name mb.11
    no 11
    cond (#s(ce no-in-rule 1 name goal evar-name <goal> no-oins 3 type +no-vars 1
                vars (<o>) op-to-cmp (equal) other-atr nil mem-changed nilvar-to-cmp (<o>)
                atr-to-cmp (object-name) avp-to-cmp ((object-name <o>))
                oins ((status active) (type holds) (object-name <o>)) omem nil tmem nil)
          #s(ce no-in-rule 2 name phys-object evar-name <object> no-oins 4 type + no-vars 2
                vars (<o> <p>) op-to-cmp (equal) other-atr ((1 object-name)) mem-changed nil
                var-to-cmp (<o>) atr-to-cmp (name) avp-to-cmp ((name <o>))
                oins ((name <o>) (weight light) (at <p>) (on <> ceiling)) omem nil tmem nil)
          #s(ce no-in-rule 3 name monkey evar-name <monkey> no-oins 3 type + no-vars 1
                vars (<p>) op-to-cmp (equal) other-atr ((2 at)) mem-changed nil
                var-to-cmp (<p>) atr-to-cmp (at) avp-to-cmp ((at <p>))
                oins ((at <p>) (on floor) (holds null)) omem nil tmem nil)
          #s(ce no-in-rule 4 name phys-object evar-name nil no-oins 1 type - no-vars 1
                vars (<o>) op-to-cmp (equal) other-atr ((1 object-name)) mem-changed nil
                var-to-cmp (<o>) atr-to-cmp (on) avp-to-cmp ((on <o>)) oins ((on <o>))
                omem nil tmem nil))
    act   (#s(ae op modify name phys-object evar-name <object> no-in-rule 2 avps ((on null)))
          #s(ae op modify name monkey evar-name <monkey> no-in-rule 3
                avps ((holds var (1 object-name))))
          #s(ae op modify name goal evar-name <goal> no-in-rule 1 avps ((status satisfied))))
    vars ((<o>) (<o> <p>) (<p>) (<o>))
    negated (1 (+ + + -))
    tt-last-fired 0
    last-inst nil)
```

Since all the names are self-explanatory as we described earlier, we shall not go through the structure here.

6.1.4 Organization of the program

The MRN-based production system interpreter consists of three major parts: pre-processing, production-cycle, and post-processing, as depicted in Figure 6.2 (also found in the program listing attached to Appendix).

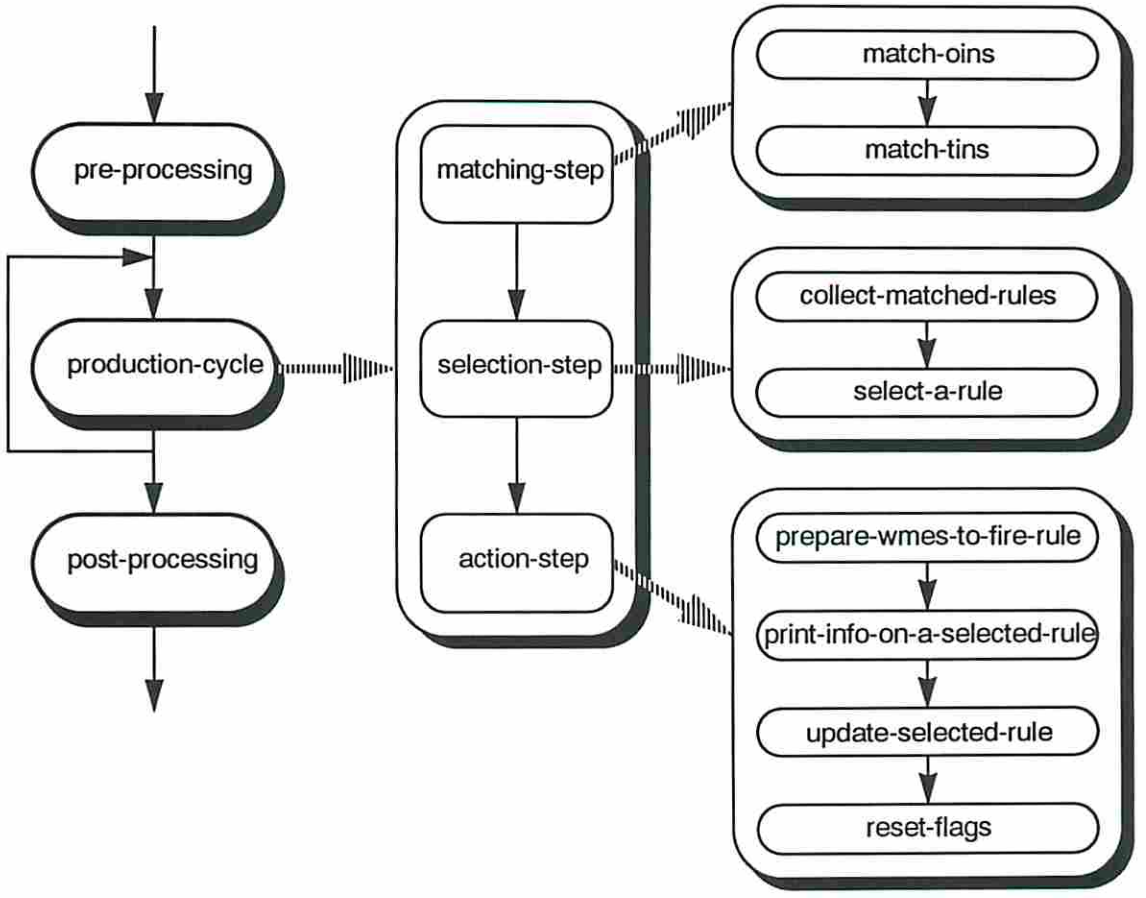


Figure 6.2: Organization of the MRN-based production system interpreter.

The first major part, pre-processing, takes in the production memory and the initial working memory, and compiles into data structures described above. After the rules are compiled, information regarding groups for the MRN network is extracted from the compiled rules. Note that the function names shown in Figure 6.2 are those actual ones used in the program and should be self-explanatory.

The second major part, production-cycle, consists of three functions: matching-step, selection-step, and action-step. The function matching-step calls two functions: match-oins and match-tins. The function match-oins matches wmes and condition elements, and returns a modified production

memory since those matched wmes are stored in memories attached to each condition element. After match-oins completes, the function match-tins is called to find if any wmes can be merged and propagated down the network. The second function of production-cycle is a function selection-step which in turn calls two functions: collect-matched-rules and select-a-rule. The third function of production-cycle is a function action-step which consists of four major parts: prepare-wmes-to-fire-rule, print-info-on-a-selected-rule, update-selected-rule, and reset-flags.

The third major part, post-processing, is designed exclusively for recollection of the statistics which are measured and stored at runtime in the network. There are many functions written for performance analysis but not listed due to space constraint. Those function names used in the program were carefully selected such that they should be self-explanatory. We will not further discuss them. A complete source code is found in Appendix.

6.2 Measurements at Compile Time

Five production system programs were chosen as benchmark programs. They all were executed on a sequential uniprocessor, Sun 4/90. Both the Rete-based OPS5 and MRN-based interpreters were tested to measure their performance. Both interpreters produce exactly the same results, in terms of the number of rule firings, the rule firing sequence, the number of wmes generated, the wme generation sequence, etc. Some facts on the five programs are measured and listed in this section along with the group information which is central to the MRN network.

6.2.1 Benchmark production system programs

There are several reasons in using benchmark systems: among them, it would allow us to compare the relative performance of different production system interpreters. Stating the results of a particular benchmark on two different systems (or approaches) usually causes people to believe that a blanket statement ranking the systems in question is being made. The proper role of benchmarking is to measure various dimensions of production system performance and to order them along with each of these dimensions. At that point, informed users will be able to choose a system that meets their requirements or will be able to tune their programming style to the performance profile.

The first problem associated with benchmarking is knowing what is being used. The five programs chosen for performance analysis are well-known ones and widely used among the researchers in the production system community. One should note here is that the size of production systems is not central to its performance evaluation. Indeed, Gupta has commented in his thesis work that (1) we should not expect smaller production systems (in terms of number of productions) to run faster than larger ones, and (2) there is no reason to expect that larger production systems will necessarily exhibit more speedup from parallelism [24]. The five programs used in this study are briefly described below:

1. *Brick Sorting* is a simple program for a robot to pick a brick from a pool of bricks and place them in ascending or descending order in size.
2. *Monkey and Banana* (MAB) is a program where a hungry monkey grabs a banana hanging from the ceiling, given a couch and a ladder in the room.
3. *N Monkeys and M Bananas* (NMAB) is an extension to the MAB with n monkeys and m bananas, where each monkey is to find a banana for itself, given a similar room environment with more bananas.
4. *Waltz Labeling* is a modified version of the original Waltz labeling algorithm in that it assigns a label (chosen from a finite set of possible labels) to each edge of a line drawing, where each label gives semantic information about the nature of the edge and the regions on each of its sides.
5. *N-Queen* is a classical problem which places n queens on nxn board such that each row, column, and diagonal contains no more than one queen.

The following information collected from the above five production system programs characterize various aspects of the benchmark programs. Note from Table 6.1 that the size of each program is not in the order of hundreds but in the order of tens. However, the problem size is unimportant when analyzing the parallelism in production system programs as Gupta has concluded in his thesis [24]. Our purpose is to measure the relative performance of the MRN approach in terms of execution time along production cycles. What is important in our performance evaluation is the information on groups of a production system program. Indeed, we find that even the small

size of a production system program such as the Brick Sorting problem would suffice, as will be discussed shortly.

Production System	No of Rules	No of CEs	No of Acts	OINs Executed	No of TINs	No of Groups	Avg CEs/Grp	Rule Firings	WMEs Generated
Brick	7	16	15	2336	2	4	4	20	60
MAB	25	70	43	8409	59	5	14	16	58
5MAB	23	60	43	45195	39	5	12	113	278
Waltz	48	198	100	174891	90	5	40	245	297
8-Queen	19	68	71	151985	36	6	11	1044	3866

Table 6.1: Characteristics of benchmark production systems

6.2.2 Measurements on grouping

Grouping the condition elements (CEs) based on the number of Attribute Value Pairs (AVPs) is central to the MRN approach. This would allow us to partition the production systems into many pieces each of which can be processed independent of the incoming newly generated wmes. Measuring the distribution of condition elements over groups at compile time, we can predict the potential parallelism in the given production systems. Figure 6.3 depicts the distribution curve, where the x-axis shows group numbers and the y-axis the percentage of each group in a particular production system program.

In the Brick Sorting problem (Figure 6.3), there are four different groups, where a group-n contains condition elements, each of which has n Attribute-value Pairs. For example, the first group Group2 occupies slightly above 30% whereas Group 5 does 6% of the total number of condition elements. To be more specific, Group 2 has 5 CEs whereas Group5 has only one CE, where the total number of CEs is 16.

As we can see from Figure 6.3, the condition elements of Monkey and Banana are relatively equally distributed over four groups, compared to that of Waltz Labeling where one group is dominant over other groups. This dominance of a group over another is not desirable and does not yield a good performance. We shall come back to this analysis after we run the programs on both approaches.

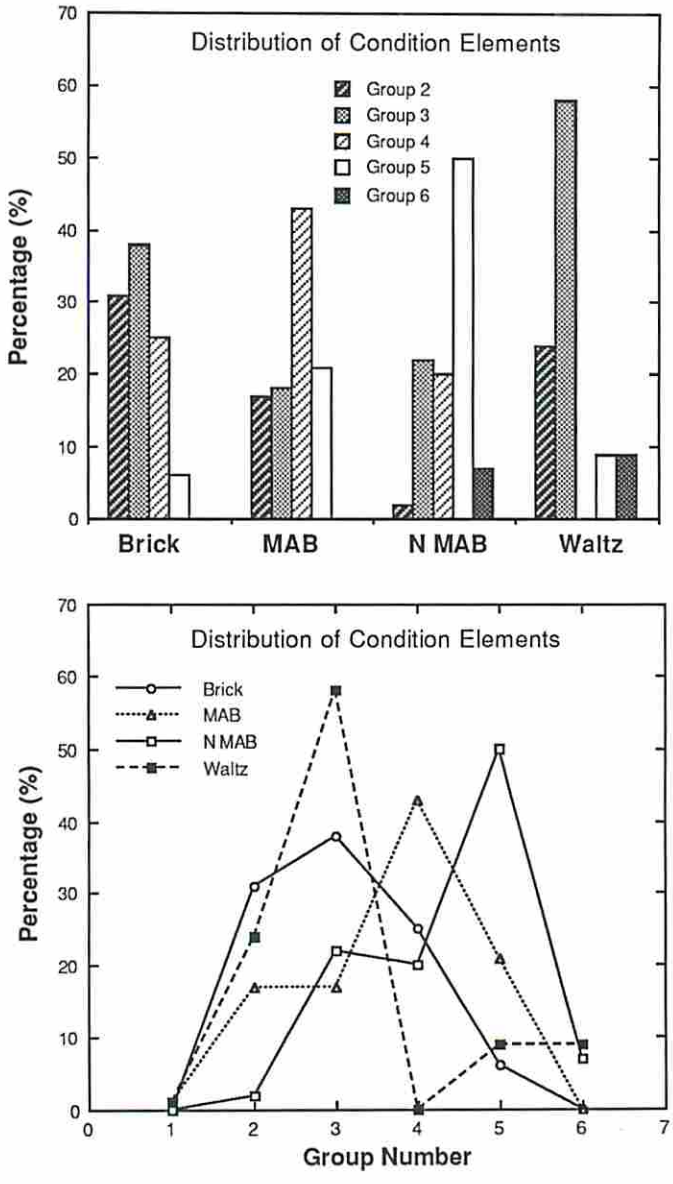


Figure 6.3: Distribution of condition elements over groups measured at compile time.

6.3 Runtime Measurements

Information measured at runtime are classified into three different categories: the execution time of a match step, the number of comparison operations for one-input nodes, and the distribution of wmes. All the measurements are done against production cycle numbers. A window of 20 production cycles is applied to each program, which suffices our purpose.

6.3.1 Execution time on one-input nodes

Figure 6.4 shows the execution time of matching one-input nodes measured at each production cycle. In the figure there are several production cycles whose execution time run off the boundary. They are intentionally left out. Several points running off the boundary are unimportant for our purpose since we wish to show the relative performance.

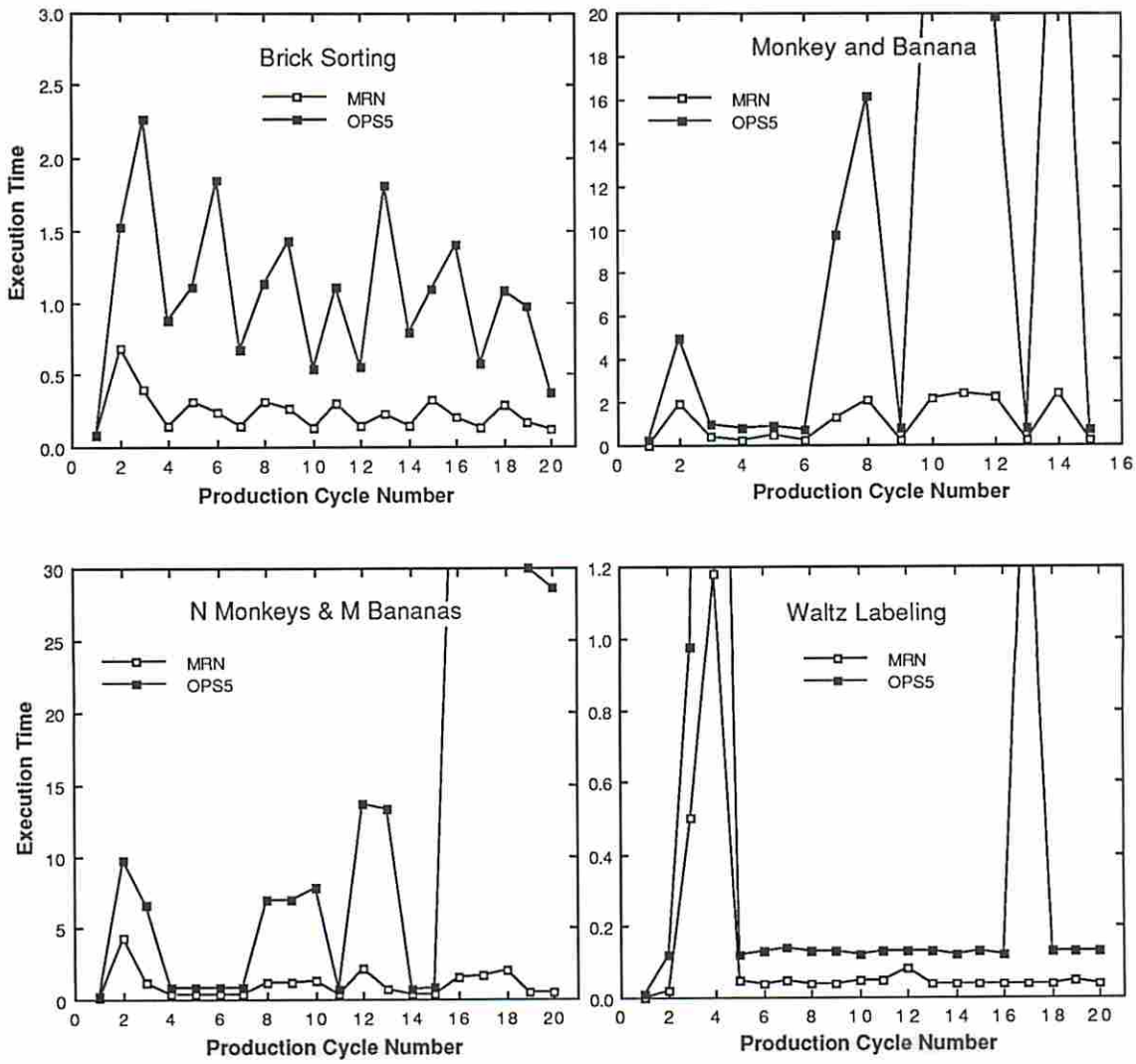


Figure 6.4: Execution time profile of matching one-input nodes.

For Brick Sorting and Waltz, it appears that both the MRN and OPS5 show a rather regular behavior while they maintain a reasonably constant distance between the two curves along the x-axis. For

example, in the Waltz, the differences between two execution time curves for the cycle numbers 5 to 16 are relatively constant, except at the cycle numbers 3, 4, and 17. A similar behavior is also observed in Brick. This kind of proportional distance between two curves is important in predicting the possible outcome of the MRN approach.

The MAB and NMAB, however, exhibit a slightly different behavior compared to Brick and Waltz. For example, the MRN curve in MAB gives an amplification factor higher than the one for Brick or Waltz. This irregular behavior is due partly to the memory management policy, *garbage collection*, in Lisp which contributes to inaccurate performance measurements. We shall give a more accurate measurement shortly. Over all, it is obvious that the MRN outperforms the OPS5 in any of the four problems.

6.3.2 Number of comparison operations on one-input nodes

Another statistic measured at runtime is counting the number of comparison operations. The comparison here is not meant to be the number of one-input nodes or two-input nodes but is an actual comparison operation in Lisp such as (equal x y), where x and y are atoms. To give a better understanding of what criterion is used, consider the following simple Lisp function member which checks whether an atom 'a' is a member of the list 'l':

```
(defun member (a l)
   (cond    ((null l) nil)
            ((equal a (car l)))
            (t (member a (cdr l)))))
```

Now, suppose that the function is called with the following values to the parameters: (member 1 '(2 6 4 7 1)). It is clear that the function member will be called five times and therefore, the number of comparison operations will be five.

As we noted earlier, the criterion, measuring the execution time of one-input nodes in real time, does not correctly reflect the real execution time due to the system load, the number of users, etc. We therefore introduce another criterion, measuring the number of an actual comparison operations, which would serve as a better indicator for our performance measurement. This assumption has been confirmed in program running which will be discussed shortly.

6.3.3 Distribution of groups

Figure 6.6 shows the runtime distribution of wmes and condition elements for four problems. Take the Brick Sorting, for example. At runtime, there is no wme generated for group2, group4, and group6. In other words, no wme with 2 AVPs, 4 AVPs, or 6 AVPS is created by any rule firing rule. Those wmes generated for Brick at runtime fall into either group3 or group5. As we can observe from Figure 6.6, there is a considerable amount of discrepancy between the runtime wme distribution and the compile CE distribution.

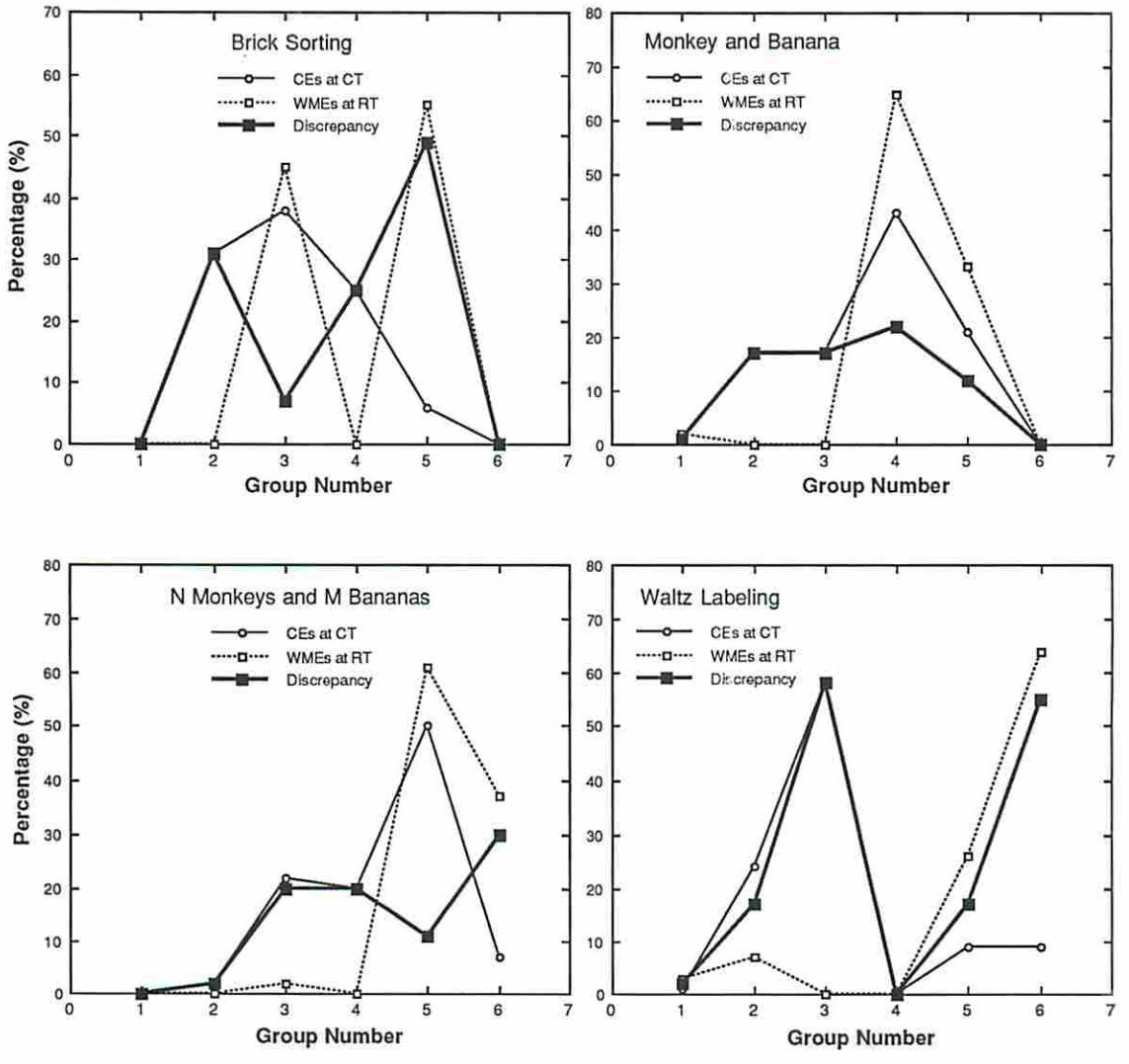


Figure 6.6: Runtime distribution of wmes.

For MAB, however, the situation becomes different. As we can observe from Figure 6.6, the discrepancy for MAB becomes much smaller compared to that for Brick. MAB and NMAB show a relatively low discrepancy whereas Brick and Waltz show a rather high discrepancy in terms of the wme distribution and the CE distribution.

The bar chart shown in Figure 6.7 gives a global view of the runtime distribution of wmes over groups for the four problems. Contrary to the compile time distribution of condition elements, most of the wmes generated at runtime fall into a few distinctive groups. As we observe from the bar chart, all the four problems have basically two groups actively working at runtime. However, all these distribution curves are problem dependent and there is no single rule which can predict the behavior of the runtime distribution of wmes. A simple conclusion we could draw from these discrepancy plots would be that more the discrepancy there is, more the speedup there will be. We shall come back to this when we discuss the ratio of MRN to Rete.

6.4 Performance Evaluation

Based on the foregoing three different types of observations, i.e., one-input match time, number of comparison operations, and distribution of wmes, we shall analyze below the performance of both approaches.

6.4.1 Comparison of MRN and OPS5

Figure 6.8 shows the ratio (or speedup) of MRN to OPS5 on one-input match time for four different programs. Here, the ratio means simply the comparison of two match time units for one-input nodes. Again, x-axis is plotted against the production cycle numbers whereas y-axis indicates the ratio of two different approaches.

For Brick Sorting, for example, the one-input match time of Ops5 at production cycle number 13 is about 8 times more than that of MRN. For NMAB, the one-input match time of Ops5 at the cycle 13 is about 17 times more than that of MRN.

Even though evaluating the two approaches based on the execution time in real time shown in Figure 6.8 may not be accurate due to the system load, the number of users, the number of garbage

collection, etc. It is clear from Figure 6.8 that there is a substantial speedup, ranging from 2 to over 20, depending on the programs and the production cycle number.

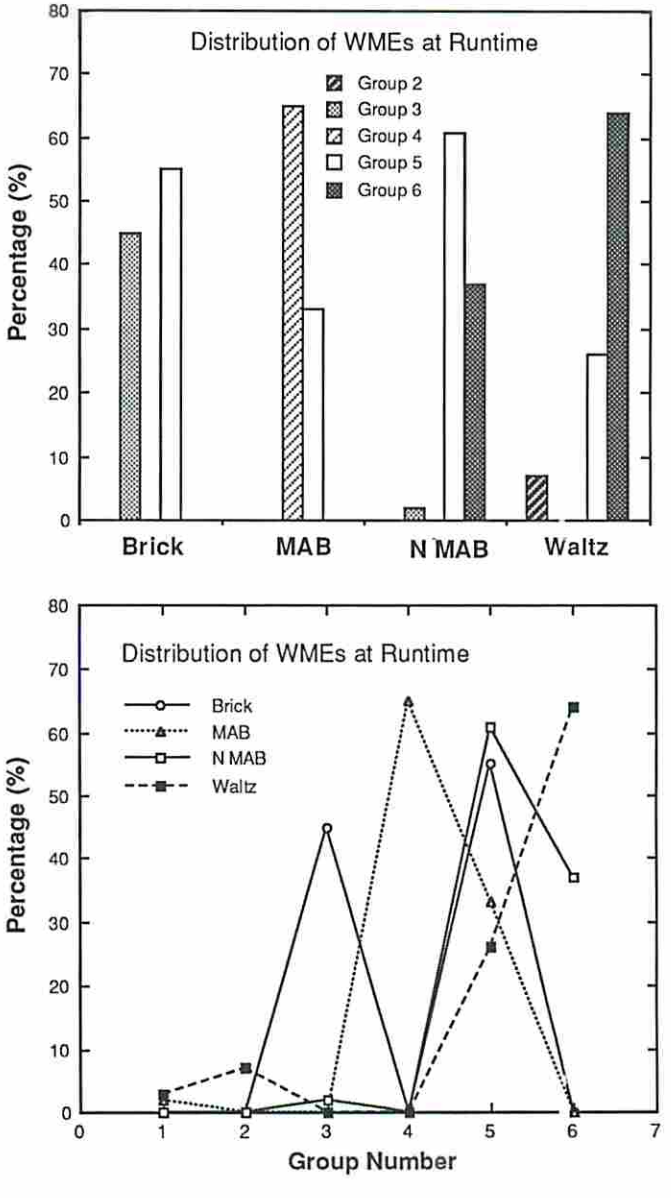


Figure 6.7: Summary on the runtime distribution of wmes.

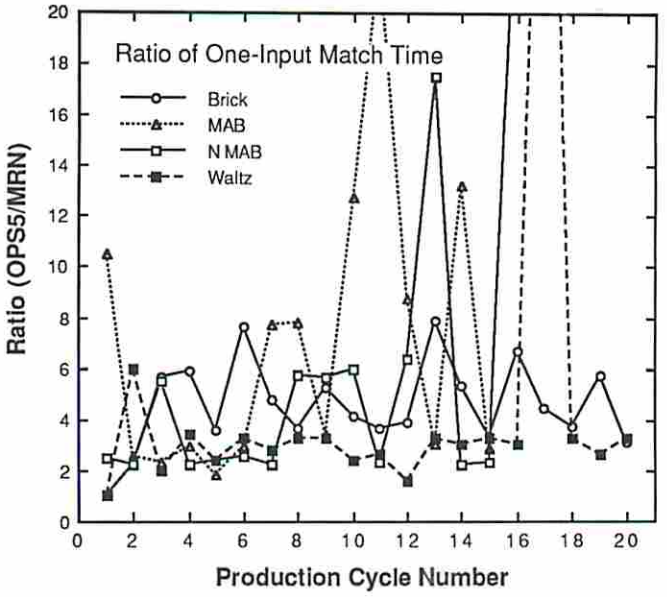


Figure 6.8: Ratio of one-input match time.

Figure 6.9 gives a more accurate performance measure. It uses the number of comparison operations at each production cycle. The x-axis is plotted against the production cycle number while the y-axis is again the ratio (or speedup) of the MRN-based match to Rete-based match. To closely examine the speedup curve, consider again the production cycle number 13 as we did for Figure 6.8

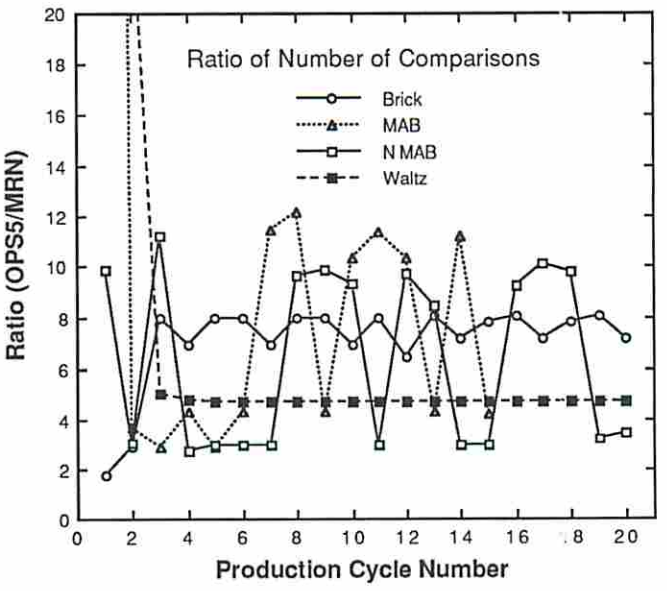


Figure 6.9: Ratio of number of comparison operations.

For Brick Sorting of Figure 6.9, the number of comparison operations performed by the Rete-based match at cycle 13 is about 8 times more than that by the MRN-based match. This speedup of 8 is exactly the same as the one we obtained from Figure 6.8.

Now, consider NMAB of Figure 6.9. The speedup, however, becomes different from what we would expect. Closely examining the NMAB curve of Figure 6.9 at cycle 13, we find that the speedup is 8! We remember that the speedup we obtained from Figure 6.8 for NMAB at cycle 13 is 17. This is not surprising because measuring the real time can be affected by many factors which we iterated several times. Nevertheless, it is clear from the two speedup curves plotted in Figure 6.8 and Figure 6.9 that the MRN-based match algorithm outperforms the Rete-based match algorithm. Since the objective here is to compare the performance of the two match algorithms, the two figures would suffice the stated objective.

Figure 6.10 summarizes the performance of the two approaches. The thin solid line with small circles indicates the average ratio of two approaches based on execution time. The thin dotted line with hollow rectangles shows the average ratio of two approaches based on the number of comparison operations. The thick solid line is the average of the two thin average curves. When the thick line is summed and averaged along the production cycle number of x-axis, it gives an eventual speedup of *six*. In other words, the average speedup of the MRN approach over OPS5 on one-input match time would reach to *six* fold for the four production programs considered in this study.

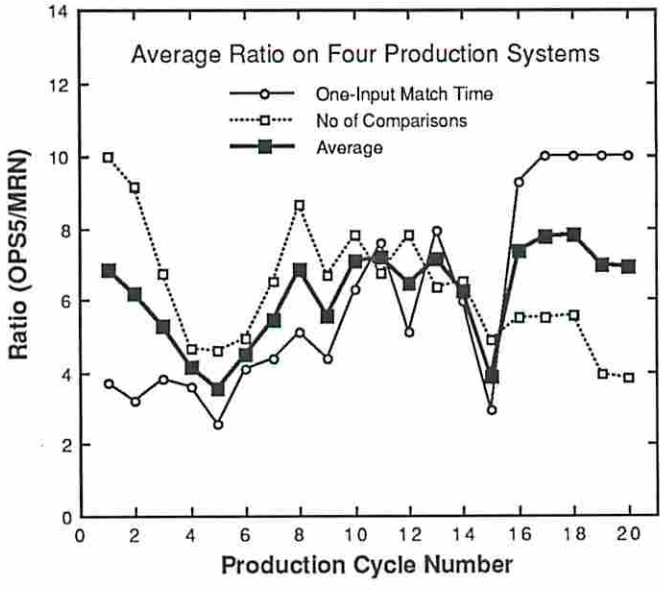


Figure 6.10: Average ratio on four production systems.

6.4.2 Discrepancy in the distribution of wmes and condition elements

It is interesting to note the discrepancy between the compile time distribution of condition elements and the runtime distribution of wmes. By finding the discrepancy between them, we can more accurately locate the behavior of each production system, thereby identifying the potential speedup for a given production system program.

Figure 6.11 displays all the discrepancies for the four programs. Note the discrepancy curves in Figure 6.11 and the speedup curves of Figures 6.8 and 6.9. Among the four discrepancy curves, the Brick curve has a high and regular behavior, which can in turn translate to a high speedup. The curve for Brick in Figure 6.11 verifies this relation in which the speedup is high when the fluctuation is low.

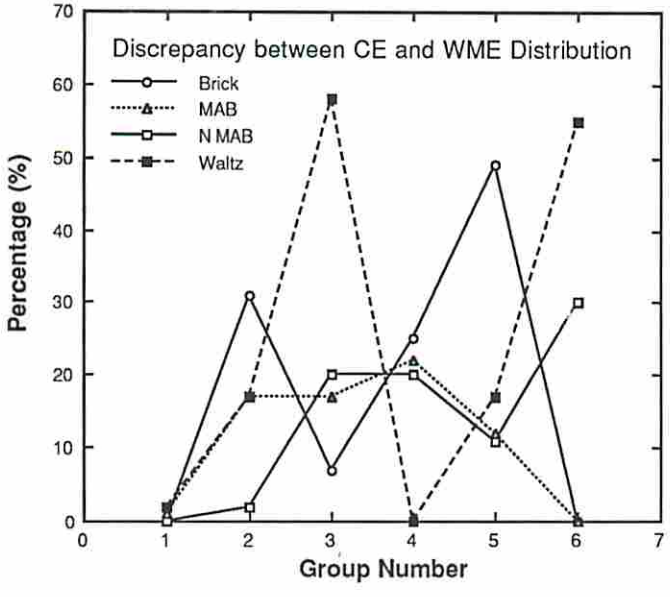


Figure 6.11: Discrepancy between CE and wme distribution.

However, the above statement on the relation between the discrepancy and the speedup would have to be further substantiated by much more experimental results. It would be difficult to relate the discrepancy curve and the speedup curve. Most problems are runtime dependent and a simple prediction rule would be problematic. Based on our observations, it would be concluded that if there is more discrepancy between the compile time distribution of CEs and the runtime distribution of wmes, the production system program would have more potential parallelism.

6.5 Summary

The main purpose of this chapter was in evaluation of the performance of the multiple root node (MRN) approach to production systems. It has been verified that the MRN approach would give a multiplicative effect on the Rete-based production systems. The two criteria used in this study, one-input match time and number of comparison operations in a window of 20 production cycles, have shown that the MRN approach can indeed give 6 fold speedup on the average. All these results are based on the real time execution, none of them in this chapter is based on simulation results.

The experimental results reported in this chapter are all based on the extraction of the parallelism in algorithm level. They are not related to the number of processing elements, as done in the previous two chapters. In a word, the MRN approach, even when implemented in a single uniprocessor Sun 4/90, gave a multiplicative effect on the Rete-based production systems as demonstrated in this chapter. When many processing elements were used, it also gave a further multiplicative effect as demonstrated in the previous two chapters.

A complete execution time of production cycles is illustrated in Figure 6.12 to help give an overall view of the two approaches. Again, the x-axis is plotted in production cycle

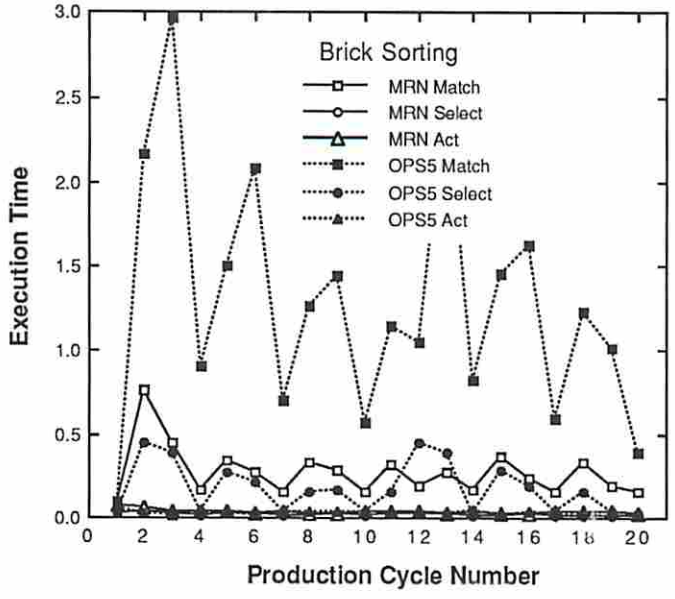


Figure 6.12: A complete execution time for two approaches.

numbers but the *y*-axis at this time is plotted in the total production cycle time. The total matching time is high compared to the selection time or action time as Forgy has indicated [16]. In any case, the MRN-based production system interpreter outperformed the Rete-based production systems.

Chapter 7

Conclusions and Future Research

Advances this thesis has made in the production system paradigm are summarized. The two approaches, the software approach in algorithm level and the hardware approach in implementation level, are summarized in an attempt to reduce the matching time of an inference cycle. Bottlenecks of the most widely used match algorithm have been identified, based on which a new algorithm, the MRN match algorithm, was developed in algorithm level. In implementation level, data-flow principles of execution have been employed as architectural models.

Those topics that are beyond the scope of this thesis but are important to advance the technology in parallel adaptive processing of production systems are listed as future research topics. The first topic, implementing the MRN-based production system interpreter on Intel iPSC/2, is suggested to further verify the performance of the MRN-based interpreter on message-passing multicomputers. The second topic, a SISAL implementation of the MRN-based interpreter, is to identify and extract the complete parallelism profile in production systems. The third topic, hierarchical representation of production system, is intended to depart from the conventional technology of production system processing and to open the new technology in the direction of adaptive processing of production systems.

7.1 Conclusions

The work presented in this thesis has advanced the technology in parallel processing of AI production systems in two aspects: First, it demonstrated the potential of data-flow multiprocessor

systems for the parallel implementation of production systems. Second, it introduced a new match algorithm for AI production systems.

To achieve the stated objectives, the data-flow principles of execution have been chosen as a computational model among various execution principles, since they provide a facility to exploit the maximum potential parallelism in the problem domain. Among the data-flow principles of execution, two principles are selected as architectural models: the dynamic micro-actor principle and the dynamic macro-actor principle. Modifications to the implementation of the basic data-flow principles of execution have been identified before these can be used in an AI computation environment.

The production system (or rule-based expert system) paradigm has been chosen as a benchmark of AI problems because it is one of the most widely used AI applications. As a particular instance of the production system paradigm, an OPS5-like production system interpreter has been adopted again due mostly to its wide use. Among the three steps of an inference cycle of production systems, the match step has been selected since it has the most potential parallelism and more importantly is the most time consuming step.

To demonstrate the applicability of data-flow principles of execution to symbolic computations, the Rete algorithm has been chosen as a benchmark of symbolic computations since it is known as the most efficient match algorithm for production systems. Inefficiencies of the Rete match algorithm when implemented in parallel machines have been identified, based on which a new match algorithm, the MRN algorithm, is introduced to better suit the multiprocessor environments.

Issues related to map the MRN algorithm to multiprocessors are identified and possible strategies have been developed for our data-flow multiprocessor environment. Allocation of condition elements and $O(n)$ iterations existing in two-input nodes to different PEs have proven effective in delivering the parallelism inherent to both the Rete algorithm and the MRN algorithm. Simultaneous distribution of many wmes to many PEs at the same time would give a further speedup in a given configuration of our data-flow architecture.

The MIT Tagged Token Data-flow Computer has been chosen for our micro-actor simulation model. The allocation and distribution strategies we developed were exercised in our simulation. The complete graph for a rule has been created to execute in the simulator model. The MRN and Rete algorithms have been successfully implemented in a data-flow processing environment.

To detect and estimate the different levels of parallelism in the production matching step, various simulations have been performed. Condition elements of the rule were executed in parallel. Simulation results indicated that our micro 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 speedup could be obtained from this approach.

A macro-actor approach for AI production systems has been demonstrated as another efficient implementation tool. Characteristics of production systems from parallel processing have been identified to suit the macro data-flow multiprocessor environment. A simple example on comparison operations has been presented in detail from the macro perspective. Several guidelines have been developed to form well-formed macros. To help clarify the macro approach, a condition element of a rule was converted to macro actors. The results of a deterministic simulation with a production system of 15 rules on the macro data-flow simulator have revealed that the macro approach is an efficient implementation for AI production systems. Indeed, the macro approach gave 17 fold speedup out of 32 PEs used. Furthermore, our MRN-based parallel matching algorithm gave an additional speedup of 3, regardless of the type of a machine used.

To evaluate the algorithmic level performance of the multiple root node (MRN) approach, a complete production system interpreter using the MRN algorithm has been implemented in Common Lisp from *scratch*. Several benchmark production system programs were chosen to evaluate the performance of the MRN match algorithm. Various experiments using the benchmark production system programs were performed on a sequential machine, Sun 4/90, in *real time* and various

statistics were collected both at compile time and at runtime. To ensure the correctness of the performance, an important criterion, a number of comparison operations, was measured against production cycles.

Performance evaluation on the two approaches, the MRN approach and the Rete-based OPS5, were made in terms of number of comparison operations and execution time. Its performance on sequential machine, SUN 4/90, has verified that the MRN approach can give a multiplicative effect on the Rete-based production systems. The two criteria used in this study, one-input match time and number of comparison operations in a window of 20 production cycles, has shown that the MRN approach can indeed give 6 fold speedup on the average. All these results were based on real time execution and were not related to the number of processing elements, as done in the aforementioned data-flow multiprocessor environments. The MRN approach gave a multiplicative effect on the Rete-based production systems when run on a sequential *uniprocessor* Sun 4/90. The MRN approach would give a further multiplicative effect when *many* processing elements were used. In any case, the MRN-based production system interpreter outperformed the Rete-based production system interpreter.

7.2 Future Research: Implementation of Production Systems on Intel iPSC/2 Multicomputer

Advent of the second generation message passing multicomputers has gained much attention on processing AI problems due to its low cost and fast message passing capability. According to the simulation results reported recently, the message passing multicomputers can effect the large matching time inherent in production system processing [1,27]. This promising simulation results are due partly to the fact that the recent technological advances in computer architecture can substantially reduce the large communication overhead incurred in message passing multicomputers.

As we have seen in the previous chapters, the MRN-based production system interpreter has been developed to suit the parallel processors. Its algorithm level improvement yielded a substantial speedup even when a single and sequential processor is used. It would be interesting to further

explore the performance of the MRN-based interpreter in message passing multicomputers. Our choice for the message passing computer is an Intel Hypercube iPSC/2 (intel Personal Super Computer) since it is available and can be readily used. The Intel iPSC was originally designed for numerical supercomputing. Recently, a version of Concurrent Lisp has become available for the iPSC that attempts to exploit the parallelism offered by the hypercube topology in parallel/distributed AI processing.

By porting the MRN-based production system interpreter on Intel Hypercube, the following two important objectives can be achieved: (1) a parallelism profiling of the MRN-based production system interpreter and the Rete-based OPS5, and (2) verification of the claim derived from the simulation results reported by Gupta et al. [1]. When production systems are executed in a real multiprocessor (or multicomputer) environment, the runtime behavior of nondeterministic AI production systems can be better understood, thereby identifying the source of parallelism or bottlenecks of the given problem under consideration. Doing so would clarify the relative performance of the MRN-based production system interpreter, which in turn would be able to advance the technology in parallel/distributed processing of production systems.

The second objective is similar to the first objective in nature. However, it differs in that experiences gained from an *actual* implementation of an AI system on a message passing multicomputer (*not* simulator) would help guide the right direction for prospective AI implementors who consider implementing AI systems on message passing computers. Considering that an implementation of a large and practical AI system is costly and often requires much time and effort, it would be advantageous to predict the possible outcome so as to minimize the effort involved.

7.3 Future Research: Implementation of Production Systems in SISAL

The second topic, a SISAL (Streams and Iterations in Single Assignment Language) implementation of the MRN-based interpreter, is intended to identify and extract the complete parallelism

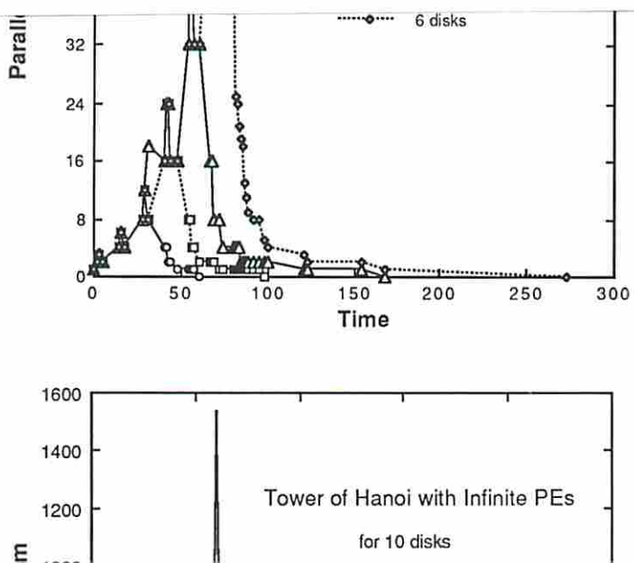
profile in production systems. The high-level language, SISAL, developed at Lawrence Livermore National Laboratory in cooperation with other institutions (Colorado State University, University of East Anglia, University of Manchester, Digital Equipment Corporation), is one of the high-level applicative languages, intended to serve as a side effect free, parallel language for multiprocessor systems [44].

In SISAL, six basic scalar types are defined: boolean, integer, real, double real, null, and character. Data structures in SISAL may be records, unions, arrays, or stream. Each basic data type has its associated set of operations, while record, union, array, and stream types are treated as mathematical sets of values just as the basic scalar types. In particular, under the forall construct, these types can be used to support identification of concurrency for execution on highly parallel processors. It is the identification of concurrency that is central to the SISAL implementation of production systems.

To further clarify the objective stated above, consider a classical example, Tower of Hanoi. Since most AI search problems are nondeterministic and I/O bound, i.e., more memory accesses than arithmetic and logic computations in nature, it would be difficult to identify the parallelism in them. However, SISAL allows us to identify the potential parallelism to a certain accuracy. The Tower-of-Hanoi can be implemented in SISAL in double recursion as follows:

```
define hanoi
type twodim = array[array[character]];
function hanoi (n:integer; from,to,use:character returns twodim)
    if n = 0 then array twodim []
    else hanoi (n-1, from, use, to) ||
        array [1:array [1:from, to]] ||
        hanoi (n-1, use, to, from)
    end if
end function
```

Using one of the capabilities SISAL provides, the parallelism profile can be identified, as plotted in Figure 7.1. The top plot shows the parallelism profile of the Tower of Hanoi by using 64 processing elements. The bottom plot assumes an infinite number of processing elements with 10 disks. The actual execution time for the bottom one is 3191, more than six times of what is shown



The main reason exhibiting such an impulse type of parallelism is that the problem itself is sequential in nature. Unlike numeric computations, moving one state to another when searching the state space requires a completion of the current state. In other words, moving one state to another can be done only when the current state is completed. A move to a next state based on the partial (or incomplete) state would likely result in a conflicting or incorrect state. This conflicting state in the search space is obviously meaningless. The burst type of parallelism profile demonstrated in Figure 7.1 clearly indicates that there something must be done to AI search problems. This is where parallel processing techniques can take part in to improve the parallelism profile.

Implementing production system interpreters in SISAL, not only can we identify the source of potential parallelism such as the one depicted in Figure 7.1 but more importantly can make an effective use of the parallelism information. A further understanding of the problem under consideration would allow us to develop a more suitable, and/or faster matching algorithm for production systems. An efficient and faster processing of production systems would in turn give us an opportunity to scale up the size of production systems which would hopefully solve the problems in real world.

7.4 Future Research: Connectionist Production Systems

A new representation technique, called *hierarchical representation*, is introduced as one of the possible topics for future research. An objective behind developing this new representation technique is to help develop connectionist adaptive production systems. As we have seen from the previous chapters, symbolic processing of production systems requires much processing power in terms of both the hardware and the software. By using a dramatically different approach like the one we introduce in this chapter, the symbolic production systems can be transformed into conventional numeric systems, where the search and match problem no longer persists. This transformation of symbolic systems to numeric systems would then allow us to relatively easily map the production systems to connectionist architectures, thereby being able to exploit the advantages the connectionist architectures have to offer.

7.4.1 Feature hierarchy in production systems

In one end of the spectrum of representation techniques used in artificial neural networks (*ANNs*) is the local representation [17,55], where a concept is assigned to a neuron. While it is quite straightforward to implement and easy to understand its mapping, the local representation reveals several major drawbacks such as difficulty in capturing information regarding *variable bindings*, weakness in *fault tolerant* computing, *varying* number of neurons for different problem size, and so on.

On the other end of the spectrum, the distributed representation is used, which assigns a concept to a pattern of activity of many neurons [29]. This method which is often said to resemble the way in which the brain works provides a graceful degradation for faulty neurons, thereby giving fault tolerant computing. However, it is very difficult to visualize the mapping and to understand how it works. Without a good learning algorithm, it is almost impossible for a large problem size to be appropriately trained for practical applications [14].

To overcome the difficulties in local and distributed representations, we introduce a *hierarchical* representation which attempts to combine the two techniques at different levels of information hierarchy. In the high level of the information hierarchy that we obtain from the problem domain, the local representation is used to partition the problem while in the low level, the distributed representation is used to implement the mapping of neurons. A set of features we derive from the production system domain is listed below in order of decreasing importance:

- f_1: A number of AVPs in a pattern
- f_2: Similarity in *attribute* parts of patterns
- f_3: Similarity in *value* parts of patterns

The value parts in the third feature include not only the constants assigned to attributes but also variables. The three features f_1, f_2, f_3 above are used as criteria to partition the patterns into groups of patterns. A tree for each feature f_i is constructed from the groups of patterns. Merging three trees

into one, we obtain a *partitioned pattern tree* (PTT). Upon constructing PPT, we build a 3-dimensional space $\Re$, called *feature space*, where a pattern p_i of the production system can then be uniquely defined as a pattern vector $p_i(f_1,f_2,f_3)$.

7.4.2 Representing PM and WM in feature space

Let $p_i=[(a_1\ v_1),...,(a_n\ v_n)]$ be a pattern (condition pattern, action pattern, or wme) in a production system, where a_i is an attribute and v_i is a fixed value or a variable. Suppose that a production system has m patterns, $P=\{p_1,...,p_m\}$. We shall define the three features, f_1,f_2,f_3, as follows: (1)Given a pattern $p_i=[(a_1\ v_1),...,(a_n\ v_n)]$, it has n number of AVPs, denoted by $\eta(p_i)=n$. (2)Two patterns $p_i=[(a_1\ v_1),...,(a_n\ v_n)]$ and $p_j=[(a_1'\ v_1'),...,(a_n'\ v_n')]$ are said to be *attribute similar*, denoted by $p_i \approx p_j$, if $\forall i,\ a_i=a_i'$. (3) p_i and p_j are said to be *value similar*, denoted by $p_i \cong p_j$, if $\forall i,\ v_i=v_i'$. From (2) and (3), we define that p_i is *equivalent* to p_j, denoted by $p_i \equiv p_j$ if $p_i \approx p_j$ and $p_i \cong p_j$,

The first feature f_1 partitions P into n subsets $P_1,...,P_n$ such that for $p_i \in P_k$, $\eta(p_i)=k$ for all i. After partitioning, each pattern p_j in P_k will have k AVPs. The second feature f_2 partitions P into m subsets $P_1,...,P_m$ such that for $p_i,p_j \in P_k$, $p_i \approx p_j$ $\forall i,j,\ i \neq j$. After partitioning, patterns in P_k will be similar in attributes. The third feature f_3 which uses the *value* similarity measure partitions P into r subsets $P_1,...,P_r$ such that for $p_i,p_j \in P_k$, $p_i \cong p_j$ $\forall i,j,\ i \neq j$. After partitioning, patterns in P_k will be similar in values. Consider two rules listed below:

Rule 1	**Rule 2**
p_1: [(a 1) (b 2)]	p_4: [(p 2) (q 3) (r 3)]
p_2: [(a 1) (b 3)]	p_5: [(a 3) (q 2)]
p_3: [(p 1) (q 2) (r 3)]	p_6: [(t 3)]
→	→
[Make (c 1) (d 2)]	[Remove 1st pattern]

Let $P=\{p_1,...,p_6\}$ be the six condition patterns. The first feature f_1 partitions P into $\{p_6\}$,

$\{p_1,p_2,p_5\}$, and $\{p_3,p_4\}$ since $\eta(p_6)=1$, $\eta(p_1)=\eta(p_2)=\eta(p_5)=2$, and $\eta(p_3)=\eta(p_4)=3$, as depicted in Figure 7.2(a). Using f_2, we obtain $\{w_6\}$, $\{p_1,p_2\}$, $\{p_5\}$, and $\{p_3,p_4\}$, as shown in Figure 7.2(b),

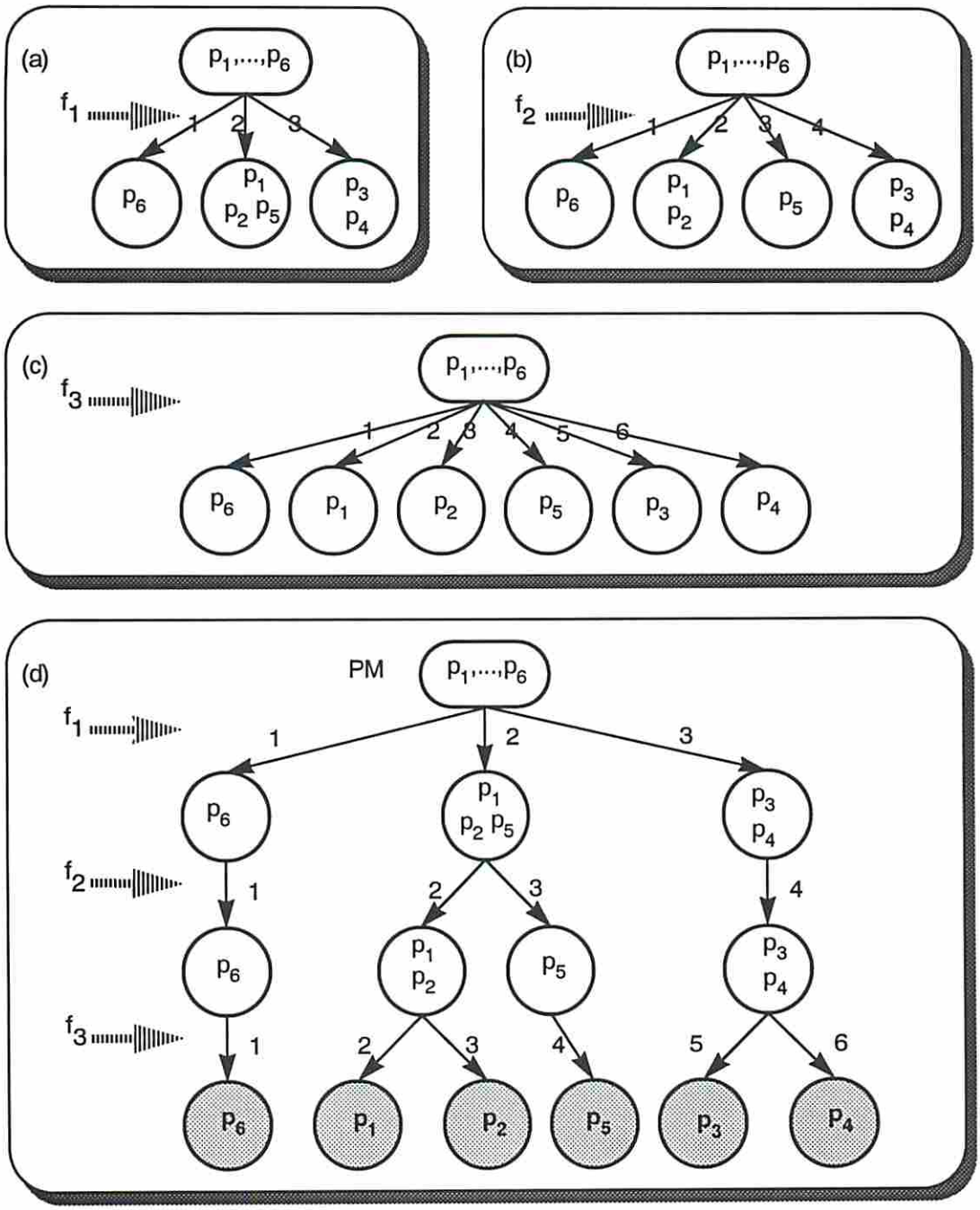


Figure 7.2: Partitioned Pattern Tree (PTT) for PM. (a) f_1 partitions P into three groups, (b) f_2 into four groups, (c) f_3 into six groups, (d) All three subtrees in (a),(b),(c) are merged to form a partitioned pattern tree. Numbers next to arcs uniquely define each pattern in the feature space.

since $p_6 \neq p_1 \approx p_2 \neq p_5 \neq p_3 \approx p_4$. The third feature f_3 partitions P into $\{p_1\}$, $\{p_2\}$, $\{p_3\}$, $\{p_4\}$, $\{p_5\}$, and $\{p_6\}$, as shown in Figure 7.2(c). Numbers on arcs indicate the group a pattern belongs to. Merging (a), (b), and (c) of Figure 7.2 yields a partitioned pattern tree, as depicted in Figure 7.2(d).

Once the partitioned pattern tree is built, the assignment process for each pattern to the feature space is automatic. The numbers next to arcs in Figure 7.2 are assigned to coordinate values (f_1,f_2,f_3) of the feature space. From Figure 7.2(d), we obtain the following coordinates: $p_1(2,2,2)$, $p_2(2,2,3)$, $p_3(3,4,5)$, $p_4(3,4,6)$, $p_5(2,3,4)$, and $p_6(1,1,1)$. Figure 7.3 shows the mapping of six patterns to corresponding points in the 2-dimensional feature space $\Re^2$ (the third feature f_3 is not shown for the simplicity of presentation).

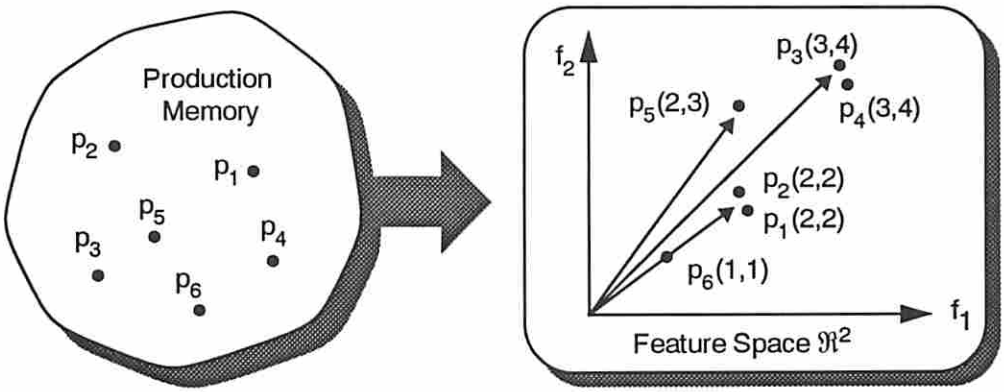


Figure 7.3: A production memory in 2-dimensional feature space.

It is interesting to note in Figure 7.3 that those patterns such as p_1,p_2 that have the same number of AVPs and are similar in attributes form a *cluster* in the 2-dimensional feature space. It is this cluster that we use to match patterns over wmes.

We are not going to implement this hierarchical representation in the proposed architecture. Instead, we shall briefly outline the way to implement the *pattern matching* procedure. Suppose that all the condition patterns are partitioned into c clusters, $C_1,...,C_c$, in the 2-dimensional feature space

$\Re^2$. A cluster C_i which represents a set of particular pattern vectors is assigned to a unit network N_i, consisting of m neurons $n_1,...,n_m$. The coordinate value (f_1,f_2) for C_i, denoted as $C_i(f_1,f_2)$, can then be trained in a unit network N_i such that when a wme w_j is presented to N_i, the corresponding cluster can be activated by N_i. By building a set of N_i, i=1,...,c and training them, we will be able to achieve $O(1)$ pattern matching time. Furthermore, by doing this the heuristics which enable the *good* or *optimal* selection of choice points in the search space can be efficiently represented in terms of energy function. Selection of a choice point in the context of search tree is basically the same as an optimization process in the context of artificial neural networks.

Bibliography

[1] Acharya, A., and Tambe, M., "Production Systems on Message Passing Computers: Simulation Results and Analysis," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[2] Arvind, and Gostelow, K.P., "The U-Interpreter," *Computer*, February 1982, pp.42-49.

[3] Arvind, and Iannucci, R.A., "Two Fundamental Issues in Multiprocessing: the Data-flow Solution," *Technical Report* TM-241, Laboratory for Computer Science, MIT, September 1983.

[4] Arvind, Kathail, V., and Pingali, K., "A Data-flow Architecture with Tagged Tokens," *Technical Report* TM-174, Laboratory for Computer Science, MIT, September 1980.

[5] Arvind, and Nikhil, R.S, "Executing a Program on the MIT Tagged-Token Data-flow Architecture," *IEEE Transactions on Computers*, March 1990, pp.300-318.

[6] Arvind, and Thomas, R.E., "I-structures: An Efficient Data Type for Functional Languages," *Technical Report* TM-178, Laboratory for Computer Science, MIT, 1980.

[7] Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs" *Communications of the ACM* 21, August 1978, pp.613-641.

[8] Bic, L., "Processing of Semantic Nets on Data-flow Architecture" *Artificial Intelligence* 27, 1985, pp.219-227.

[9] Brownston, L., Farrell, R., Kant, E., and Martin, N., Programming Expert Systems in OPS5, Addison-Wesley Publishing Company, Reading, MA, 1985.

[10] Buchanan, B.G. and Shortliffe, E.H., Rule-Based Expert Systems, Addison-Wesley Publishing Company, Reading, MA, 1984.

[11] Dennis, J.B, First Version of a Data-Flow Procedure Language, MAC Technical Memorandum 61, MIT, 1973.

[12] Dennis, J.B., Lim, W.Y.-P., and Ackerman, W.B, "The MIT data-flow engineering model," in *Proceedings of the IFIP 9th World Computer Conference*, Paris, September 1983, Information Processing 83, North-Holland, pp.553-560.

[13] Duda, R.O., Gaschnig, J., Hart, P.E., Konolige, K., Reboh, R., Barrett, P., and Slocum, J., De-

velopment of the PROSPECTOR Consultation System for Mineral Exploration, *Technical Report*, SRI Projects 5821 and 6415, SRI International, Inc., 1978.

[14] Fahlman, S., and Hinton, G., "Connectionist Architecture for Artificial Intelligence" *Computer*, January 1987, pp.100-109.

[15] Fikes, R., and Nilsson, N., "STRIPS: a New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence* 2, 1971, pp.189-208.

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

[17] Gallant, S., "Connectionist expert systems," *Communications of the ACM* 31, February 1988, pp.152-169.

[18] Gaudiot, J-L., and Ercegovac, M.D., "Performance Evaluation of A Simulated Data-flow Computer with Low-resolution Actors," *Journal of Parallel and Distributed Computing*, Academic Press, 1985, pp.321-351.

[19] Gaudiot, J-L., "Data-Driven Multicomputers in Digital Signal Processing Applications" in *Proceedings of the IEEE*, September 1987.

[20] Gaudiot, J-L. and Bic., L. (Eds.), Advanced Topics in Data-flow Computing, Prentice Hall, Englewood Cliffs, NJ, 1990.

[21] Gaudiot, J-L., and Sohn, A., "Data-Driven Multiprocessor Implementation of the Rete Match Algorithm," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1988, pp.256-260.

[22] Gaudiot, J-L., and Sohn, A., "Data-Driven Parallel Production Systems," *IEEE Transactions on Software Engineering*, March 1990, pp.281-293.

[23] Gupta, A., "Implementing OPS5 Production Systems on Dado" in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1984, pp.83-91.

[24] Gupta, A., Parallelisms in Production Systems, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[25] Gupta, A., Forgy, C.L., Kalp, D., Newell, A., and Tambe, M., "Parallel OPS5 on the Encore Multimax," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1988, pp.271-280.

[26] Gupta, A., Forgy, C.L., Newell, A., and Wedig, R., "Parallel Algorithms and Architectures for Rule-Based Systems," in *Proceedings of the International Symposium on Computer Architecture*, June 1986, pp.116-120.

[27] Gupta, A., and Tambe, M., "Suitability of Message Passing Computer for Implementing Production Systems," in *Proceedings of the National Conference on Artificial Intelligence*, Saint

Paul, MN, August 1988, pp.687-692.

[28] Gurd, J.R., Kirkham, C.C., and Watson I., "The Manchester prototype data-flow computer," *Communications of the ACM* 28, January 1985, pp.34-52.

[29] Hinton, G.E., McClelland, J.L., and Rumelhart, D.E., "Distributed Representations," in Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol 1: Foundations, J.L. McClelland and D.E. Rumelhart (Eds.), MIT Press, 1986.

[30] Hinton, G.E., and Anderson, J.A. (Eds.), Parallel Models of Associative Memory (Updated Edition), Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[31] Hiraki, K., Sekiguchi, S., Shimada, T., "Status Report of SIGMA-1: a Data-flow Supercomputer," in Advanced Topics in Data-flow Computing, J-L. Gaudiot and L. Bic (Eds.), Prentice Hall, Englewood Cliffs, NJ, 1990.

[32] Hiraki, K., Shimada, T., and Nishida, K., "A Hardware Design of the SIGMA-1: A Data-flow Computer for Scientific Computations," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1984, pp.851-855.

[33] Hopfield, J.J., and Tank, D.W., "Neural Computation of Decisions in Optimizing Problems," *Biological Cybernetics* 52, 1985, pp.141-151.

[34] Ishida, T., "Optimizing Rules in Production System Programs," in *Proceedings of the National Conference on Artificial Intelligence*, Saint Paul, MN, August 1988, pp.699-704.

[35] Ishida, T., "Methods and Effectiveness of Parallel Rule Firing," in *Proceedings of the IEEE Conference on Artificial Intelligence Applications*, Santa Barbara, CA, March 1990.

[36] Ishida, T., and Stolfo, S.J., "Towards the Parallel Execution of Rules in Production System Programs," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1985, pp.568-575.

[37] Kelly, M.A., and Seviora, R.E., "A Multiprocessor Architecture for Production System Matching," in *Proceedings of the National Conference on Artificial Intelligence*, August 1987, pp.36-41.

[38] Kelly, M.A., and Seviora, R.E., "An Evaluation of DRete on CUPID for OPS5," in *Proceedings of the International Joint Conference on Artificial Intelligence*, Detroit, MI, August 1989, pp.84-90.

[39] Kuo, S., Moldovan, D., and Cha, S., "Control in Production System with Multiple Rule Firings," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1990, pp.243-246.

[40] Kuo, S., and Moldovan, D., "Performance Comparison of Models for Multiple Rule Firing," in *Proceedings of the International Joint Conference on Artificial Intelligence*, Sydney, Australia, August 1991.

[41] Lin, C.M., "Numerical Partial Differential Equation Solvers on Variable-grain Data-flow Multiprocessor Systems," *Ph.D Thesis*, CENG TR:91-13, Department of Electrical Engineering - Systems, University of Southern California, 1991.

[42] McDermott, J., "R1: A Rule-based Configurer of Computer Systems," *Artificial Intelligence* 19, 1982, pp.39-88.

[43] McDermott, J., and Forgy, C., "Production System Conflict Resolution Strategies," in Pattern Directed Inference Systems, D. Waterman and F. Hayes-Roth (Eds.), Academic Press, New York, NY, 1978.

[44] McGraw, J.R., and Skedzielewski, S.K., SISAL: Streams and Iterations ina Single Assignment Language, Language Reference Manual, Version 1.2, *Technical Report* M-146, Lawrence Livermore National Laboratory, March 1985.

[45] Miranker, D.P., Treat: A New and Efficient Match Algorithm for AI Production Systems, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[46] Miranker, D.P., Kuo, C., Browne, J.C., "Parallelizing Compilation of Rule-based Programs," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1990, pp.II 247-251.

[47] Moldovan, D.I., "Rubic: A Multiprocessor Rule-Based Production Systems," *IEEE Transactions on Systems, Man, and Cybernetics* 19, July/August 1989, pp.699-706.

[48] Oshisanwo, A.O., and Dasiewicz, P.P., "A Parallel Model and Architecture for Production Systems," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1987, pp.166-169.

[49] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., and Yuba, T., "An Architecture of a Data-flow Single Chip Processor," in *Proceedings of the International Symposium on Computer Architecture*," Jerusalem, Israel, May 1989, pp.46-53.

[50] Schmolze, J. "A Parallel Asynchronous Distributed Production System," in *Proceedings of the National Conference on Artificial Intelligence*, Boston, MA, July 1990, pp.65-71.

[51] Schor, M., Daly, D., Lee, H.S., and Tibbitts, R., "Advances in Rete Pattern Matching," in *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986, pp.226-232.

[52] Schreiner, F., and Zimmermann, G., "PESA-1: A Parallel Architecture for Production Systems," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1987, pp.166-169.

[53] Shaw, D.E., "On the Range of Applicability of an Artificial Intelligence Machine," *Artificial Intelligence* 32, 1987, pp.151-172.

[54] Sohn, A., and Gaudiot, J-L., "Multilayer of Ring-Structured Feedback Network for Produc-

tion System Processing," in *Proceedings of the IEEE International Conference on Tools for Artificial Intelligence*, Washington, DC, October 1989, pp.457-464.

[55] Sohn, A., and Gaudiot, J-L., "Connectionist Production Systems in Local Representation," in *Proceedings of the IEEE International Joint Conference on Neural Networks*, Washington, DC, January 1990, vol.II pp.199-202.

[56] Sohn, A., and Gaudiot, J-L., "Representation and Processing Production Systems in Connectionist Architectures," *International Journal of Pattern Recognition and Artificial Intelligence* 4, June 1990, pp.199-214.

[57] Sohn, A. "Technical Memos on the C-Pyramid Image Processing Architecture, the Viewer Graphics Package, the Face Recognition Systems, and the Transputer Multiprocessor," *Lab Memos*, Center for Neural Engineering, University of Southern California, 1990.

[58] Sohn, A., and Gaudiot, J-L., "A Connectionist Approach to Learning Legal Moves in Tower-of-Hanoi" in *Proceedings of IEEE International Conference on Tools for Artificial Intelligence*, Fairfax, Virginia, November 1990.

[59] Sohn, A., and Gaudiot, J-L., "A Macro Actor/Token Implementation of Production Systems on a Data-flow Multiprocessor" in *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, August 1991.

[60] Sohn, A., and Gaudiot, J-L., "Connectionist Production Systems in Local and Hierarchical Representation," in Applications of Learning and Planning Methods, N. Bourbakis (Ed.), World Scientific Publishing Co., Teaneck, NJ, 1990, pp.165-180.

[61] Sohn, A., and Gaudiot, J-L., "A Survey on the State-of-the-Art Research in Parallel Processing of Production Systems" *International Journal of Artificial Intelligence Tools* 1, January 1992.

[62] Srini, V.P., "An Architectural Comparison of Data-flow Systems," *Computer*, March 1986, pp.68-87.

[63] Stolfo, S.J., "Initial Performance of the DADO2 Prototype," *Computer*, January 1987, pp.75-83.

[64] Stolfo, S.J., and Miranker, D.P., "The DADO Production System Machine," *Journal of Parallel and Distributed Computing* 3, Academic Press Inc., 1986, pp.269-296.

[65] Tenorio, M.F.M., and Moldovan, D.I., "Mapping Production Systems into Multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 1985, pp.56-62.

[66] Touretzky, D.S. and Hinton, G.E., "Symbols Among the Neurons: Details of a Connectionist Inference Architecture," in *Proceedings of the International Joint Conference on Artificial Intelligence*, August 1985, pp.238-243.

[67] Touretzky, D.S. and Hinton, G.E., "A Distributed Connectionist Production System," *Cognitive Science* 12, 1988, pp.423-466.

[68] Veen, A.H, "Data-flow Machine Architecture," *ACM Computing Surveys* 18, December 1986, pp.365-396.

[69] Wah, B.W., Lowrie, M.B., and Li, G.-J., "Computers for Symbolic Processing," *Proceedings of the IEEE* 77, April 1989, pp.509-540.

[70] Yoo, N., and Gaudiot, J-L., "A Macro Data-flow Simulator," *Technical Report*, CENG-89-27, Dept. of E.E.-Systems, University of Southern California, October 1989.