Execution of OPS5 Production Systems
on a Massively Parallel Machine

Bruce K. Hillyer
David Elliot Shaw

Department of Computer Science
Columbia University

September 1984

CUCS-147-84

Abstract

In recent years, the development of *expert systems* implemented by rule-based *production systems* has emerged as one of the dominant paradigms in the field of artificial intelligence. While production systems offer important advantages in large-scale AI applications, their use in such applications is typically very costly in execution time. In this paper, we describe an algorithm for executing production systems expressed in the OPS5 language on a massively parallel multiple-SIMD machine called NON-VON, portions of which are currently under construction at Columbia University. The algorithm, a parallel adaptation of Forgy's Rete Match, has been implemented and tested on an instruction-level simulator.

We present a detailed performance analysis, based on the implemented code, for the averaged characteristics of six production systems having an average of 910 inference rules each. The analysis predicts an execution rate of more than 850 production firings per second using hardware comparable in cost to a VAX 11/780. By way of comparison, a LISP-based OPS5 interpreter running on a VAX 11/780 typically fires 1 to 5 rules per second, while a Bliss-based interpreter executes 5 to 12 rules per second.

1 Introduction

After several decades of research on artificial intelligence, rule-based *production systems* have emerged as one of the most important and widely employed tools for the implementation of expert systems and other AI software. In general terms, a production system consists of a set of condition/action rules, or *productions*, a *working memory* representing the current "state of the world", and an *interpreter* that repeatedly executes a three-phase *cycle*:

1. *Match*. The interpreter identifies all rules whose conditions are satisfied by the current contents of working memory.

2. *Select*. One of the matching rule instantiations is selected.

3. *Act*. The working memory is modified as specified by the action part of the selected rule.

A production system organization facilitates the modular, incremental growth of knowledge bases, and allows for the useful but unplanned interaction of independently-specified rules [Winston, 1977; Nilsson 1980]. While a few production systems have already found commercial application, their use in certain other domains is precluded by slow execution speeds. This is particularly true in the case of real-time systems characterized by severe and inflexible time constraints, and in applications where high throughput is necessary to make the use of such systems cost-effective. A number of researchers [Sauers and Walsh, 1983; Forgy & McDermott, 1977; Lenat and McDermott, 1977; McCracken, 1979; Lenat et al., 1979; Buchanan, 1982; Hayes-Roth et al., 1983] have considered the problem of efficiency in the execution of production systems, and have proposed techniques to increase the speed of rule-based inferencing.

One approach to the efficient execution of production systems involves the use of parallel hardware. Forgy [1980] considered the problem of executing production systems in parallel on the ILLIAC IV, but was forced to significantly modify the production system paradigm in order to obtain reasonable performance. Stolfo and Shaw [1982] subsequently proposed a highly parallel machine called DADO, which was intended specifically for the execution of production systems; an early prototype of the DADO machine is presently operational. More recently, members of the

DADO project have investigated a number of issues related to languages and algorithms for the parallel execution of production systems [Stolfo, 1984]. The present paper describes and analyzes an algorithm for executing production systems on a parallel machine called NON-VON, which was designed not for the execution of production systems in particular, but rather, for application to a wide range of symbolic information processing tasks. A prototype of the NON-VON machine having 63 processing elements became operational in January 1985, and a larger prototype is under construction.

In particular, this paper presents an algorithm for the parallel execution of production systems implemented using the language OPS5 [Forgy, 1981], developed by Forgy and others at Carnegie-Mellon University. The algorithm may be regarded as a parallel version of Forgy's Rete Match [1982]. A LISP-based OPS5 interpreter executing the sequential Rete Match algorithm on a VAX 11/780 typically fires between 1 and 5 rules per second, while a Bliss-based interpreter executes between 5 and 12 productions per second [Gupta, 1984 (private communication)]. By way of comparison, the results presented in this paper predict that a NON-VON machine having approximately the same hardware cost as the VAX 11/780 should execute more than 850 rules per second. This result is based on measurements obtained by Gupta and Forgy [1983] of the static and dynamic characteristics of six production systems having an average of 910 inference rules each.

To establish the background for the work reported here, the next two sections will discuss the OPS5 production system language and the sequential Rete Match algorithm for production system execution, respectively. Section 4 provides an overview of the NON-VON architecture, while section 5 explicates the details of the machine configuration and performance assumptions that are used in our analysis. An algorithm for the implementation of OPS5 on NON-VON is described in section 6. Section 7 analyzes the storage requirements of this algorithm, while section 8 presents a detailed analysis of its performance characteristics. The derivation of the statistics employed in our performance analysis are presented as an appendix to the paper.

2 OPS5 Production Systems

The production system language OPS was first described by Forgy and McDermott [1977]. Several subsequent versions have appeared, with OPS5 being the most widely known. We have chosen OPS5 as the vehicle for our investigations into parallel execution of production systems for several reasons:

1. It is widely known, and has been evaluated favorably by other researchers [Hayes-Roth et al., 1983].

2. It has been used to implement a large and successful commercial production system [McDermott, 1980].

3. Static and dynamic characteristics of several OPS5 production systems have been measured [Gupta and Forgy, 1983].

4. Its speed can be increased significantly by parallel execution, even though the language was designed for sequential processing.

It should be noted, however, that other researchers [Miranker, 1984a] are actively engaged in the development of a production system language specifically designed for parallel execution; such a language may well prove better suited to the capabilities of parallel machines.

The essential elements of the OPS5 language are outlined below; a more complete exposition can be found in [Forgy, 1981]. By way of illustration, Figure 1 shows a pair of productions whose execution results in the printing of a sorted list of all numbers in working memory.

Insert Figure 1 (Example OPS5 Productions) here.

A corresponding set of sample working memory elements is presented in Figure 2; they specify that the current task is to sort, that the output counter is 0, and that there are three numbers to be sorted: 17, 5, and 23.

Insert Figure 2 (Example Working Memory Elements) here.

A rule expressed as a production in OPS5 consists of a *production name*, a conjunction of (possibly negated) clauses known as *condition elements*, and an arrow

followed by one or more *actions*. The condition elements of a production are collectively known as the *condition* or *left-hand side* (LHS) of the production. Similarly, the actions are the *right-hand side* (RHS). Each condition element consists of a *class name* and one or more *terms*. The class name is the first item in the condition element. Each term consists of an *attribute name*, a *relational operator*, and a *value*. Attribute names are prefixed with an up-arrow. Although attribute-value pairs are the usual form of expression, OPS5 conditions may be expressed in a positional notation by omitting attribute names. Common actions in OPS5 write values to output, and remove, modify, or create facts in the working memory.

Numeric and string values, which may be constants or variables, occur in OPS5. Variables are denoted by enclosing the name in angle-brackets. The permitted relational operators are $<$, $<=$, $>$, $>=$, $=$, $<>$, and $<=>$. The first six have their usual meanings, but only $=$ and $<>$ may be applied to string values. The operator $<=>$ evaluates to *true* provided an attribute and value are of the same type. If no operator is explicitly specified in a term, $=$ is assumed. OPS5 has grouping operators to express conjunctions and disjunctions of multiple terms involving an attribute.

A working memory element is similar in form to a condition element, but contains neither operators nor variables, since it expresses a specific fact about the world modeled by the production system. Each working memory element is assigned a unique 32-bit integer *time-tag* upon creation. The tag serves as a compact identifier for the working memory element, and also permits the distinction of current facts from old information. A working memory element is said to *match* a condition element provided all the constraints specified by relational operators hold. The left-hand side of a production is said to be *satisfied* provided that:

1. For every non-negated condition element there exists a working memory element that matches it.

2. For every negated condition element, there does not exist a working memory element that matches it.

3. Each variable is bound to the same value in all occurrences.

An OPS5 interpreter executes a production system by cycling through the following three-step process, halting before the third step if no production is satisfied by the current working memory.

1. *Match* the working memory elements with the conditions of all productions: Each ordered tuple of working memory elements satisfying the corresponding non-negated condition elements of a production is called an *instantiation* of that production. The collection of all such instantiations is called the *conflict set*.

2. *Select* one instantiation from the conflict set according to certain predefined criteria. This step is known as *conflict resolution*. Conflict resolution strategies provided in OPS5 favor instantiations containing recent information, and prefer productions having restrictive conditions. The former tends to focus the system's attention on one task at a time, and the latter applies special case rules in preference to general ones.

3. *Act* on the chosen instantiation by performing the actions specified in the production's right-hand side. These actions perform input and output, and modify the contents of working memory. The modifying actions can *make* new working memory elements, *modify* one or more terms in some working memory elements in the instantiation, and *remove* elements of the instantiation from working memory. Performing the actions specified by a production is sometimes called *firing* the production.

3 The Rete Match Algorithm

Of the three steps in the production system cycle, the matching phase has proven in practice to be the most time-consuming. According to Forgy [1979], more than 90% of the execution time in a uniprocessor implementation is consumed by matching. A naive implementation of the interpreter would match the condition part of each rule in turn against the entire contents of the working memory. Forgy's Rete Match algorithm exploits his observation that firing an OPS production causes only a few changes to working memory, and that these changes have few effects on the conflict set. Hence a computational savings results if the production system is compiled into a dataflow graph, with state information saved at each node during execution. A change to working memory is entered into initial nodes of the graph. Consequent state changes then propagate through the graph, updating information stored in intermediate nodes. State changes in terminal nodes of the graph represent changes to the conflict set. Figure 3 shows an example

dataflow graph (as used on NON-VON) corresponding to the production named sort-work that is depicted in Figure 1.

Insert Figure 3 (Example Dataflow Graph for NON-VON) here.

The graph can be viewed as a collection of tests that progressively determine which productions are ready to fire. First, the *intra-condition tests* check that attributes in a working memory element satisfy relational operators, and that variables occurring more than once in a condition element are bound consistently. Any working memory element satisfying all intra-condition tests for a condition element is stored as a *token* in the α-*mem node* corresponding to that condition element. Subsequently, *inter-condition tests* are performed in *two-input nodes* to verify consistent binding of variables across multiple condition elements in a production's left-hand side. This testing occurs in *AND-nodes* for non-negated condition elements, and *NOT-nodes* for negated condition elements. At the output of each AND-node and each NOT-node in the graph is a β-*mem node* to store tokens. A token in a β-mem node represents an ordered tuple of working memory elements that jointly satisfy all non-negated condition elements that are ancestors of that node.

The intra-condition tests are local, in that each examines terms of only one working memory element. Entry of a token into an α-mem or β-mem node triggers the more complex inter-condition testing, which proceeds as follows. First, the two-input node following the memory node is identified. Second, the *opposite* memory node that serves as the other input is located. Third, the new token is matched with *all* members of the opposite node to test for consistent variable bindings, in accordance with the type of two-input node. If consistent bindings are found, the output tokens from this two-input node are formed, and they become new entries to the subsequent β-mem node. In the case of terminal two-input nodes, the result is an addition to (or deletion from) the conflict set.

4 The NON-VON Machine

This section outlines the essentials of the general NON-VON architecture, in support of the analysis of section 8. A fuller description of the architecture is found in [Shaw, 1982] and [Shaw and Sabety, 1984]. Although all portions of the general machine architecture are mentioned here for completeness, only certain subsystems are required to execute the algorithms described in this paper. Section 5 presents the reduced configuration assumed for OPS5 production system execution.

The top-level organization of the general NON-VON machine is illustrated in Figure 4.

Insert Figure 4 (Organization of the NON-VON Machine) here.

NON-VON has two principal components, known as the *primary processing subsystem* and the *secondary processing subsystem*. NON-VON is connected to a *host machine*, a general purpose computer serving as a front end device for interactions with the user.

The primary processing subsystem is organized as a binary tree. It consists of a large number of *small processing elements* (SPE's), each having an 8-bit ALU, a very small RAM, and communication connections to three neighboring SPE's, which are known as the *parent*, *left child*, and *right child*. In addition, each SPE is capable of communicating, within a single instruction cycle, with two additional SPE's, called the *left neighbor* and *right neighbor*. These neighbors are the predecessor and successor in an inorder traversal of the primary processing subsystem tree. Each leaf node in the tree is also connected by bit-serial lines to four other leaves known as the *North*, *South*, *East*, and *West* neighbors, providing efficient support for an orthogonal mesh-connected communication topology. (The mesh connections are not used in the execution of OPS5, however.)

Each SPE (as currently fabricated) contains a local RAM consisting of a 64 x 8-bit section and a 64 x 1-bit section. A prototype chip containing eight SPE's is described in detail in [Shaw and Sabety, 1984]. The SPE's do not store programs locally, but instead receive instructions that are broadcast to them from some

higher level in the primary processing subsystem tree, as described below. This mode of processing was named single instruction-stream, multiple data-stream (SIMD) by Flynn [1972].

In the top five to ten levels of the primary processing subsystem, each SPE is connected to a *large processing element* (LPE). The LPE's are general-purpose microcomputers having large RAM's, and supporting locally stored programs. Unlike the SPE's, the LPE's are capable of operating asynchronously in multiple instruction-stream, multiple data-stream (MIMD) mode [Flynn, 1972]. In particular, LPE's at the roots of several subtrees of the primary processing subsystem (possibly at different levels) can broadcast separate instruction streams to be executed simultaneously by all SPE's below them, giving NON-VON the capability for what is sometimes referred to as *multiple-SIMD* execution. Each LPE also has an *active memory controller* to generate control signals and to cache instructions and data for its subtree of SPE's.

The LPE's are connected by a high-bandwidth interconnection network. For moderate numbers N of LPE's (say, $N < 128$), a two-stage *root-point network* consisting of $N^{1/2}$ x $N^{1/2}$ crossbar switches gives lower latency than a $\log(N)$-stage 2 x 2 crossbar network such as a butterfly or omega, at comparable cost. The use of such a high-bandwidth network is essential to a number of NON-VON algorithms involving large collections of data. The algorithms presented in this paper, however, do not make use of the LPE network.

The secondary processing subsystem incorporates a substantial number (perhaps 32 to 256) of disk drives. Each drive is connected via an *intelligent head unit* to an LPE in the primary processing subsystem, forming a very high bandwidth interconnection between these two subsystems. In addition to ordinary disk I/O, intelligent head units can perform certain computationally simple data filtering and hashing operations "on the fly", passing results to the associated LPE's. In the production system algorithms reported in this paper, the secondary processing subsystem is not needed, as all the required data (even for rather large production systems) can fit within the primary processing subsystem.

An early prototype of the NON-VON architecture has been operational at Columbia since January, 1985. This prototype, called NON-VON 1, contains 63 SPE's, each of which embodies some, but not all of the features described above, and one VAX 11/750 that serves as the sole LPE and host. A larger, significantly enhanced prototype called NON-VON 3 is currently under construction. This machine will embody 8,191 SPE's, again operating under the control of a single VAX 11/750 The machine is being implemented using 3 micron custom nMOS chips, each containing four SPE's, which were developed using the MOSIS "silicon brokerage" system at ISI.

5 Configuration and Performance Assumptions

A large-scale NON-VON primary processing subsystem might comprise as many as a million SPE's, together with a thousand or more LPE's. To execute production systems, however, a much smaller machine will suffice. In particular, we have assumed a primary processing subsystem comprising 16K SPE's for purposes of the analysis presented in this paper. The system is assumed to contain 32 LPE's, all associated with the fifth level of the tree; the 31 LPE's that would be associated with the first through fourth levels in a general NON-VON machine are not required for the execution of the production system algorithm. We assume that each SPE contains 64 bytes of RAM, as is the case in the current NON-VON design. Such a configuration would embody 4096 integrated circuit chips for the SPE's and 640 chips for the LPE's, assuming that 20 chips are required to implement each LPE. We also assume a dedicated host, together with a bus (which need not be as fast as that which would be incorporated in a general NON-VON machine) connecting the host to the LPE's. We assume the LPE's and host to be capable of executing three million instructions per second, a figure chosen to correspond roughly with the performance of 32-bit microprocessors such as the AT&T 32100 and the Motorola M68020.

Figure 5 depicts the reduced NON-VON configuration assumed for production system execution.

Insert Figure 5 (NON-VON: Reduced Configuration for OPS5) here.

NON-VON uses a two-speed clock. The short clock period is for the broadcast of an instruction to all SPE's through a high-fanout tree implemented in fast bipolar logic, and the execution of that instruction. The long clock period permits a signal to propagate through combinational logic from the root of the tree to the leaves, and back to the root again. There are two special instructions that require this long communication step. The RESOLVE instruction requires two long clock periods to identify the first (in an inorder enumeration of the tree nodes) of an arbitrary collection of SPE's having a certain flag register set. The *linear neighbor communication* instructions require two long clock periods to permit all SPE's to communicate simultaneously with their predecessors or successors in an inorder traversal of the binary tree. On the basis of preliminary chip tests and calculations for a tree of 16K SPE's, we assume a period of 350 ns. for the fast clock (30 ns. broadcast + 320 ns. execution) and 3 us. (100 ns. per level) for the slow clock. In the analysis of performance given in section 8, the fast and slow clock periods are counted separately.

The following three sections of the paper describe an algorithm and performance analysis for the execution of OPS5 on NON-VON. Considerable detail is given, to show how a heterogeneous massively-parallel machine can be applied to this task, and to provide the reader with a basis for assessing the performance figures derived in section 8.

6 Execution of OPS5 on NON-VON

This section presents observations about the potential parallelism embodied in OPS5, a description of how the Rete Match is processed on NON-VON to exploit the identified parallelism, and an example to clarify this processing.

Our algorithm has its roots in Algorithm 3 of [Gupta, 1984]. This algorithm was designed for the DADO machine [Stolfo and Shaw, 1982], in a configuration consisting of 1023 identical processors. Our algorithm is designed for a NON-VON configuration having 32 large processing elements, each somewhat more powerful than a DADO PE, and 16K small processing elements for a greater degree of associative parallelism. The following discussion presents the rationale for our approach to this problem. Since the experimental implementation of an OPS5

interpreter for NON-VON comprises more than 1500 lines of LISP code, we describe portions of the processing relevant to the performance analysis without formally specifying details of the entire algorithm.

6.1 Parallelism in OPS5

As discussed previously, the execution cycle for an OPS5 production system has three steps: match, select, and act. In the Rete match algorithm, the match phase has two components: the highly local intra-condition testing, and the subsequent inter-condition testing that evaluates a dataflow graph to combine previous results and save state in α-mem and β-mem memory nodes.

Three levels of potential parallelism can be identified in this execution cycle:

1. The intra-condition testing can be performed in a massively parallel manner using associative processing techniques. This has been previously noted by Stolfo and Shaw [1982]. The NON-VON production system algorithm guarantees very rapid completion of this step, in time dependent on static characteristics of the production rules. This contrasts with other implementations that depend on hashing techniques to control the amount of matching. Massive parallelism is also applicable during the deletion of facts from working memory. NON-VON simultaneously finds all instances of a fact in all α-mem and β-mem nodes, and removes all affected memory tokens simultaneously.

2. There is a modest amount of potential concurrency in the evaluation of inter-condition testing in two-input nodes of a Rete dataflow graph. This has been observed by Gupta [1984], who recommends partitioning the production rules into 32 subsets (based on the same empirical data on which our own analysis is based) to exploit parallelism in this phase. In [Oflazer, 1984], it is determined that for two specific production systems, the maximum available parallelism factor in the inter-condition testing is approximately 7. The generality of this result is unknown.

3. We find no significant parallelism in the select and act phases, although substantial portions of the act phase can be overlapped with the following match phase.

The heterogeneous architecture of the NON-VON machine is well suited to the exploitation of these varying degrees of parallelism.

1. The intra-condition testing is performed in the SPE's in two massively parallel SIMD computation steps. The first step simultaneously evaluates

individual terms of all condition elements. The second step determines the satisfaction of all condition elements by a parallel communication in time proportional to the number of terms in the longest condition element.[1] The synchronous nature of SIMD execution was found not to limit the rate of processing in this phase.

2. The moderate parallelism of the inter-condition testing is done in 32 LPE's, with NON-VON operating under a partitioned-SIMD execution discipline. Pure SIMD processing would have been a serious constraint during this portion of the execution, since evaluation of the Rete dataflow graphs presents a high degree of data sensitivity. The use of subtrees of SPE's as *active memories* enhances the throughput of the LPE's by providing a fast associative search capability.

3. The select and act phases, which have little inherent parallelism in OPS5, are performed in a single relatively fast host processor, although the LPE's and SPE's do some "bookkeeping" and overlapped processing for the next match phase at this time.

The overlapped processing is as follows. During the action phase of the production system cycle, the host executes the right hand side of the selected instantiation. This commonly results in the addition of facts $f_1...f_k$ to working memory. For each f_i the host assigns a time-tag for identification and converts attribute values to tokens to obtain the *working memory token* t_i. Each t_i is installed in a table of working memory elements in the host, and is transmitted to the LPE's for use in the next matching phase of the production system cycle. The matching phase for t_1 starts in the LPE's and SPE's while the host asynchronously creates and transmits $t_2...t_k$. With the exception of the time for t_1, the host processing for an addition to working memory overlaps matching, and does not contribute to the running time of the algorithm. Similarly, LPE's asynchronously finish the matching phase for each t_i, depending on the amount of activity in the dataflow graph of each partition. Thus the host receives conflict set changes from LPE's that finish early, overlapped with continued matching in other LPE's. Only one synchronization point occurs in the production system cycle: to be consistent with

---

[1]This can be improved to time logarithmic in the number of terms in the longest condition element with a worst-case 50% decrease in SPE utilization, by techniques closely related to the allocation schemes for database records described in [Shaw and Hillyer, 1982].

the semantics of OPS5, all changes in the conflict set must reach the host prior to the completion of conflict resolution for the next cycle.

6.2 Description of Processing

This section gives an overview of the procedures executed by the host, LPE's, and SPE's during the execution of an OPS5 program, and the following section presents a concrete example. Salient steps are detailed in the analysis of section 8.

The host processor is responsible for controlling the overall computation and for communicating with the user. Prior to the commencement of execution, the host obtains a collection of production rules from the user (or from disk, at the direction of the user), partitions them into subsets as described further below, and sends one subset to each LPE. Each subset is compiled by its LPE into a dataflow graph. These graphs are similar to those of the sequential Rete Match algorithm, but the dataflow graph used by NON-VON is smaller, with input nodes representing entire condition elements, rather than individual attribute constants and variables. This increases parallelism during each execution cycle of the production system: all condition elements are tested in one parallel step by the 16K SPE's before the 32 LPE's commence their dataflow graph processing.

Host processing during the three phases of the production system execution cycle proceeds as follows:

1. During the match phase, the host receives messages from the LPE's, which report changes to the conflict set. In our implementation the host maintains the conflict set as a list[2] sorted by the OPS5 conflict resolution criteria.

2. During the select phase, the host chooses a production instantiation to be fired. In our implementation this amounts to removing the item at the head of a sorted list.

3. During the act phase, the host executes I/O specified in the right hand side of the chosen production, creates new working memory tokens to represent facts to be added to working memory, and broadcasts messages to the LPE's. Each message contains a working memory token to be added or deleted.

---

[2]Average length 16 (Appendix, item 5).

Each LPE is responsible for a subset of the production system rules and for a subtree of SPE's. The LPE generates an instruction stream for the SPE's, evaluates the dataflow graph for the production system partition stored in that subtree, and obtains the resulting conflict set changes.

In particular,

1. During the match phase, each LPE broadcasts SIMD code to its SPE's to associatively determine additions and deletions to the set of condition elements that are satisfied. A list of these changes is built. Each addition is represented by a new memory token which, according to the Rete technique, is stored in an α-mem or β-mem node in the dataflow graph. This is implemented by associatively locating an available SPE and storing the token there. Insertion or deletion of a token T in a (non-terminal) α-mem or β-mem node M is followed by evaluation of the two-input node N that follows M in the dataflow graph. The evaluation of N is implemented as follows. First the LPE looks in its table of two-input nodes to determine the memory node M' that is the other input of N. Next, an associative probe is performed in all SPE's in the subtree to locate tokens in M'. An associative match is performed to discover which tokens T'_i in M' have all their variables bound consistently with the variables in T. The final step in evaluating N depends on whether N is of type AND or NOT, and on whether T is a left or right input to N. Suppressing details, we state that in most cases the LPE retrieves matching tokens T'_i, and concatenates them in turn with T to form new memory tokens T''_i that are placed in the memory node at the output of N. Changes in terminal memory nodes of the dataflow graph represent changes to the conflict set; these changes are reported to the host as they are detected.

2. During the select phase, which is just long enough for the host to extract the first element of its conflict set list, the LPE's are idle.

3. During the act phase of the host, the LPE's receive messages from the host that contain working memory tokens and commands to add or delete the tokens from working memory. These tokens are stored in appropriate LPE tables, and are processed as described above: the matching phase for LPE's and SPE's is overlapped with the act phase of the host.

The SPE's in a particular LPE-rooted subtree serve as an active memory. They contain the condition element terms of compiled productions, and the token memory for the Rete dataflow graph. The SPE associative processing capability facilitates

the rapid, highly parallel evaluation of condition elements with respect to new working memory elements, as well as the parallel evaluation of the two-input nodes in the dataflow graph evaluation process.

Specifically,

1. During the match phase, tokenized attribute-value pairs are broadcast to the SPE's by their controlling LPE. The SPE's, which each contain one term of a condition element, then simultaneously evaluate the satisfaction of all terms. Next, chains of adjacent SPE's communicate in parallel to determine the satisfaction of entire condition elements, and the ID's of satisfied condition elements are retrieved by the controlling LPE. During the two-input testing in the Rete dataflow graph, all SPE's containing a token for the relevant memory node simultaneously check the consistency of variable bindings, and the LPE retrieves successful matches. Deletion of a working memory element token T causes all concatenated β-mem tokens that contain T to become unsupported; they are all associatively found and removed by the SPE's in one highly parallel step (with additional matching required when a token is deleted from the right input of a NOT node.

2. During the select phase the SPE's are idle.

3. During the act phase, the SPE's associatively find and delete the token for the production instantiation that has been fired.

6.3 Example of OPS5 Data and Processing

This section presents an example of a condition element based on a simple AI "blocks world", shows how it is stored in NON-VON SPE's, and describes how it is tested on a sample working memory element.

The following condition element will match a block that has no square faces, saving the edge lengths in the variables $<l>$, $<w>$, and $<h>$. It ensures that the width of the block is different from the length, and that the height is different from the length and width. Note that the first occurrence of a variable simply binds its value, while later occurrences refer to this value for comparison. Braces enclose conjunctions of terms pertaining to an attribute.

```
(block ^length <l>
       ^width  { <w> <> <l> }
       ^height { <h> <> <l> <> <w> })
```

This condition element occupies 4 NON-VON SPE's, as depicted in Figure 6. Each SPE holds a comparison operator and two 32-bit integers, which may be either constants, or variables that are filled by attribute value tokens at runtime.

        Insert Figure 6 (Condition Element Stored in Four SPE's) here.

To do the matching of a working memory element, for instance (block ^length 3 ^width 4 ^height 5), two main steps are required. In the first step, the truth or falsity of each term is determined by SIMD execution of code that each LPE broadcasts to the SPE's in its partition. The operations performed for this example are:

1. Associatively probe for all SPE's having a ^classname variable, and broadcast the token for "block" to them.

2. Associatively probe for all SPE's having a ^length variable, and broadcast the value 3 to them.

3. Associatively probe for all SPE's having a ^width variable, and broadcast the value 4 to them.

4. Associatively probe for all SPE's having a ^height variable, and broadcast the value 5 to them.

5. In parallel, evaluate the stored relational operator on the two stored values in all SPE's.

In the second step, the truth or falsity of inter-condition tests for entire condition elements is determined as follows:

1. Associatively locate all SPE's that hold the first term of a condition element.

2. In parallel, send the boolean result of the comparison in the first terms to the SPE's having second terms of condition elements, where logical conjunction is performed. Now the second terms contain an indication of whether both the first and second terms of a condition element were satisfied.

3. Send this result in parallel to SPE's containing the third terms, where logical conjunction is performed again.

4. Continue for a number of steps one less than the largest number of terms in the longest condition element. Any SPE holding the last term of a condition element will then contain TRUE or FALSE for the entire condition element.

After the intra-condition testing has been performed in a partition, the LPE associatively enumerates the ID's of condition elements that were satisfied. In NON-VON, this enumeration occurs in time proportional to the number that are satisfied, independent of the total number of condition elements. The expected number of satisfied condition elements per partition is less than one (Appendix, item 4). The LPE places α-mem tokens for satisfied condition elements on a stack of pending *memory node additions*.

The memory node additions in a partition are handled sequentially by the controlling LPE. Since the expected number of node additions per partition. is less than one, this is both fast and economical.

To perform a memory node addition, the LPE performs several steps. The LPE broadcasts SIMD code to associatively allocate SPE's to hold the token, and then code to store the token. Then the LPE determines which two-input node is triggered by that memory node addition. The two-input node is evaluated in two steps. First, the opposite input memory for the node--a distributed set of tokens stored in SPE's--is activated by an associative search. Second, an associative match of relevant variable bindings is performed in parallel between the new token and all members of the opposite memory. Any consistent bindings are discovered by an associative probe. Portions of successfully matched tokens are reported to the LPE, and new memory node additions/deletions are placed on the stack.

The LPE recognizes an insertion of a token into any memory node at the bottom of the Rete network to be the addition of a production instantiation to the conflict set. Such a token is also reported to the host for conflict set resolution.

When all pending memory node additions have been processed, the LPE sends a completion signal to the host, which performs conflict resolution when all LPE's are finished. A centralized algorithm is reasonable for this step since the average size of the global conflict set is 16 (Appendix, item 5), and the average number of

changes to the conflict set is 5.3 per firing cycle (Appendix, item 6). The host then executes the right hand side actions of the chosen production, sending messages to the LPE's to effect changes to working memory.

6.4 Partitioning Production Systems

In the production systems examined in Gupta and Forgy [1983], approximately 30 (Appendix, item 3) condition elements are satisfied by a typical single addition to working memory. As noted earlier, Gupta [1984] suggests obtaining parallelism by dividing OPS5 production systems into 32 partitions that execute concurrently. The problem of obtaining a suitable partitioning is separable from the problem of executing the resulting partitions, and as such is beyond the scope of this paper. Relevant to the present discussion, however, is the fact that the goal of partitioning is to distribute the two-input node testing uniformly over the partitions, despite the fact that cascading of two-input node activations through the Rete dataflow graph is necessarily sequential. Although results have been reported in [Ishida, 1984] and [Oflazer, 1984], the partitioning problem remains an area for future work.

A "good" partitioning method is assumed to exist, where the meaning of "good" is determined by the statistical parameters given in the performance analysis section, as justified in the appendix. Somewhat similar assumptions have previously been adopted by Gupta [1984] and Miranker [1984b].

During execution, a partitioned production system could exhibit transient over-concentration of α-mem and β-mem tokens in individual partitions, requiring significantly more storage than for the average case. It remains to be seen whether this is a problem in practice. One could speculate that such over-concentration, should it occur,

- results from a production system programming style, or
- is due to the nature of the OPS5 execution semantics, or
- is the consequence of a particular partitioning algorithm, or
- is an unavoidable element of partitioned execution of production systems.

These possibilities have greatly differing implications for appropriate remedies. We are not aware of any results reported in the literature that illuminate these issues, and thus regard the (potential) problem of over-concentration as an open question.

7 Hardware Capacities Required

The storage required in an SPE is 24 bytes plus 18 1-bit flags. This represents space for one term of a condition element, one α-mem token or portion of a β-mem token, and one relevant binding. More specifically, the byte space is allocated as follows:

- Two attribute name ID's (1 byte each)
- Two attribute-value tokens (4 bytes each)
- A condition element ID (2 bytes)
- An α-mem or β-mem node ID (2 bytes)
- A working memory element ID stored in an α-mem or β-mem node (4 bytes)
- A relevant binding value or conflict set member ID (4 bytes)
- The ID number of the SPE (2 bytes).

The 1-bit flags mark subsets of SPE's that contain:

- A condition-element term
- The first term of a condition-element
- The last term of a condition element
- An α-mem or β-mem token
- The first cell of a token
- The last cell of a token
- A working memory element ID in a token
- A relevant binding in a token
- The type (string or numeric) of the first value
- The type of the second value
- The type of the relevant binding
- A condition element that feeds the right-hand input of a NOT node
- A member of the conflict set
- Compare for equal
- Compare for not equal
- Compare for less
- Compare for less-or-equal
- Compare for same type

Thus the 64 byte RAM of current NON-VON SPE's is of sufficient capacity to store a term-evaluating node as well as a memory node and relevant binding. The extra RAM could hold more tokens (but with a decrease in execution speed), or possibly data such as the dataflow graph connections that would otherwise be stored in the LPE's.

The average of the production systems examined has 4 condition elements per production (Appendix, item 7) and 3 intra-condition tests per condition element (Appendix, item 10). Thus the number of SPE's required for production memory in NON-VON is about 12 times the number of productions. The average of the production systems examined has 910 productions (Appendix, item 11), requiring 10,920 SPE's for condition element terms. The analysis presented below assumes 16K SPE's. Quadrupling the number of SPE's would permit systems of approximately 5400 rules to be executed, but might decrease the clocking speed by as much as 10 percent, absent technological compensation.

The maximum aggregate number of tokens in α-mem and β-mem nodes is about 4600 (Appendix, item 12), and the average token requires 2 SPE's (Appendix, item 21); thus, about 9200 SPE's are required to store the tokens. A NON-VON having 16K SPE's is sufficient. Note from the storage allocation described previously that each SPE can hold both a condition element term and part of a token at the same time. Note also that the number of tokens stored in an SPE could be increased if necessary, although the execution speed would be reduced somewhat.

A host storage capacity of one megabyte would accommodate the following:

1. A symbol table of string tokens.

2. The current members of the conflict set.

3. Working memory elements with time tags indicating order of creation.

4. Right-hand sides of productions, indexed by production ID's.

5. Software for the interpretation of OPS5, including communication with the user and with the LPE's.

We estimate that 256K bytes would be sufficient to store the data and code required by an LPE, which includes:

1. Working memory elements with time tags indicating order of creation. (This duplicates information stored in the host, but reduces the need for communication.)

2. Tables encoding the Rete dataflow graph.

3. Precompiled sequences of SPE instructions that the LPE will command the active memory controller to broadcast into the subtree, to perform data storage, associative matching, and associative retrieval.

4. Procedures to be executed by the LPE in performing its portion of the production system work, and for compiling productions into the dataflow graph.

8 Performance of Rete Match on NON-VON

Execution speeds of 1 to 12 cycles (or "rule firings") per second on a VAX 11/780 are typical for OPS5 systems of the size analyzed in this paper [Gupta, 1984 (private communication)]. Using data from [Gupta and Forgy, 1983], we obtain below a predicted execution speed for NON-VON of 861 production firings per second. The formulas suggest that NON-VON's advantage may increase as production systems become larger. Intuitively, NON-VON is insensitive to the total number of objects because of its associative processing capabilities.

We have written and tested on an instruction-level simulator an experimental compiler and runtime system for the execution of OPS5 on a one-LPE NON-VON. By examining the NON-VON instructions, we determine the number of slow and fast clock cycles required for each of the SIMD processing steps, as a function of parameters of the production system. Adding approximate overhead values for non-overlapped execution in LPE's and the host[3] gives the time required per production firing cycle[4], and hence the rate of production system execution on NON-VON. These calculations are presented in three sections below. The first derives the time required for an addition to working memory, the second calculates the time for a deletion, and finally, the overall time including conflict resolution is obtained.

In the following analysis, F denotes the fast clock period for the SPE's, S denotes

---

[3]The figures given for host and LPE processing are estimates, unlike the SPE figures, which are derived from actual code.

[4]We assume that the only actions in the right-hand sides of productions are additions, deletions, and modifications of working memory elements. The right-hand side of a production expressed in OPS5 can cause arbitrarily large amounts of I/O and can call any function written in LISP, but the time consumed by such operations does not give information about the performance of the production system inferencing engine.

the slow clock period for the SPE's, and H denotes the average instruction time for the LPE and host microprocessors.

8.1 Addition to Working Memory

The analysis for an addition to working memory will be carried out in six parts.

1. The time for broadcasting the attribute-value pairs of a new working memory element.

2. The time for intra-condition testing.

3. The time for processing α-mem node additions.

4. The time for processing β-mem node additions.

5. The time for evaluating two-input node tests.

6. The time for processing that may arise upon deletion of a token from the right-hand input of a NOT node.

The first step in processing a working memory addition is to store it into a table in the LPE, assigning a time-tag. Next, the LPE broadcasts the attribute-value pairs. For each term, the attribute ID is associatively matched with the attribute ID's stored in all SPE's. Then the value of the attribute is broadcast and stored in parallel by those SPE's for which the attribute ID matched. The time required is given by

$$T_{broadcast} = 26F \times nAttr_{class} + 5F + LPE_{broadcast}$$

where

$nAttr_{class}$, the number of attribute-value pairs in this working memory element's class, is 11.4 (Appendix, item 14), and

$LPE_{broadcast}$, the number of non-overlapped LPE and host instructions, is simply assumed to be 60. This reflects the time required to store into the LPE's table and assign a time tag.

Thus $T_{broadcast} = 301F + 60H$.

The next step is the actual intra-condition testing. First, the two values stored in an SPE containing a term are compared. Second, the success of the comparison relative to the stored relational operator is determined. Third, the conjunction of the terms in a condition element is evaluated in time proportional to the longest condition element. Finally, matching condition element ID's are reported out, and tokens for corresponding α-mem node additions are stacked. The time required is given by

$$T_{match} = 86F + (maxTerms - 1) \times (3F + 2S)$$
$$+ \text{ successes} \times (6F + 2S) + LPE_{match}$$

where

maxTerms, the largest number of terms in a condition element, is 9 (Appendix, item 15),

successes, the maximum number of condition elements that are satisfied in any partition, on the average, is 3 (Appendix, item 4),

LPE_{match}, the number of non-overlapped LPE instructions to construct and stack tokens resulting from successes is simply assumed to be 40.

Thus $T_{match} = 128F + 22S + 40H$.

The processing required for an α-mem node addition includes removing the top entry (a node ID and a working-memory element ID) from the stack of pending additions, looking up relevant binding indices from the Rete net array (subscripted by node ID), copying relevant binding values from the current working memory element (array access), selecting an available SPE to hold the token and bindings, if any, and broadcasting the token into the SPE for storage. The time required for the α-mem node additions resulting from an addition to working memory is given by

$$T_{\alpha\text{-mem}} = \text{nAdds} \times (\ 21F + 2S + (\text{tokLen} - 1) \times (14F + 4S) + \text{wmIDs} \times 4F$$
$$+ \text{relBinds} \times 6F + \text{strToks} \times 1F + 3F \text{ if rhsNOT})$$
$$+ LPE_{\alpha\text{-mem}}$$

where

nAdds, the maximum number of α-mem additions in any partition, is 3 (Appendix, item 4), on average,

tokLen, the average length for an α-mem token, is 1 (Appendix, item 17),

wmIDs, the number of working memory id's stored in a token, is 1 for α-mem, since there is just one condition-element above each α-mem node,

relBinds, the number of relevant bindings in a token, is 1 (Appendix, item 16),

strToks, the number of relevant bindings that are string tokens rather than numeric tokens, is assumed to be half of relBinds, or 0.5,

the term "3F if rhsNOT" is for setting a flag if this token is in a memory node that is the right-hand input of a NOT node. This flag facilitates rapid processing of deletions, as discussed later. The entry of a token into the right-hand input of a NOT node is relatively rare, as discussed in (Appendix, item 18). Thus the total contribution of the strToks and rhsNOT terms is assumed to be 1 fast cycle.

$LPE_{\alpha\text{-mem}}$, the number of LPE instructions to perform the appropriate array accesses, is simply assumed to be 20.

Thus $T_{\alpha\text{-mem}} = 96F + 6S + 20H$.

The processing and formula for additions to β-mem nodes is the same as for α-mem nodes. The parameter values that differ are:

nAdds is 2 (Appendix, item 19),

tokLen is 3 (Appendix, item 17),

wmIDs is 3 (Appendix, item 13).

Thus $T_{\beta\text{-mem}} = 136F + 20S + 20H$.

The memory allocation scheme that associatively finds available SPE's to hold tokens has the same behavior as the classical first-fit algorithm. Since most allocation requests are for token space of size 1, and none are for more than 10 or so, first-fit should behave well, and the need for compaction of free space should be rare, unless almost all of the memory is occupied. Although the addition of a token to a β-mem node can trigger an incremental memory space compaction if token memory has become fragmented and is nearly full, we assume that compaction is sufficiently infrequent to be negligible. Support for this is given by the 2/3 rule, also known as the "50% rule" [Knuth, 1968], together with the observation (section 7) that token memory is less than 60% full when running the production system analyzed here. Experimental examination of actual token memory behavior in partitioned OPS5 production systems remains an area for future work.

An addition of a token to an α-mem or β-mem node initiates the processing of a two-input node in the Rete dataflow graph. The purpose of a two-input node is to evaluate relational operators that reference variables bound in previous condition elements. The right-hand input of a two-input node is from a single condition element, the "current" one. The left-hand input is from other condition elements of the production. The two-input node examines variables that are bound in previous condition elements and referenced for comparison in the current one. The values of such variables are *relevant bindings* for this two-input node. In summary, an addition to an α-mem or β-mem node causes comparison of relevant bindings with tokens stored in the opposite input of the two-input node.

The LPE performs array accesses to determine the node ID of the opposite input, to identify the comparison operators for this node, and to extract the current

binding from the token whose addition triggered this processing. An associative probe activates all tokens in the opposite node. The current binding and comparison operators are broadcast to those tokens, and associative matching identifies tokens having all comparisons satisfied[5]. In the case of AND nodes, the working memory element ID's are retrieved from satisfied tokens, and new tokens are formed and stacked for addition to the β-mem node that receives the output of this two-input node. Processing a right-hand input to a NOT node is initially similar to that for an AND node, but after working memory ID's are retrieved from satisfied tokens, deletions (rather than additions) are stacked for further processing, as accounted for separately below. In contrast, upon addition to the left-hand input of a NOT node, the input token is copied to the output β-mem node only if no opposite tokens were satisfied.

$$
\begin{aligned}
T_{\text{two-input}} = \\
\text{nAdds} \times \{\ P_e \times (8F + 2S) \\
+ P_o \times [\ \text{nBind} \times (\ 2F + 2S + \{\ 4F \text{ if } <=> \text{ string} \\
5F \text{ if } <=> \text{ number} \\
10F \text{ if } =,<>,<,<=,>,>= \ \}) \\
+ (11F + 2S) \\
+ \text{ if not LHS input of NOT node, then} \\
7F + 2S + \\
\text{nSuccessfulMatch} \times (4F + 2S + \text{tokLen} \times (7F + 2S)) \\
+ \text{ if nBind} > 1 \text{ then } 4S \times [(\text{nBind} - 1) \text{ DIV } 2] \\
+ \text{ if nBind even then } 1F + 2S \quad]\ \} \\
+ LPE_{\text{two-input}}
\end{aligned}
$$

where

nAdds, the maximum number of α-mem plus β-mem additions in any partition, is 5 (Appendix, items 4, 19) on average,

P_e, the probability that the opposite input memory is empty,

---

[5]The idea of storing relevant bindings in tokens to facilitate associative matching is found in [Gupta, 1984].

is 0.7 (Appendix, item 20)

P_O, the probability that the opposite input memory has tokens, is 0.3 (Appendix, item 20)

nBind, the number of relevant bindings in a token, is 1 (Appendix, item 16),

it is assumed that the computational savings for the $<=>$ operator is never obtained,

it is assumed that the computational savings for entry to the LHS of NOT two-input nodes is never obtained,

nSuccessfulMatch, the total number of successful matches during the comparison, is 2, since these cause the 2 β-mem node additions (Appendix, item 19),

tokLen, the average length for a token in the opposite memory, is 2 (Appendix, item 21),

$LPE_{two-input}$, the number of non-overlapped LPE instructions to do the appropriate array accesses and create and stack any output tokens, is simply assumed to be 50.

Thus $T_{two-input} = 127F + 34S + 50H$.

The processing of token deletions induced by an addition to the right-hand input of a NOT node is as follows. The goal is to remove all tokens in descendant nodes in the Rete net that depend on the token that is deleted. This is done by associatively activating all those nodes, associatively matching the working memory element ID's comprising the deleted token to find tokens that depend on the deleted one, and erasing all such tokens in parallel. The time for this processing is given by

$$T_{rhsNOTdeletion} = e_{del}$$

$$x \{ \ 69F + \text{nDescendants} \times 8F + \text{wmIDs} \times (2F + 2S)$$
$$+ \ \text{if wmIDs} > 1 \ \text{then} \ 4S \times [(\text{wmIDs} - 1) \ \text{DIV} \ 2]$$
$$+ \ \text{if wmIDs even} \ 1F + 2S$$
$$+ \ \text{nCSdel} \times (8F + 2S)$$
$$+ \ (\text{maxLen} - 1) \times (3F + 2S) \ \}$$
$$+ \ \text{LPE}_{\text{rhsNOTdeletion}}$$

where

e_{del}, the expected maximal number of deletions in any partition as a result of an entry into a NOT node, is 0.6 (Appendix, item 23). (It makes sense to use a non-integral expectation since synchronization is not required after each addition or deletion.)

nDescendants, the average number of β-mem nodes in the Rete net that are descendants of a NOT node is 2 (Appendix, item 24),

wmIDs, the average number of working memory element ID's stored in a β-mem token, is 3 (Appendix, item 13),

nCSdel, the number of conflict set members deleted as a result of deleting one token is 0.16 (Appendix, item 26),

maxLen, the length of the largest token, is 9 (Appendix, item 15)

$\text{LPE}_{\text{rhsNOTdeletion}}$ is assumed to be 40 instructions.

Thus $T_{\text{rhsNOTdeletion}} = 70F + 16S + 40H$.

The total number of processing cycles resulting from an addition to working memory, T_{add}, is the sum of the 6 partial results obtained above.

$$T_{\text{add}} = T_{\text{broadcast}} + T_{\text{match}} + T_{\alpha\text{-mem}} + T_{\beta\text{-mem}}$$
$$+ \ T_{\text{two-input}} + T_{\text{rhsNOTdeletion}}$$

$$= 858F + 98S + 230H$$

8.2 Deletion from Working Memory

Deletion of a working memory element is considerably faster than addition, since the principal actions are associatively searching among all α-mem and β-mem nodes to locate tokens that depend on that working memory element, and erasing all such tokens in parallel. It also is necessary to delete the working memory element from the LPE table. There are two complications that arise, however. First, deletion of a working memory element from the right-hand input of a NOT node can "unblock" tokens in the left-hand input. This case is infrequent, but if it occurs, processing for the comparison of relevant bindings is necessary. Second, deletion of a working memory element may cause the removal of members of the conflict set. This will happen naturally in token memory, but the affected instantiations must be retrieved by the LPE so the tokens can be removed from the global conflict set table. The time required for deletion of working memory elements is given by

$$
\begin{aligned}
T_{del} = {} & 80F + 6S + maxLen \times (6F + 4S) \\
& + rhsNOT \times [10F + (tokLen - 1) \times (7F + 2S) + matchCost] \\
& + nCsDel \times (8F + 2S) \\
& + LPE_{del}
\end{aligned}
$$

where

maxLen, the length of the largest token, is 9 (Appendix, item 15),

rhsNOT, the maximum number of tokens in any one partition removed from right-hand inputs of NOT nodes as a consequence of a working memory deletion, is 1 (Appendix, item 25),

tokLen, the average length for a token in the opposite memory, is 2 (Appendix, item 21),

matchCost, the number of cycles to match relevant bindings, is given by the formula for $T_{two-input}$ given above, but with the nAdds parameter $=$ 1 for the token removed in this case, and the

31

nSuccessfulMatch parameter = 0 (Appendix, item 28), so
matchCost = 15F + 3S + 50H,

nCsDel, the average number of instantiations removed from the conflict set as a result of the deletion of an arbitrary working memory element, is 0.16 (Appendix, item 26),

LPE_{del} is simply assumed to be 40 instructions.

Thus $T_{del} = 167F + 47S + 90H$.

8.3 Total Time per Production Firing

Since there are 2.21 changes (additions/deletions) to working memory per production firing (Appendix, item 27), the time for these changes (Appendix, item 29) is given by

$$T_{firing} = 2.21 \times (T_{add} + T_{del})/2 + T_{rhs}$$

where

T_{rhs} is the number of host instructions needed for conflict resolution and evaluation of the chosen production's right-hand side.

Given the assumption that $T_{rhs} = 500$ [Gupta, 1984],

$$T_{firing} = 1133F + 160S + 854H.$$

From section 5 we have F = 350 nanoseconds, S = 3 microseconds, and H = 333 nanoseconds. Hence $T_{firing} = 1161$ microseconds, which yields an execution rate of 861 productions per second.

9 Conclusions

NON-VON's projected rate of production evaluation reflects the performance of a heterogeneous architecture designed for rapid symbolic computation. The massive parallelism of NON-VON's small processing elements was designed to be particularly efficient in executing such operations as associative matching and data storage and

retrieval. The large processing elements and host are few enough in number that it is feasible to make them quite powerful, with the relatively large RAM memories needed to hold control tables and data structures such as the compiled Rete network, together with substantial amounts of program code.

NON-VON's partitioned-SIMD mode of execution, in which instructions are broadcast to the SPE's for execution, avoids a need to replicate identical code in thousands of processing elements. In addition, the storage capacity of NON-VON's small processing elements is well matched to the size of typical condition element terms and memory nodes. (Viewed in a more general context, the NON-VON small processing element was designed to have a capacity on the order of the size of a typical "record".) In coarser grain, strictly MIMD machines the fit is not as natural, and software mechanisms such as hash tables are used to accommodate the storage of several items in each processing element, amortizing the cost of the larger processor and program code over a greater quantity of data.

Although the formulas and parameters that predict NON-VON's performance on this problem have been analyzed in considerable detail, the limitations of our analytic techniques should not be ignored. An important area for future work is the validation of our analysis by experimental measurements.

Acknowledgments

References

B. G. Buchanan, "New research on expert systems", *Machine Intelligence 10*, J. E. Hayes et al. (eds.), Halsted Press, New York, 1982, pp. 269-299.

Michael J. Flynn, "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, vol. c-21, September 1972, pp. 948-960.

Charles L. Forgy, *On the Efficient Implementation of Production Systems*, Ph D. Thesis, Carnegie-Mellon Computer Science Department, February 1979.

Charles L. Forgy, "Note on production systems and Illiac IV", Technical Report, Carnegie-Mellon Computer Science Department, July 1980.

Charles L. Forgy, *OPS5 Users' Manual*, Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

Charles L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem", *Artificial Intelligence*, vol. 19, no. 1, September 1982, pp. 17-37.

Charles L. Forgy and John McDermott, "OPS, a domain-independent production system language", *IJCAI-77, Proceedings of the fifth international joint conference on artificial intelligence*, August 1977, pp. 933-939.

Anoop Gupta, "Implementing OPS5 production systems on DADO",*Proceedings of the 1984 international conference on parallel processing*, August 21-24, 1984, pp. 83-91.

Anoop Gupta and Charles L. Forgy, "Measurements on production systems", Technical Report, Carnegie-Mellon Computer Science Department, 1983 (undated).

Frederick Hayes-Roth et al. (eds.), *Building Expert Systems*, Addison-Wesley, Reading, Mass., 1983.

Toru Ishida and Salvatore J. Stolfo, "Simultaneous firing of production rules on tree-structured machines", Technical Report, Columbia Computer Science Department, 1984.

Donald E. Knuth, *The Art of Computer Programming: Volume 1, Fundamental Algorithms*, Addison Wesley, Reading, Massachusetts, 1968.

Douglas B. Lenat, et al., "Cognitive economy in artificial intelligence systems", *IJCAI-79, Proceedings of the sixth international joint conference on artificial intelligence*, Tokyo, August 20-23, 1979, pp. 531-536.

Douglas B. Lenat and John McDermott, "Less than general production system architectures", *IJCAI-77, Proceedings of the fifth international joint conference on artificial intelligence*, August 1977, pp. 928-932.

Donald L. McCracken, "Representation and efficiency in a production system for speech understanding", *Proceedings of the sixth international joint conference on artificial intelligence*, Tokyo, August 20-23, 1979, pp. 556-561.

John McDermott, "R1: A rule-based configurer of computer systems", *Artificial Intelligence*, vol. 19, no. 1, September 1982, pp. 39-88.

Daniel P. Miranker, "A framework for discussing HERBAL", Working Memo, Columbia Computer Science Department, 1984a.

Daniel P. Miranker, "Performance estimates for the DADO machine: A comparison of TREAT and RETE", *International conference on fifth generation computer systems*, November 1984b, pp. 449-457.

Nils J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, Calif., 1980.

Kemal Oflazer, "Partitioning in parallel processing of production systems", *Proceedings of the 1984 international conference on parallel processing*, August 21-24, 1984, pp. 92-100.

Ron Sauers and Rick Walsh, "On the requirements of future expert systems", *IJCAI-83, Proceedings of the eighth international joint conference on artificial intelligence*, August 1983, pp. 110-115.

David Elliot Shaw, *Knowledge-Based Retrieval on a Relational Database Machine*, Ph.D. Thesis, Stanford Department of Computer Science, 1980.

David Elliot Shaw, "Organization and operation of a massively parallel machine", in Guy Rabbat (ed.), *Computers and Technology*, Elsevier - North Holland, 1985.

David Elliot Shaw and Bruce K. Hillyer, "Allocation and manipulation of records in the NON-VON supercomputer", Technical Report, Columbia Computer Science Department, 1982.

David Elliot Shaw and Theodore M. Sabety, "The multiple-processor PPS chip of the NON-VON 3 supercomputer", *Integration: The VLSI Journal* (accepted for publication), 1984.

Salvatore J. Stolfo, "Five parallel algorithms for production system execution on the DADO machine", *AAAI-84, Proceedings of the national conference on artificial intelligence*, August 6-10, pp. 300-307.

Salvatore J. Stolfo, Daniel Miranker, and David Elliot Shaw, "Architecture and applications of DADO: a large-scale parallel computer for artificial intelligence", *IJCAI-83, Proceedings of the eighth international joint conference on artificial intelligence*, August 1983, pp. 850-854.

Salvatore J. Stolfo and David Elliot Shaw, "DADO: a tree-structured machine architecture for production systems", *AAAI-82*, *Proceedings of the national conference on artificial intelligence*, August 18-20, pp. 242-246.

Patrick H. Winston, *Artificial Intelligence*, Addison Wesley, Reading, Mass., 1977.

Appendix

NON-VON's performance in executing OPS5 depends on several parameters of the production system in question. For the analysis presented in this paper, statistics have been derived from averages over the six production systems measured in [Gupta and Forgy, 1983]. Although most values are obtained directly from that work, some plausible inferences have been necessary, and actual measurements of the needed parameters would be preferable. The derivations and justifications for the statistics employed are given below. All references to pages and tables cite [Gupta and Forgy, 1983].

1) The average number of α-mem node additions per change to working memory is 5.09 (p. 25, Table 5-2, line 1).

2) The average static sharing of α-mem nodes is 3.51 (p. 23, Table 4-5, line 2).

3) The average number of condition elements having all intra-condition tests satisfied by an arbitrary working memory addition (without sharing) is approximated as 17.87, the product of 1) and 2) above. That this is an approximation since it is the *dynamic* sharing of α-mem nodes that is relevant; Gupta states (private communication) that 30 is a more accurate value, so we use 30 for the analysis.

4) The average number of α-mem additions per partition is 0.94, the ratio of 30 (item 3 above) to 32, the number of partitions. If the α-mem additions were evenly spread over the partitions, 30 partitions would have one addition each, and 2 would have none. If, however, the additions occur randomly, with a uniform distribution over the 32 partitions, clustering would cause an expected maximum of 3.4 additions in some partition. It is hoped that intelligent partitioning of the production system can do better than randomizing the activity, but this remains an open question. Oflazer [1984] reports 3 heuristics for partitioning production systems; all 3 perform better than a random distribution of rules. He does not state the effect of his heuristics on α-mem node additions. We assume that as a result of partitioning, on the average there are at most 3 α-mem additions in any partition.

5) The average size of the conflict set is 16.0 (p. 30, Table 5-8, line 3).

6) The average number of changes to the global conflict set per firing cycle is 5.3 (p. 29, Table 5-6, line 3).

7) The average number of condition elements per production is 4.11 (p. 21, Table 3-8, line 2).

8) The average number of attributes in a condition element is 3.71 (p. 21, Table 3-8, line 5).

9) The average number of variables in a condition element is 1.56 (p. 21, Table 3-8, line 6).

10) The average number of intra-condition tests for a condition element is 2.93, the difference between 8) and half of 9). This is based on the assumption that each variable occurs once to be bound and once in an inter-condition test. All attribute occurrences that are not associated with an inter-condition test must be for intra-condition testing. This analysis neglects the effect of conjunction and disjunction expressions, which although rare, would tend to raise the number of intra-condition tests.

11) The average number of productions is 909.83 (p. 21, Table 3-8, line 1).

12) The maximum aggregate number of α-mem and β-mem tokens is 4616 (p. 30, Table 5-8, line 6.)

13) The number of working memory element ID's stored in an α-mem token is 1. The size of a β-mem token ranges from 2 (most frequent because of the progressive filtering performed by the Rete net) up to the number of positive condition elements in a production (average 4). Thus we say the average number of working memory element ID's in a β-mem tokens is 3.

14) The number of attributes per class ranges from 1 to 152 (pp. 19-20, Tables 3-1 through 3-6). Since the OPS5 language manual states that the maximum number of attribute slots per class is 126, we see that for some classes, attribute names have been mapped by OPS5 literal declarations to shared physical locations. A static weighted average of all 41 most frequent classes listed in these tables, assuming no attribute folding (i.e., 152 distinct physical locations are permitted in a class) gives 11.4 attributes per class.

15) The average of the largest number of terms in a condition element is 9, by inspection and averaging of the values ranging from 7 to 11 found in (p. 13, Figure 3-4).

16) The average number of relevant bindings in a token is 1. This is derived in the following way. The average number of relevant bindings

is the quotient of the average number of tokens in an opposite node,
divided by the average number of tests that are performed when a token
is inserted. For an AND node this value is 0.88 (p. 27, Table 5-4,
line 4 divided by line 3), and for a NOT node this value is 0.97 (p. 28,
Table 5-5, line 4 divided by line 3). Note also that a static figure of
0.8 is obtained from (p. 21, Table 3-8, line 7). Thus we state that the
average number of relevant bindings in a token is 1. This applies both
to α-mem and to β-mem tokens.

17) The average length of a token is the greater of the number of relevant
bindings, or the number of working memory ID's stored in the token. For
α-mem tokens, both figures are 1. For β-mem tokens, the number
of working memory ID's is 3, as described in 13), so the average length
of a β-mem token is 3.

18) The average number of entries to the right-hand input memory of an AND
node is 22.37 (p. 27, Table 5-4, line 1). The average number of entries
to the right-hand input memory of a NOT node is 4.73 (p. 28, Table 5-5,
line 1). Thus 17% of right-hand entries are to NOT nodes.

19) The average number of additions to β-mem node memory resulting from
an addition to working memory is 6.31 (p. 26, Table 5-3, line 1). If
6 additions are independently uniformly distributed over 32 partitions,
the expected maximum number in any partition is 1.41. We assume that
as a consequence of partitioning the productions, on the average there
are at most 2 β-mem additions in any partition.

20) For an AND node, the average number of entries to the right-hand input
is 22.37, and to the left-hand input is 7.2 (p. 27, Table 5-4, line 1).
Thus 76% of entries to AND nodes are to the right-hand input, and 24% are
to the left-hand. When entering the right-hand input, the opposite node
is empty 87.17% of the time, and when entering the left-hand input, the
opposite is empty 43% of the time (p. 27, Table 5-4, line 2). Thus the
weighted probability is 0.77 that an entry to an AND node will find the
opposite empty.

For a NOT node, the average number of entries to the right-hand input
is 4.73, and to the left-hand input is 1.2 (p. 28, Table 5-5, line 1).
Thus 80% of entries to NOT nodes are to the right-hand input, and 20% are
to the left-hand. When entering the right-hand input, the opposite node
is empty 70.33% of the time, and when entering the left-hand input, the
opposite is empty 25.5% of the time (p. 28, Table 5-5, line 2). Thus the
weighted probability is 0.61 that an entry to a NOT node will find the
opposite empty.

Thus we say that the probability of finding the opposite node empty is

0.7, and the probability of finding tokens in the opposite node is 0.3.

Gupta (private communication) points out that this does not hold for "long-chain" activations that propagate a token through consecutive two-input tests during one execution cycle.

21) The length of all tokens in α-mem nodes is 1, and the average length for tokens in β-mem nodes is 3 (Appendix, item 17). We assume that most tokens are in α-mem nodes, and the β-mem tokens are short because of the progressive filtering action of the Rete net, so we say the average length of an arbitrary token is 2.

22) The probability that an addition to a NOT node triggers a deletion is 0.12, which is 0.3 x 0.49 x 0.8, where 0.3 is the probability that the opposite node is occupied (p. 28, Table 5-5, line 2), 0.49 is the average number of tokens that successfully match given that the opposite node is occupied (p. 28, Table 5-5, line 5), and 0.8 is the probability that an entry to a NOT node is to the right-hand input (p 28, Table 5-5, line 1).

23) The expected maximal number of deletions in any partition as a result of an entry into a NOT node is calculated as follows. There is a total of 5.93 entries to NOT nodes in all partitions (p. 28, Table 5-5, line 1). Let $T(n,k)$ denote the probability that an aggregate of n entries to NOT nodes triggers k token deletions. We round 5.93 to 6, and using the probability 0.12 that any 1 entry triggers a deletion, obtain through elementary probability theory that $T(6,0) = .4644$, $T(6,1) = 0.38$, $T(6,2) = 0.1295$, and the sum of $T(6,k)$ for $k > 2$ is 0.0261. Next we assume the 6 entries are uniformly distributed over the 32 partitions, and perform elementary combinatorial calculations to obtain the number of entries in each partition. (The assumption of uniform distribution is discussed in item 4, above.) We state the two most common occurrences. The probability that six partitions have one entry each is 0.61. The probability that one partition has two entries and four other partitions have one entry each is 0.34. Multiplying these and similar values by the $T(6,k)$ and grouping by the maximum number of deletions triggered gives the probability $D(k)$ that k deletions are triggered. We obtain $D(0) = 0.46$, $D(1) = 0.48$, $D(2) = 0.06$, and all other values are nearly 0. Hence the expected maximum number of deletions in any partition is 0.60.

24) The average number of β-mem nodes that are descendants of a NOT node is no more than 2. This is supported by two considerations. The average number of β-mem nodes for a production is one less than the average number of condition elements (Appendix, item 7), hence it is 3. The average number of AND nodes is 1845 (p. 21, Table 4-1, line 4), but the average

number of NOT nodes is only 481 (p. 21, Table 4-1, line 5), so it is likely that the ancestors of a β-mem node are AND nodes.

25) On average, 1.48 tokens are removed from right-hand inputs of NOT nodes as a consequence of a working memory deletion. This figure is obtained in the following way. The average number of tokens in all nodes is 1289 5 (p. 30, Table 5-8, line 5). The average number of working memory elements is 295.75 (p. 30, Table 5-8, line 1). Thus there are 4.36 tokens per working memory element. Since a average token contains 2 working memory element ID's (Appendix, item 21), an average working memory element is represented in 8.72 tokens. Since less than 17% of all-tokens are stored in right-hand inputs of NOT nodes (Appendix, item 18), the figure of 1.48 is obtained. We assume that at most one occurs in any partition. Actual dynamic measurements of this statistic are needed.

26) The average number of working memory elements is 295 (p. 30, Table 5-8, line 1). The average size of the conflict set is 16 (Appendix, item 5). The average number of condition elements in a production is 4.11 (Appendix, item 7). Since negated condition elements are not represented in a conflict set instantiation, and assuming 3/4 of the condition elements are non-negated, the average size of an instantiation is 3. Thus 3x16/295, or 0.16, is the average probability that deletion of a working memory element removes a member of the conflict set.

27) The average number of changes to working memory per production system cycle is estimated as follows. From (p. 20, Table 3-7, lines 1-3) we calculate the number of changes to working memory as a percent of the total number of rhs actions by summing the percent of actions that are MAKE, the percent that are REMOVE, and twice the percent that are MODIFY (since a modify is a make and a remove). These product of these values with the (static) number of actions per production (p. 21, Table 3-8, line 3) gives an estimate of the number of changes per working memory that results from firing a production. The average of the values thus obtained is 2.21 changes per firing. A value derived from dynamic measurements would be preferable.

28) The number of additions to β-mem resulting from the deletion of a working memory element was considered too insignificant to analyze by Gupta [1984], and no statistics were presented for this case in [Gupta and Forgy, 1983]. We assume these statistics were aggregated with those for two-input processing and β-mem additions that result from the addition of working memory elements, and thus they have already been accounted for in the addition portion of the analysis.

29) The formula for total time per production formula assumes that the number

of working memory additions is equal to the number of working memory deletions, which should be true over the long-term, assuming working memory size does not grow without bound. We also note a comment from Anoop Gupta [private communication, 1985]: "The assumption is made that the same partition exhibits worst-case performance for all of the 2.21 changes. That is probably not so, and will result in overall better performance than predicted."

```
(p sort-work
   (current-task ^taskname sort)
   (counter ^value <n>)
   (number ^value <x> ^used no)
 - (number ^value < <x> ^ used no)
 -->
   (write <x>)
   (modify 2 ^value (compute <n> + 1))
   (modify 3 ^used yes))

(p sort-done
   (current-task ^taskname sort)
   (counter ^value <total>)
 - (number ^used no)
 -->
   (write <total> items sorted)
   (remove 1))
```

; If current task is to sort
;   and output counter is n
;   and there is an unused number x
;   but no smaller unused number
; Then
;   write x to output
;   increment output counter
;   and mark x as used.

; If current task is to sort
;   and output counter is total
;   but no unused number remains
; Then
;   write total number of items
;   and terminate sorting task

Figure 1:   Example OPS5 Productions

```
(current-task ^taskname sort)
(counter ^value 0)
(number ^value 17 ^used no)
(number ^value  5 ^used no)
(number ^value 23 ^used no)
```

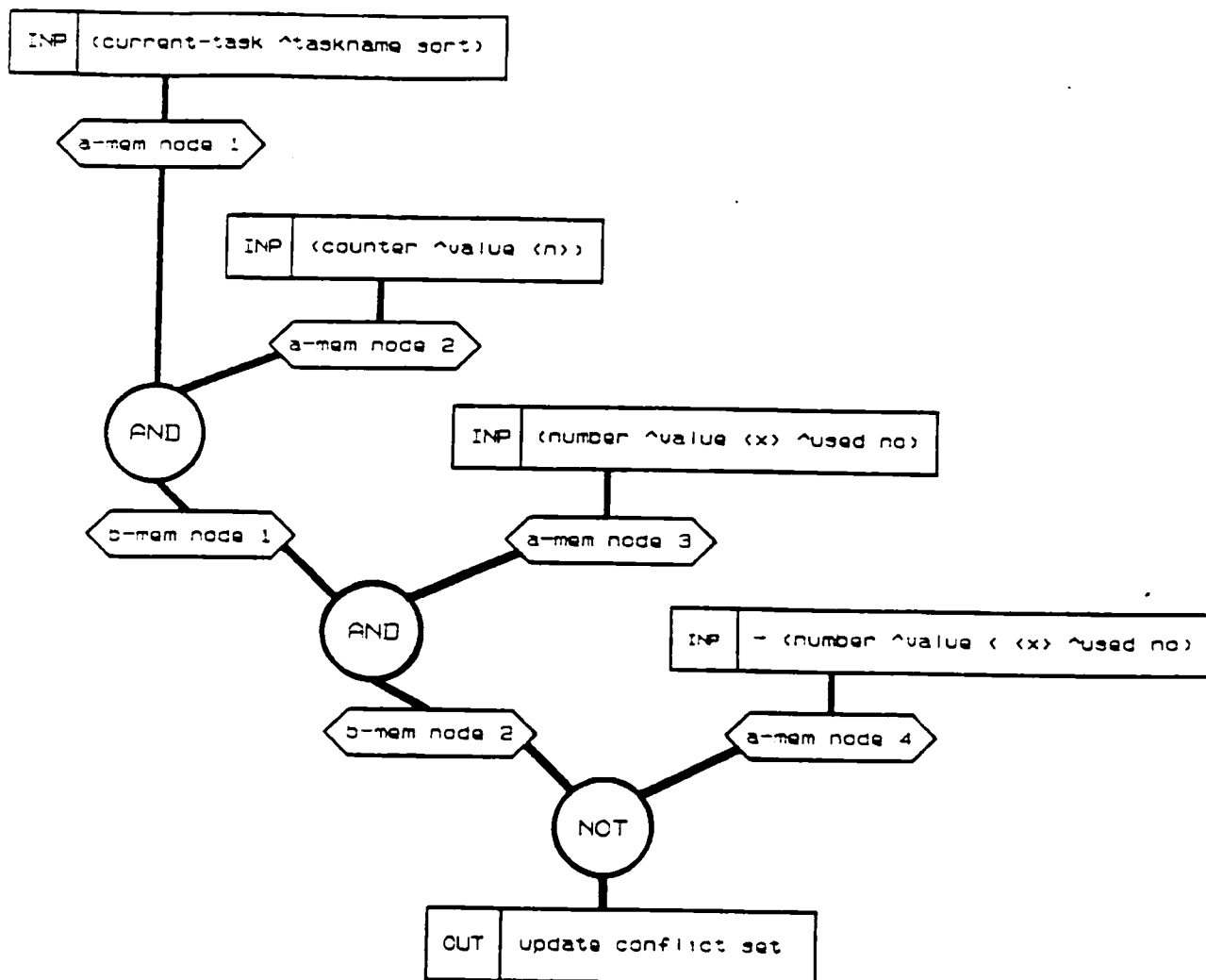Figure 2:   Example Working Memory Elements
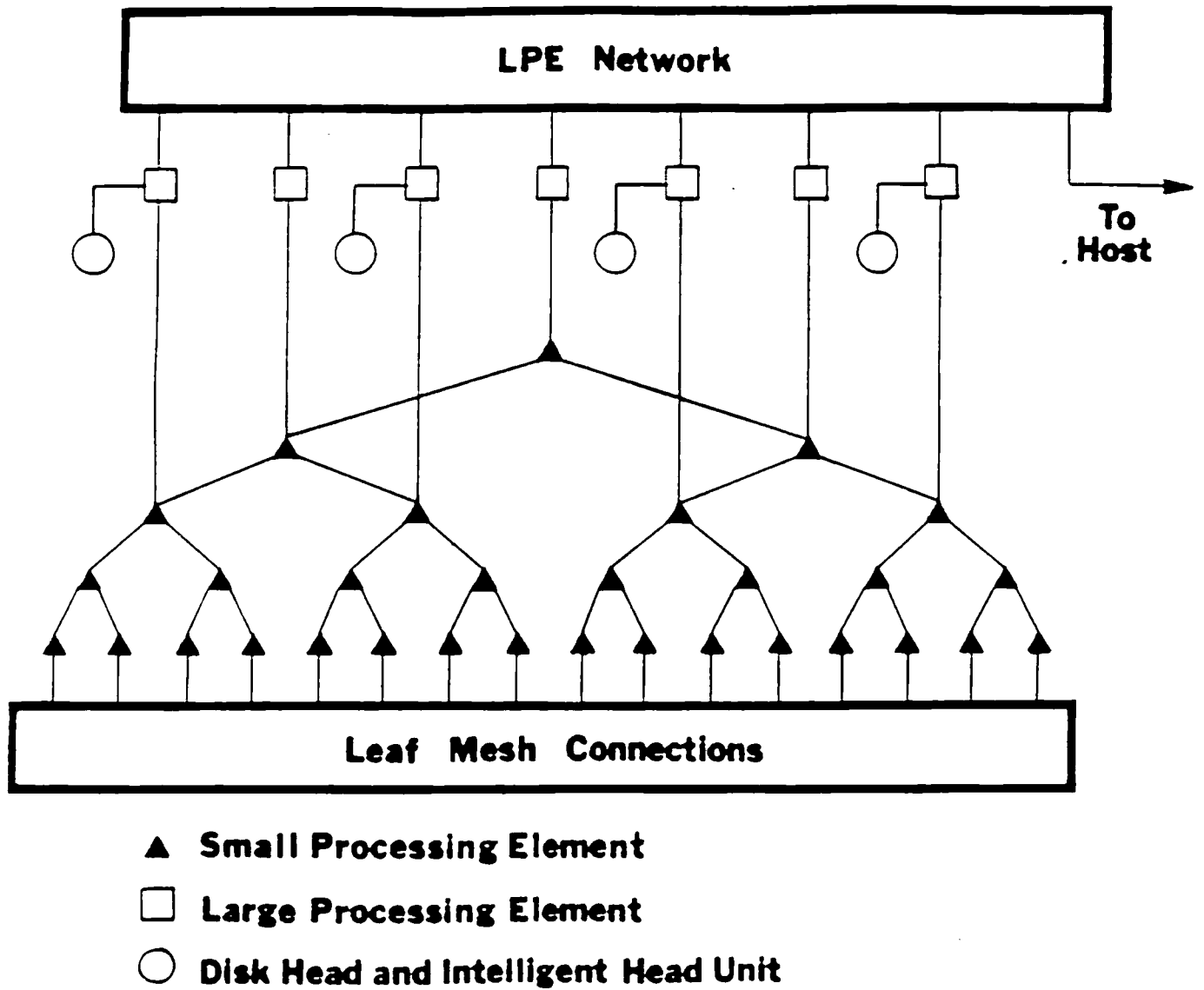
Figure 3: Example Dataflow Graph for NON-VON

▲ Small Processing Element

☐ Large Processing Element

◯ Disk Head and Intelligent Head Unit

Figure 4: Organization of the NON-VON Machine

▲ **Small Processing Element**

☐ **Host**

◯ **Large Processing Element**
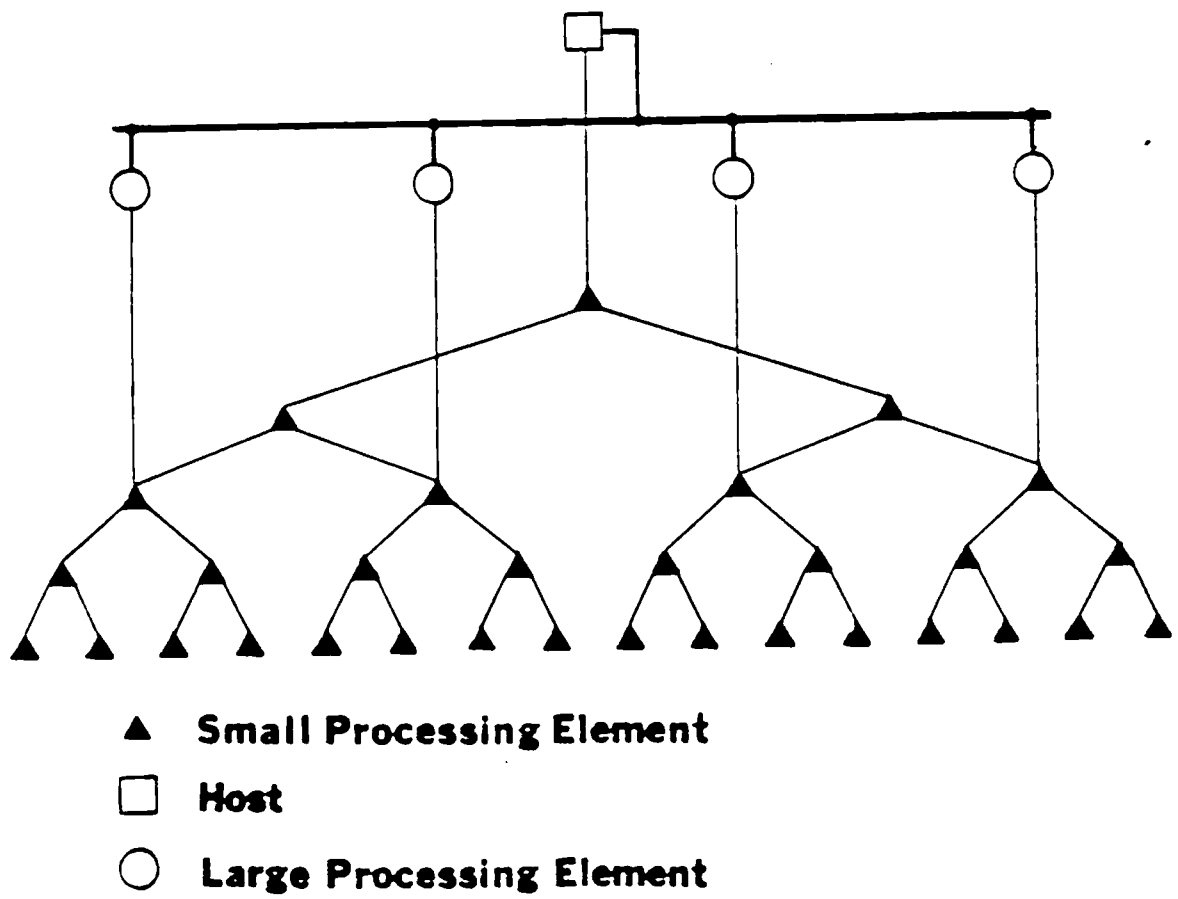
Figure 5: NON-VON: Reduced Configuration for Production Systems
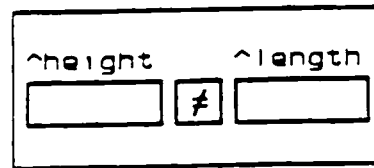
SPE 1:

```
^classname   const
[        ] [=] [        ]
```

SPE 2:

```
^width      ^length
[        ] [≠] [        ]
```

SPE 3:

```
^height     ^length
[        ] [≠] [        ]
```
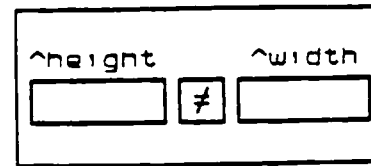
SPE 4:

```
^height     ^width
[        ] [≠] [        ]
```

Figure 6: Condition Element Stored in Four SPE's