# A Concurrent Architecture for Serializable Production Systems

José Nelson Amaral[1] and Joydeep Ghosh[2]
Department of Electrical and Computer Engineering,
University of Texas at Austin,
Austin, Texas 78712

**Abstract**

This paper presents a new production system architecture that takes advantage of modern associative memory devices to allow parallel production firing, concurrent matching, and overlap among matching, selection, and firing of productions. We prove that the results produced by the architecture are correct according to the serializability criterion. A comprehensive event driven simulator is used to evaluate the scaling properties of the new architecture and to compare it with a parallel architecture that does global synchronization before every production firing. We also present measures for the improvement in speed due to the use of associative memories and an estimate for the amount of associative memory needed. Architectural evaluation is facilitated by a new benchmark program that allows for changes in the number of productions, the size of the database, the variance between the sizes of local data clusters, and the ratio between local and global data. Our results indicate that substantial improvements in speed can be achieved with a very modest increase in hardware cost.

**keywords:** Production Systems, Parallel Architectures, Performance Evaluation, Rete Network, Benchmarking, TSP, Rule Partitioning, System Level Simulation.

[1] J. N. Amaral (*amaral@music.pucrs.br*) was supported by a fellowship from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). He is currently with the Electrical Engineering Department at Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) - Brazil.

[2] J. Ghosh (*ghosh@pine.ece.utexas.edu*) was supported in part by NSF under grant ECS-9307632 and by Faculty Development Awards from TRW Foundation and Schlumberger.

# Contents

**1 Introduction** **1**

**2 Background** **1**

**3 Architectural Model** **3**

3.1 Basic Definitions . . . . . 4

3.2 System Overview . . . . . 6

3.2.1 Detailed Processor Model . . . . . 10

3.2.2 Conflict Set Management . . . . . 11

3.2.3 Broadcasting Interconnection Network Arbitration . . . . . 14

3.3 Correctness of the Processing Model . . . . . 15

**4 Production Partitioning Algorithm** **19**

**5 Performance Evaluation** **22**

5.1 Benchmarking . . . . . 22

5.1.1 A Contemporaneous TSP . . . . . 23

5.1.2 A Production System Solution for CTSP . . . . . 23

5.1.3 Confusion of Patents Problem . . . . . 26

5.1.4 The Hotel Operation Problem . . . . . 26

5.1.5 The Game of Life . . . . . 26

5.1.6 The Line Labeling Problem . . . . . 27

**6 Performance Measurements** **27**

6.1 Parallel Firing Speedup . . . . . 28

6.2 Effectiveness of Associative Memories . . . . . 31

6.3 Associative Memory Size . . . . . 34

6.4 Use of Bus . . . . . 35

**7 Concluding Remarks** **35**

**8 Acknowledgements** **36**

# List of Figures

1 Parallel Machine Model . . . . . 8

2 Processing Element Model . . . . . 10

3 (a) Antecedents of Fireable Instantiations Memory; (b) Fireable Instantiations Memory. . . 12

4 Pending Matching Memory . . . . . 13

5 Speed improvement measures comparing "a" curves representing the new architecture that eliminates oversynchronization with the "s" curves of an idealized synchronous architecture that solves conflict set in one time step. Both systems use associative memories. . . . . 29

6 Speed improvement measures comparing "a" curves representing the new architecture that eliminates oversynchronization with the "s" curves of an idealized synchronous architecture that solves conflict set in one time step. Both systems use associative memories. . . . . 30

# List of Tables

1 Possible dependencies between $R_n$ and $R_m$. . . . . 18

2 Static measures for the CTSP benchmark according to $C$ and $\mu_c$ . . . . . 25

3 Static Measures for Benchmarks Used. . . . . 27

4 Speedup over synchronized architecture using the same number of processors. . . . . 31

5 Speedup due to use of CAMs$^3$. . . . . 33

6 Maximum and average "crest" for memory size (bytes). . . . . 34

7 Percentage of time that the bus is busy. . . . . 35

# 1 Introduction

Considerable efforts have been made towards speeding up production system machines in the past twenty years [6, 29]. Originally, production systems were realized as interpreted language programs for sequential machines. The high cost of matching motivated the development of concurrent matching systems and, subsequently, systems that also allowed multiple productions to be fired at the same time. In a separate line of research, modern compile optimization techniques were developed to run production system programs more efficiently on general purpose sequential machines.

These efforts have led to great advances in the understanding of the issues involved in the construction of faster production system machines, but only limited improvement in actual performance. Also, there have been few attempts to integrate progress made in different areas: the use of the restrictive commutativity criterion for correctness and the notion of a match-select-act "cycle" forced even advanced architectures to perform synchronization before each production firing; compile optimization techniques were mostly restricted to sequential machines; many of the concurrent matching engines were constructed with a large number of small processors and were not combined with parallel firing techniques. Moreover, parallel processing researchers failed to take advantage of the fact that, in typical production systems, reading operations are performed much more often than writing ones.

We propose a novel parallel production system architecture that uses the less restrictive serializability criterion for correctness. This architecture eliminates the concept of a production system "cycle", thus eliminating the need to construct a global "conflict set" and to perform global synchronization before each production firing. Productions are partitioned among processors based on information about the workload of each production and on production dependencies identified through compiling techniques. The use of modern content addressable memories allows a new production to be selected to fire before all the matches resulting from previous production actions are complete. This architecture follows an early recommendation of Gupta and Forgy [15], i.e., that a parallel production system machine be constructed with a small number of relatively powerful processors.

# 2 Background

Attempts to speed up Production Systems (PS) date back to 1979 when Forgy created the Rete network, a state saving algorithm to speed up the matching phase of PS [11]. Following a 1986 study by Gupta, which indicated that a significant portion of the processing time in a Rete-based PS machine is consumed in the matching phase [14], substantial efforts were made to improve this phase. This includes concurrent

implementations of the Rete network [13, 16, 25, 24, 39], generalization of the Rete network [30], elimination of internal memories from the Rete network to increased speed [31], extension of the Rete network for compatibility with real-time systems [9], and the use of message-passing computers to implement the Rete network [2]. Progress in other aspects of production system machines included compile time optimization for the Rete network [19], nondeterministic resolution for the conflict set combined with parallel firing of productions [21, 27], loosely coupled implementations of production systems [21], and the use of meta rules to solve the conflict set [40]. Comprehensive surveys of the research towards speeding up production systems are found in the works of Kuo and Moldovan [29] and Amaral and Ghosh [6].

The issue of which criterion to use for correctness in the execution of a production system is still an open question. The two most prominent candidates are the *commutativity criterion* and the *serializability criterion*. When commutativity is used, a set of rules can be executed in parallel if and only if the result is the same that would be produced by *any* possible sequential execution of the set. Under serializability it is enough that the result produced by the parallel execution be equal to *at least one* sequential execution of the set [36].

The commutativity criterion proposed by Ishida and Stolfo [20] is favored by programmers because it allows for easy verification of correctness in a production system. However, it is very restrictive and the amount of parallelism extracted from a PS using this criterion is very low. The use of the serializability criterion increases the amount of parallelism available but makes the verification of correctness in a program more difficult. Nevertheless, if serializable production systems are proven to be sufficiently faster than commutable ones, development tools will be created to aid the verification of correctness.

Schmolze and Snyder [38] studied the use of confluence to control a parallel production system. They suggest the use of term rewriting systems [17, 26] to verify the confluence of a production set. They argue that a confluent production set that is guaranteed to terminate will produce the same final result independent of the sequence in which the productions are executed. Therefore, for such a class of systems, the verification of correctness with the serializability criterion would not impose an extra burden in the programmer.

The need to improve other phases of production execution besides the match cycle is now evident [6]. In this paper we present a parallel architecture based on the serializability criterion of correctness. The architecture exploits the high read/write ratio of production systems, and the increased importance of associative search operations when global synchronization is eliminated, to yield a fast and efficient production system engine. The next section presents the architectural model and proves that its operation is correct. In section 4 we present a partitioning algorithm that performs the assignment of productions to processors. Section 5 describes the benchmarks used to study performance and introduces a new benchmark

program. Section 6 presents comparative measurements with a synchronized architecture and an evaluation for the volume of activity in the bus and the size of associative memories.

# 3 Architectural Model

The parallel architecture presented in this paper stems from the realization that improvements restricted to the matching phase of the traditional match-select-act cycle of Production Systems (PS) fail to produce significant speedup. Even machines that allow concurrent execution of the acting and matching phases, while maintaining the global production selection, yield limited improvements in speed. The architecture proposed here allows parallel firing of productions allocated to distinct processors. Within a processor, activities related to matching, acting and selecting are concurrent. Thus the next instantiation to be fired may be selected even before the Rete network updates due to a previous production firing are completed.

Such aggressive parallelism is possible because the concept of a match-select-act cycle is eliminated. The principle of firing the most recent and specific instantiation is replaced by an approximation of it: only instantiations that are known at the time of the selection are considered, we call this a *partially informed* selection mechanism. The use of associative memories allows for quick elimination of instantiations that are no longer fireable. We also replace the restrictive commutativity criterion by the serializability criterion of correctness. The use of serializability reduces the number of situations in which synchronization is necessary, increasing the amount of parallelism available.

On surveying measurements published by other authors [14, 33], we found that the ratios of reading and writing operations in the benchmarks studied are between 100 and 1000. We also found that in complex benchmarks that bear more similarity with "real life" problems, this ratio tends to be higher than in "toy problems". This is primarily because productions have a larger number of antecedents than consequents in such problems [4]. Our observation motivates an architecture based on a broadcasting network over which only writing operations occur. Such an architectural model imposes limits to the number of processors used. However, two characteristics of PS make them compatible with an architecture with a moderate number of processors: the amount of inter-production parallelism is limited and, as a PS grows, the size of the database grows much faster than its production set.

Section 3.1 presents basic definitions that set the environment for the processing model. Section 3.2 introduces the architectural organization and expands on the processor model, conflict set management, and processor operation. Section 3.3 presents a theorem that demonstrates that the results produced by the processing model is correct according to the serializability criterion of correctness.

### 3.1 Basic Definitions

A Production $R_i$ consists of a set of antecedents $A(R_i)$ and a set of consequents $C(R_i)$: the antecedents specify the conditions upon which the production can be fired; the consequents specify the actions performed when the production is fired.

**Definition 1** *The database manipulated by a Production System consists of a set of assertions. Each assertion is represented by a* **Working Memory Element** *(WME), notated by $W_k$. A WME consists of a class name and a set of attribute-value pairs. The class name and the set of attribute names of a WME together characterize its* **type**, *$T[W_k]$.*

**Definition 2** *Each production antecedent specifies a type of WME and a set of values for its attribute-value pairs. A WME $W_k$ is* **tested** *by an antecedent if it has the specified type. An antecedent is* **matched** *by a WME if the WME has the type specified* and *all the values in the antecedent match the ones in the WME.*

**Definition 3** *If an antecedent of a production $R_i$ tests WMEs of type $T[W_k]$, then we say that $W_k$* **is tested by** *the production $R_i$, this is notated by $W_k \rhd A(R_i)$.*

**Definition 4** *A* **non-negated antecedent** *tests for the presence of a matching WME in the memory. A* **negated antecedent** *tests for the absence of any matching WME in the memory. A production $R_i$ is said to be* **fireable** *if all its non-negated antecedents are matched and none of its negated antecedents are matched.*

The consequent of a production can specify three kinds of actions that modify WMEs: addition, deletion, or modification.

**Definition 5** *A WME $W_k$* **is modifiable** *by the consequents of a production $R_i$ iff the firing of $R_i$ adds (deletes) any WME of type $T[W_k]$ to (from) the Working Memory. This is denoted by $W_k \rhd C(R_i)$.*

**Definition 6** *If an antecedent of production $R_i$ tests for the presence of a WME $W_k$, this is a* **positive** *test, notated by $S_{A(R_i)}[W_k] = +$, which is read as "$R_i$ has at least one antecedent that tests for the presence of a WME of type $T[W_k]$". In a similar fashion, if the test is for absence of $W_k$, it is a* **negative** *test, denoted by $S_{A(R_i)}[W_k] = -$.*

**Definition 7** *When the consequent of a production specifies the addition of a WME $W_k$ to Working Memory, it is a* **positive** *action, denoted by $S_{C(R_i)}[W_k] = +$. A* **negative** *action specifies the deletion of a WME $W_k$, denoted by $S_{C(R_i)}[W_k] = -$.*

**Consequence 1** *The notation $S_{A(R_i)}[W_k] \neq S_{C(R_j)}[W_l]$ implies that $W_k \rhd A(R_i)$, $W_l \rhd C(R_j)$, and that the $R_i$ test of $W_k$ is positive (negative) while $R_j$ action on $W_l$ is negative (positive).*

In the processing model discussed in section 3.2 some productions fire locally while others need to change WMEs that are stored in the local memory of remote processors. The following definitions describe important situations that appear in the execution of the model.

**Definition 8** *A WME $W_k$ is* **local** *to a processor $P_i$ iff $W_k$ is stored in the local memory of $P_i$; $W_k$ is not stored in the local memory of any other processor $P_j$; and there is no production allocated to a processor other than $P_i$ that changes $W_k$.*

**Definition 9** *A WME $W_k$ is* **pseudo-local** *to a processor $P_i$ iff $W_k$ is stored in the local memory of $P_i$; $W_k$ is not stored in the local memory of any other processor $P_j$; and there is at least one production allocated to $P_j \neq P_i$ that changes $W_k$. We say that $P_j$* **shares** *$W_k$.*

For example, a WME that is written by many processors and read by only one processor is *pseudo-local* for the processor that reads it; it is a *shared* WME for all processors that write it. A processor does not store shared WMEs in its local memory.

**Definition 10** *A production $R_n$* **fires locally** *in a processor $P_i$ iff $\forall\, W_k \rhd C(R_n)$, $W_k$ is local or pseudo-local to $P_i$.*

**Consequence 2** *A production that does not fire locally, is said to be a* **global production**. *Such a production must propagate actions to remote processors.*

**Definition 11** *A production $R_n$* **enables** *a production $R_m$ iff $\exists\, W_k$ such that $S_{C(R_n)}[W_k] = S_{A(R_m)}[W_k]$. A production $R_n$* **disables** *a production $R_m$ iff $\exists W_k$ such that $S_{C(R_n)}[W_k] \neq S_{A(R_m)}[W_k]$.*

**Definition 12** *A production $R_n$ has an* **output conflict** *with a production $R_m$ iff $\exists W_k$ such that $S_{C(R_n)}[W_k] \neq S_{C(R_m)}[W_k]$.*

Productions that can fire locally are classified as Independent of Network Transactions (INT) or Dependent on Network Transactions (DNT), according to their dependencies with other productions that belong to other processors. INT and DNT productions have to be mapped and processed differently for correct execution according to the serializability criterion. Productions are partitioned into disjoint sets with one set assigned to each processor. $R_n \in P_i$ indicates that production $R_n$ *belongs* to processor $P_i$. The Working Memory is distributed among the processors in such a way that a processor stores in its local memory all and only the WMEs tested by its productions.

**Definition 13** *A production that can fire locally is DNT if and only if at least one of the following conditions holds:*

**(i)** *two of the antecedents of the production are changed by the consequents of a single production allocated to another processor: one of these changes produces an enabling dependency and the other produces a disabling one;*

**(ii)** *the production has two conflicting writes with a production allocated to another processor;*

**(iii)** *the production has an output conflict and a disabling dependency with a production allocated to another processor.*

At compile time, after the set of productions is partitioned among the processors, the set of antecedents and the set of consequents of each production are analyzed to determine whether the production is global, local INT, or local DNT. To check if a production is local DNT is a simple matter of checking if any of the conditions of definition 13 holds.

**Definition 14** *A production $R_n$ is* **INT** *iff $R_n$ can fire locally and $R_n$ is not* **DNT**.

An INT production can start firing at any time as long as its antecedents are satisfied. A DNT production $P_i$ only starts firing after all tokens generated by a production $P_j$, currently being fired by a remote processor, are broadcast in the network and consumed by the processor that fires $P_i$. This prevents $P_i$ and $P_j$ actions from being intermingled, avoiding thus non-serializable behavior.

### 3.2 System Overview

The architectural model proposed in this paper consists of a moderate number of processors interconnected through a broadcasting network. The set of productions is partitioned among these processors with each

production assigned to exactly one processor. A processor reads data only from its local memory, i.e., no read operations are performed over the network. Due to the absence of network reads and the low frequency of network writes, a simple bus should be adequate as the broadcasting system. This conclusion is supported by detailed experimental results showing the bus not to be a bottleneck even for a twenty processor system. A number of associative memories implement a system of lookaside tables to allow parallel operations within each processor. This scheme does not allow parallel production firing within a processor, but allows the match-select-act phases of a PS to overlap. A snooping directory isolates the activities in remote processors from the activities in a local processor, and interrupts a local operation only when pieces of data that affect it are broadcast over the network.

The parallel architecture is formed by identical processors connected via a Broadcasting Interconnection Network (BIN), as shown on Figure 1. At start-up the I/O processor (I/OP) loads the productions on all processors. System level I/O and user interface are also handled through the I/OP. The main components of each processor are the Snooping Directory (SD), the Matching Engine (ME), the Production Controller (PC) and the Instantiation Controller (IC). Whenever a processor $P_i$ needs to broadcast a change to a WME that is stored in other processors local memories, $P_i$ creates a token to broadcast in BIN. The Snooping Directory is an associative memory that identifies whether a token broadcast on BIN conveys an action relevant to the local processor. Relevant tokens are kept in a Broadcasting Network Buffer (BNB) until the IC and the ME are able to process it. The Matching Engine is a Rete-based matcher that implements a state-saving algorithm. The IC uses specialized memory structures to maintain and rapidly update the list of fireable instantiations. To perform this task, it has to monitor the outputs of ME as well as the firing of local (through PC) and global (through SD) productions. One of the memories controlled by IC is the Firing Instantiation Memory (FIM) that keeps a list of all the production instantiations that are enabled to fire. The Production Controller (PC) selects an instantiation to be fired from the list maintained by IC, and, whenever necessary, synchronizes the production firing with BIN operations to guarantee that production firings appear to be atomic.

Productions are divided in three categories: local INT, local DNT, and global. The firing of a local INT production does not require BIN ownership because all its actions modify local WMEs only. Therefore, upon selecting an INT production, the PC immediately propagates its actions to ME and IC. To avoid interleaving of actions belonging to distinct productions, all tokens broadcast in BIN during local production firing are buffered in BNB. These tokens are processed as soon as the local firing finishes. When a local DNT production is selected, its execution has to wait until the BIN changes ownership, which is an indication that the firing of a global production has been concluded. The local DNT production is then fired in the same fashion as a local INT.

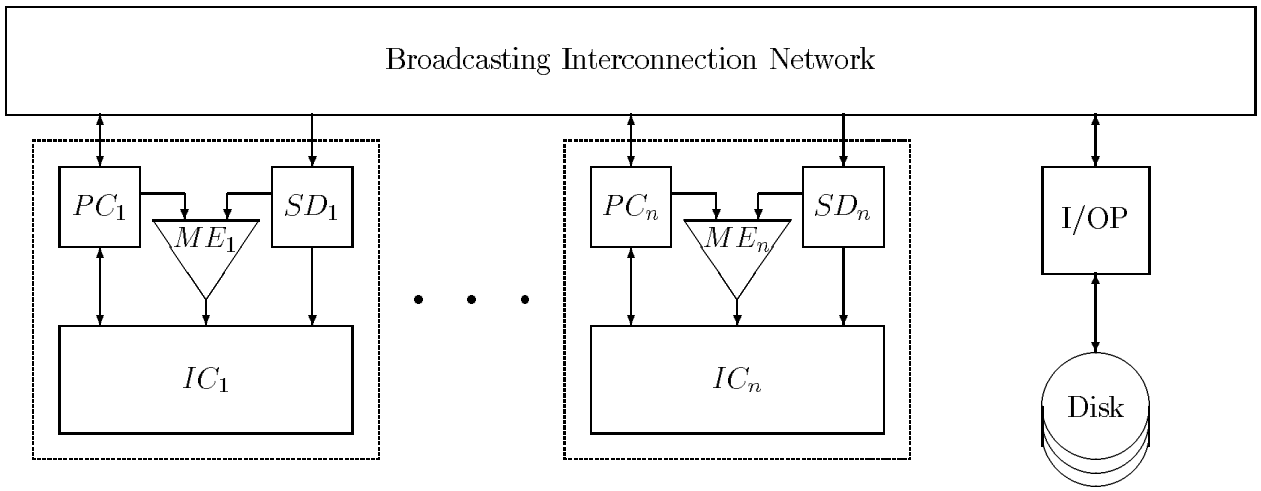


Figure 1: Parallel Machine Model

A global production modifies shared WMEs, i.e., WMEs that belong to the antecedents of productions assigned to other processors. Thus, these changes need to be broadcast to all processors. When a global production is selected, PC acquires access to the BIN, processes all outstanding changes in the BNB, and, if the selected production is still fireable, proceeds to broadcast tokens with changes to shared WMEs. The BIN ownership is not released until all actions that change shared WMEs are broadcast. After releasing the BIN, PC prevents any incoming token from proceeding to local processing. These tokens are buffered in BNB and processed locally after the local execution of the selected production is complete. This avoids write interleaving in the local memories and guarantees an atomic operation for production firing within a processor.

The main steps in the machine operation are presented below in an algorithmic form. The steps of the algorithm are performed by different structures of the processing element.

**PRODUCTION-FIRING**

1. **execute** all outstanding tokens in BNB on first-come first-serve basis
2. **select** a fireable instantiation $I_k$ in FIM
3. **if** $I_k$ is global
4. **then** Request BIN ownership
5. **while** BIN ownership is not granted
6. **execute** tokens broadcast in BIN captured by SD
7. **if** $I_k$ is still fireable
8. **then broadcast** actions that change shared WMEs
9. **execute** actions that change shared WMEs

10. **release** BIN
11. **else end** PRODUCTION-FIRING
12. **else if** $I_k$ is DNT
13. **then while** BIN ownership does not change
14. **execute** tokens broadcast in BIN captured by SD
15. **if** $I_k$ is still fireable **and** $I_k$ has local actions
16. **then disable** local execution of any incoming token
17. **execute** local actions
18. **enable** local execution of incoming tokens

Note that no production is fired while there are outstanding tokens in BNB. The selection of a fireable instantiation in step 2 of PRODUCTION-FIRING is done according to the "pseudo-recency" criterion: the most recent instantiation in FIM is selected This is not a true recency criterion because ME may still be processing a previous token, and thus the instantiations that it will produce are not in FIM yet.

The test in step 7 is necessary because between the time the BIN was requested and the time its ownership is acquired, incoming tokens might have changed the status of the production selected to fire. If this occurs, the firing of the selected production is aborted. Steps 12-14 are executed for productions that are dependent on network transactions, as defined in section 3.1. If such productions were to start firing while a remote processor is in the middle of a production execution, the intermingling of actions could result in non-serializable behavior. Notice that the BIN is released in step 10, before changes to local memory take place. To guarantee that no token is processed before the local changes are executed, buffering of tokens in BNB in step 15 is activated immediately upon releasing the BIN.

The architectural model presented in this section bears some similarity with the systems proposed by Schmolze and Goel [37] and Ishida *et al.* [21]. In all three systems, each production is uniquely assigned to one processor and all WMEs tested by the production are stored locally. Contrary to the architecture presented in this paper, the systems proposed in [21] and [37] use a taxing synchronization mechanism and require each processor to keep a list of all dependencies that each production has with other processors. The bus-based architecture with snoopy mechanism presented in this paper substantially simplifies synchronization and avoids the potential for incorrect behaviour or deadlock. Similar synchronization mechanisms are nowadays employed for cache coherency in several commercial medium-scale multiprocessor systems [18].

### 3.2.1 Detailed Processor Model

The processor architecture is detailed in Figure 2. The Instantiation Firing Engine (IFE) implements the outgoing interface with the Broadcasting Interconnection Network (BIN) and synchronizes internal activities. The IFE selects an instantiation to be fired among the ones stored in the Fireable Instantiation Memory (FIM). If the production selected to fire is global, the IFE places a request for ownership of the BIN. Upon receiving BIN ownership, IFE waits until all outstanding tokens stored in BNB are processed. If the selected instantiation becomes unfireable due to such processing, IFE has to abandon it and select a new instantiation. Otherwise IFE broadcasts tokens with changes to the shared WMEs, releases the BIN, and executes the local actions.

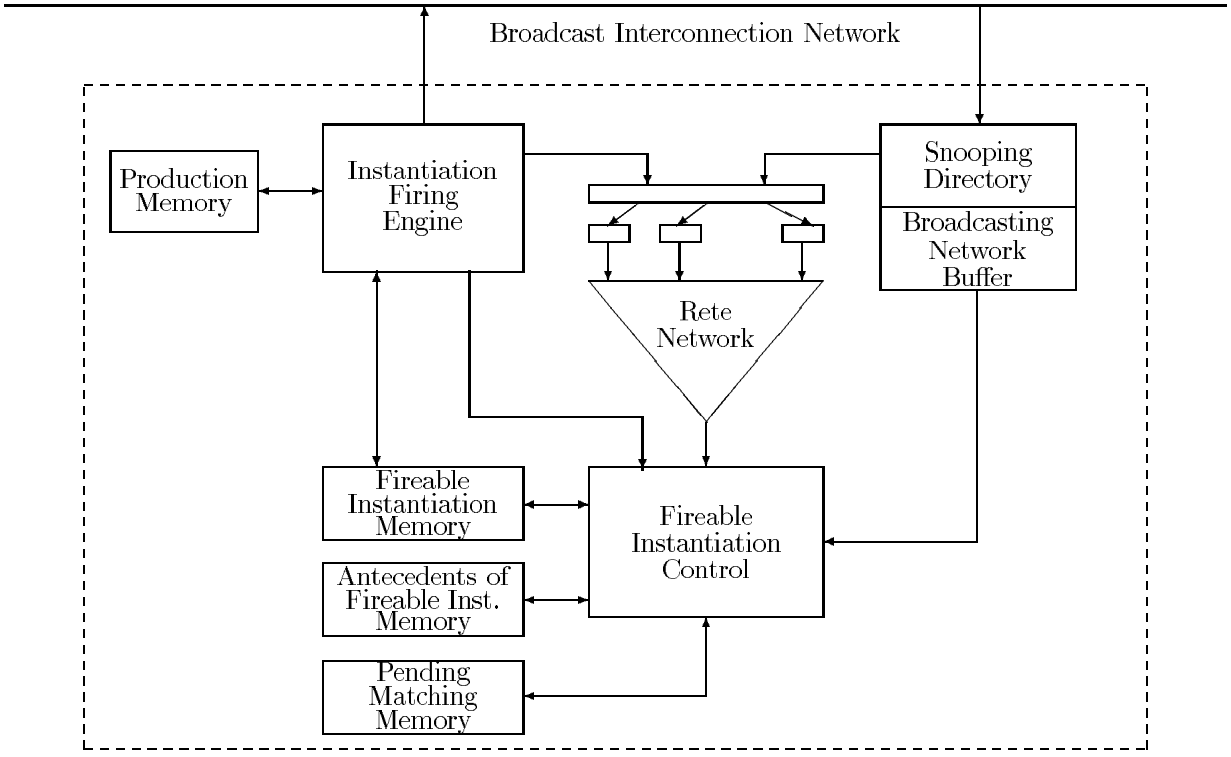


Figure 2: Processing Element Model

The Snooping Directory (SD), along with the Broadcasting Network Buffer (BNB), implements the incoming network interface. The Snooping Directory is an associative memory that contains all WME types that belong to the antecedent sets of the productions assigned to the processing element. BNB is used to store tokens broadcast on BIN and captured by SD during the local firing of a production, or during the execution of local actions of a global production. The tokens stored in BNB are processed as soon as the firing of the current production finishes. In the rare situation in which BNB is full, a halt

signal is issued to freeze the activity on BIN. When the halt signal is reset, the activity in the bus resumes: the same processor that had BIN ownership continues to broadcast tokens as if nothing had happened.

Whether a WME change is originated locally or captured from BIN, it needs to be forwarded to the Rete network and to the Fireable Instantiation Control (FIC). Like the original Rete network, the one used in this architecture has $\alpha$ and $\beta$-memories. To avoid the high cost of waiting for the removal of a WME, which was pointed out by Miranker [31], negated antecedents are stored in both $\beta$-memories and in the fireable instantiations produced for the conflict set. The presence of the negated conditions in this representation allows the quick removal of non-fireable instantiation when a new token is processed. There is a possibility that a WME change previously processed by FIC and not yet processed by Rete disables an instantiation freshly generated by Rete. To avoid a possibly non-serializable behavior, before adding a new instantiations to FIM, FIC checks it against the Pending Matching Memory (PMM), which stores all tokens still to be processed by Rete. The deletion of an instantiation from FIM is also performed by FIC. The operation of FIM, AFIM, PMM and FIC are explained in greater detail in section 3.2.2.

### 3.2.2 Conflict Set Management

The Fireable Instantiation Control (FIC) uses the Antecedents of Fireable Instantiation Memory (AFIM) to maintain a list of all enabled instantiations in the Fireable Instantiation Memory (FIM). AFIM and FIM are fully associative memories with capability to store don't cares in some of their cells. The fields in each line of FIM and AFIM are shown in Figure 3. FIC maintains an internal timer that is used to time stamp each instantiation added to FIM. Each line of AFIM stores either a WME that is the antecedent of a fireable instantiation, or an $\alpha$-test that specifies an instantiation negated antecedent. Its fields are:

**Presence -** indicates whether the AFIM line is occupied. It is used to manage the space in the memory.

**Negated -** indicates whether this line stores a WME or a negated antecedent.

**Type -** stores the WME type.

**Bindings -** contains the values stored in each attribute-value pair of the WME. Notice that the name of the attribute does not need to be stored. Symbolic names are translated into integer values at compile time.

**$\alpha$-test -** is used only for negated antecedents: specifies the $\alpha$-test to be performed to verify a production antecedent.

**Instantiation -** indicates which fireable instantiations have this antecedent.

The maximum number of attribute-value pairs in a single WME is limited by the size of the field *Bindings* in AFIM. A situation in which a WME has more attribute-value pairs than this limit is handled at compile time by splitting this WME into different WMEs with subsets of attribute-value pairs and performing the corresponding changes in the entire source code.

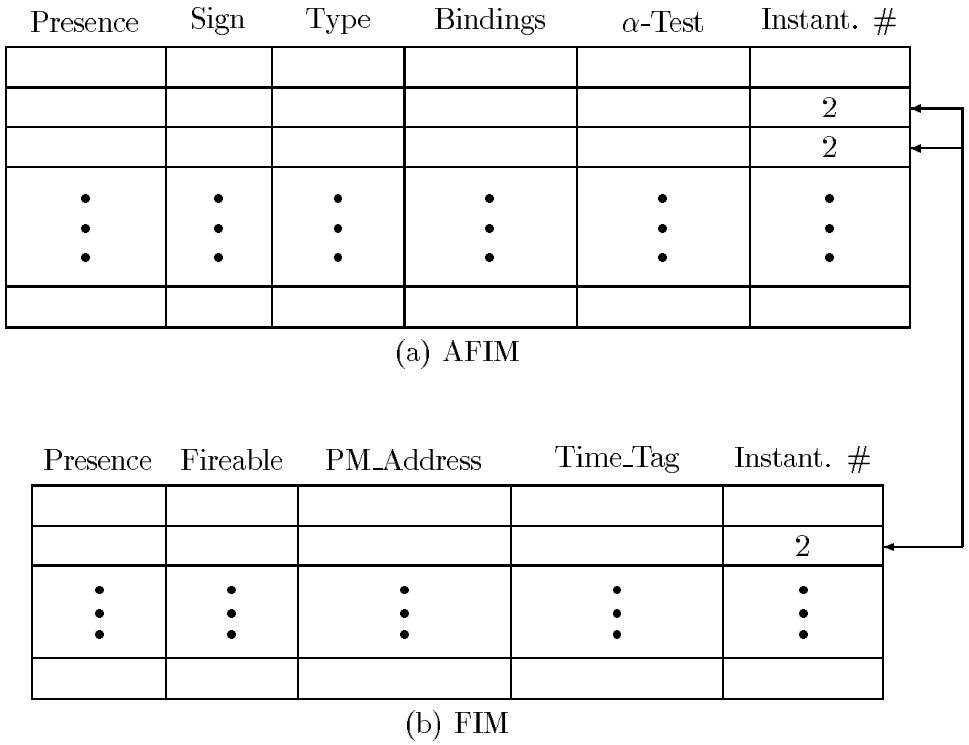


Figure 3: (a) Antecedents of Fireable Instantiations Memory; (b) Fireable Instantiations Memory.

Notice that because AFIM stores antecedents of fireable instantiations, most of the variables are bound, therefore the *bindings* field stores mostly constants. For an easy handling of unbound variables, which match any value, the *bindings* field of AFIM is a ternary memory. Besides the values 0 and 1, it can also store a "don't care" value X. Such a memory might be implemented using two bits per cell, or using actual ternary logic in VLSI. One example of the latter is the Trit Memory developed by Wade [43]. One alternative to implement a non-bound value is to add a tag bit to *bindings* that indicates whether the value is bound or not. The advantage of this representation is that there is only one extra bit per word. Each line in FIM stores one fireable instantiation, with the following fields:

**Presence -** indicates whether the line is occupied;

**Fireable -** indicates whether the instantiation stored in the line is still fireable[4].

**PM_Address -** contains a pointer to the Production Memory indicating where the production actions are stored.

**Time_Tag -** record the time in which the instantiation became fireable. It is used to implement a *pseudo-recency criterion* to select an instantiation to be fired.

The third piece of memory managed by FIC is a fully associative memory called Pending Matching Memory (PMM). When a token is placed in the input nodes of the Rete network, it is also stored in PMM. The token is removed from PMM when the Rete network produces a signal indicating that all changes to the conflict set originated by that token have being processed. Upon receiving a new fireable instantiation from Rete, FIC associatively searches PMM. FIC has to perform an independent search for each antecedent of the new instantiation. If any line of PMM indicates the deletion (addition) of a WME that matches a non-negated (negated) condition of the instantiation, the new instantiation is ignored[5]. If no such line is found in PMM, FIC records the new instantiation in one line in FIM and stores each one of its antecedents in a separate line in AFIM. Figure 4 shows the organization of PMM with four fields:

**Presence -** indicates whether there is a WME stored in the line.

**Sign -** indicates whether this WME has been added to or deleted from the working memory.

**Type -** stores the type of WME.

**Bindings -** records the bindings of the WME.

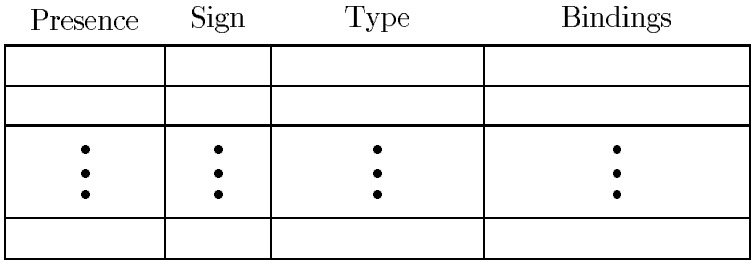


Figure 4: Pending Matching Memory

[4]An instantiation is only removed from FIM after an incremental garbage collector removes the corresponding antecedents from AFIM.

[5]This instantiation must be ignored because the entry found in PMM indicates that a token received after the one that enabled the instantiation, which is not yet fully processed in Rete, will disable it.

During the execution of a token, FIC performs three actions in parallel: send the token to the Rete network input; add the token to PMM; and update FIM and AFIM. To update AFIM and FIM, first FIC executes an associative search in AFIM for entries with the same WME present in the token, but with opposite sign. For each matching entry in AFIM, FIC marks the corresponding instantiation in FIM as unfireable. Finally FIC resets the presence bit for these entries in AFIM. This process leaves "garbage" in FIM and AFIM, consisting of all the non-fireable instantiations still present in FIM plus the antecedents of these instantiations in AFIM.

FIC has an *Incremental Garbage Collector* that searches FIM for an instantiation $I_k$ that is non-fireable. FIC performs an associative search in AFIM and remove all antecedents of $I_k$, and then eliminates $I_k$ from FIM. To guarantee the consistency of FIM and AFIM, the garbage collection is always performed as an atomic operation. For efficiency, the position in FIM in which the last garbage collection was executed is kept internally in FIC, and is used as the starting point of the next search. If FIM and AFIM are not full, garbage collection is performed at least once between two instantiation additions. Whenever FIM or AFIM are full, extra garbage collection is executed to free space. This solution trades memory space for speed: a WME that is tested by antecedents of many instantiations is stored many times in AFIM.

### 3.2.3 Broadcasting Interconnection Network Arbitration

Access arbitration in a broadcasting network is a well studied problem. In this machine we adopt the scheme used in the first prototype of the Alpha architecture by DEC [42]. During startup each processor is assigned an arbitrary priority number from 0 to $N$. $N$ is the highest priority and 0 is the lowest. When a processor requests the network, it uses its priority. The requester with highest priority is the winner and is granted access to the network. The winner has possession of the network as long as it needs to write all consequents of *one* production. After releasing the network, the winner sets its own priority to zero. All processors that had a priority number less than the winner increment their priority number by one, regardless of whether they made a request.

This scheme works as a round robin arbitration if all processors are requesting the network at the same time. If fewer processors are requesting the network, this mechanism creates the illusion that only these active processors are present in the machine.

In section 3.2 we establish that broadcast writes need to be kept in a buffer while a processor is firing local productions. When this buffer overflows, a *halt* signal is issued by the processor. This signal stalls all network broadcasting activities, giving time for the overloaded processor to consume its tokens and alleviate its buffer load. When the stall signal is removed, the network continues its activity without

any change in the ownership. To avoid a great impact in the speed of the machine, the buffer must be sufficiently large to avoid frequent stalling of the network.

### 3.3 Correctness of the Processing Model

This section investigates whether the machine proposed in section 3 correctly executes a production system. The correctness criterion used is serializability [36] and the condition of ownership is stated in axiom 1.

**Axiom 1** *A WME $W_k$ is stored in the local memory of a processor $P_i$ iff $W_k \rhd A(R_n)$ and $R_n \in P_i$.*

**Theorem 1** *Giving the parallel machine model presented in this document, the definition of local DNT, local INT, and global productions, Axiom 1 is a necessary and sufficient condition of ownership to guarantee correct execution of a production system under the serializability criterion of correctness.*

**Proof**:

First we prove that axiom 1 is necessary. For the sake of contradiction, suppose that the ownership condition stated in axiom 1 is not satisfied. Assume that there is a production $R_n \in P_i$ and a WME $W_k$, such that $W_k \rhd A(R_n)$ and $W_k$ is not stored in the local memory of $P_i$. Because reading operations are not allowed in the broadcasting network, $P_i$ cannot perform the matching of $R_n$. Therefore a production system cannot be executed in such a machine. Thus, axiom 1 is necessary.

To prove that axiom 1 is sufficient, we must show that, in every possible circumstance, the results produced by this model could be obtained by a sequential execution of the productions. Therefore, we must analyze all situations in which parallel execution might occur and show that each one of them results in a serializable outcome. Because there is no parallel production firing within a processor, the following analysis is restricted to concurrent firing of productions allocated to distinct processors. Inter-processor parallelism occurs in two situations: among productions firing locally in distinct processors and between a production being broadcast over the BIN and one (or more) firing locally. All situations described below involve two productions allocated to distinct processors being fired concurrently.

**Situation 1:** *Productions that have only local WMEs in its antecedents and consequents.*

The fact that all antecedents and consequents are local indicates that the productions being fired in parallel are completely independent of productions allocated to other processors, therefore

the same results produced by the parallel firing could be obtained by any sequential firing of the same productions.

**Situation 2:** *A production $R_m \in P_j$ enables a production $R_n \in P_i$; $R_m$ and $R_n$ might have non-conflicting shared outputs; $R_m$ does not disable $R_n$; $R_n$ fires locally.*

Since $R_n$ fires locally, all WMEs that are changed by both $R_n$ and $R_m$ are pseudo-local for $P_i$ and shared for $P_j$. Because those are non-conflicting outputs and $R_m$ enables $R_n$, parallelism occurs when $R_n$ starts firing after being enabled by an action of $R_m$ and before $R_m$ finishes broadcasting changes to the network. The firing of $R_n$ prevents the changes broadcast by $R_m$ from being processed locally until $R_n$ finishes. As long as the actions broadcast by $R_m$ are queued and processed after $R_n$ finishes, the result is the same as if $R_n$ would have been fired after $R_m$ finished. Thus, it is serializable.

**Situation 3:** *A production $R_m \in P_j$ disables a production $R_n \in P_i$; there is no enabling dependencies between $R_m$ and $R_n$; $R_m$ and $R_n$ might have non-conflicting shared outputs; $R_n$ fires locally.*

The only possibility for the parallel firing of $R_m$ and $R_n$ is for $P_i$ to start firing $R_n$ before $P_j$ had broadcast any action that disables $R_n$. Even if $P_j$ had broadcast some of the shared non-conflicting outputs when $R_n$ starts firing, the effect is the same as firing $R_n$ before $R_m$. Therefore, the result is serializable.

**Situation 4:** *A production $R_n \in P_i$ changes a pseudo-local WME $W_k$ and a production $R_m \in P_j$ modifies $W_k$. $R_n$ fires locally.*

Because $R_m$ modifies $W_k$, $R_m$ is a global production. It is necessary to analyze three different cases:

Case 1: $W_k$ is the only shared output between $R_n$ and $R_m$.

Notice that the (possibly) conflicting WME $W_k$ is exclusively stored in $P_i$. Therefore if $P_i$ disables the BIN before $P_j$ broadcast changes to $W_k$, the result is the same of firing $R_n$ before $R_m$. If $P_i$ disables BIN after changes to $W_k$ are broadcast, the result is equivalent to firing $R_n$ after $R_m$. In both cases it is serializable.

Case 2: $R_n$ and $R_m$ have more than one shared output, but no more than one of them is conflicting.

The concern with multiple shared outputs is that the actions of the local and the global production might be intermingled. This would happen if $P_i$ would inhibit actions from the network after $P_j$ broadcast some but not all actions of $R_m$. Since $R_m$ has only one action conflicting with $R_n$, the interruption of the remote firing will either take place before or after this conflicting action is broadcast. If the interruption occur before the conflicting action is executed in $P_i$, the result is equivalent to $R_n$ firing before $R_m$. If it occurs after, the result is equivalent to $R_n$ firing after $R_m$. In either case this situation results in a serializable behavior.

Case 3: $R_n$ and $R_m$ have more than one conflicting action.

In this case, if intermingled execution would be allowed, non-serializable behavior would result. However, according with condition (ii) of definition 13 $R_n$ is DNT and therefore cannot start firing until the network changes ownership, indicating that the global production either has finished or has not started. This ensures the necessary synchronization and results in serializable behavior.

**Situation 5:** *A production $R_n \in P_i$ is enabled and disabled by a production $R_m \in P_j$; $R_n$ fires locally.*

In this situation, there would be a non-serializable behavior if production $R_n$ would be allowed to fire after $P_j$ had broadcast the action that enables $R_n$ and before the action that disables $R_n$ is broadcast. This situation does not occur because, according to condition (i) of definition 13, $R_n$ is DNT: it only starts firing when the network changes ownership.

**Situation 6:** *A production $R_n \in P_i$ is enabled by a production $R_m \in P_j$; $R_n$ has one output conflict with $R_m$; $R_n$ and $R_m$ may or may not have shared non-conflicting outputs; and $R_n$ fires locally.*

Parallelism occurs if $R_n$ starts firing in $P_i$ after the action that enables $R_n$ have been broadcast by $P_j$ and before $P_j$ finishes broadcasting $R_m$ actions. If at that point the conflicting action has been already broadcast, the result will be equivalent to firing $R_n$ before $R_m$. If the conflicting action has not been broadcast, the result is equivalent to $R_m$ firing before $R_n$. Either way, the result is serializable.

**Situation 7:** *A production $R_n \in P_i$ is disabled by a production $R_m \in P_j$; $R_n$ has one output conflict with $R_m$; $R_n$ and $R_m$ may or may not have shared non-conflicting writes; $R_n$ fires*

*locally.*

This situation could result in non-serializable behavior if $R_n$ were to start firing after $P_j$ broadcasts the conflicting action of $R_m$, and before the action that disables $R_n$ is broadcast. However, this cannot occur because, according to condition (iii) of definition 13, $R_n$ is DNT.

Situations 1 through 7 deal with possible dependencies involving two productions $R_m$ and $R_n$ allocated to distinct processors. The local firing of $R_n$ in all situations indicates that its consequents change only local or pseudo-local WMEs. Table 1 helps to verify that every possible combination of dependencies among two productions in this situation have being analyzed. In this table a "-" indicates no dependencies, "1" indicates one dependency, "1+" indicates one or more dependencies, "2+" indicates two or more dependencies, and "X" indicates zero or any number of dependencies. Table 1 has five columns: "Enabling" column indicates the number of actions in $C(R_m)$ that enable $R_n$; "Disabling" indicates the number of actions in $C(R_m)$ that disable $R_n$; "Non-Conflicting Write" indicate the number of non-conflicting shared actions between $R_n$ and $R_m$; "Non-Conflicting Write" indicate the number of non-conflicting shared actions between $R_n$ and $R_m$; and "Situation" indicates which of the situations analyzed in this proof covers each case. Every possible combination of dependencies between two productions is covered in table 1.

| Enabling | Disabling | Non-Conflicting Write | Conflicting Write | Situation |
|---|---|---|---|---|
| - | - | - | - | 1 |
| 1+ | - | X | - | 2 |
| - | 1+ | X | - | 3 |
| - | - | 1+ | - | 4, case 1 |
| - | - | X | 1 | 4, case 2 |
| X | X | X | 2+ | 4, case 3 |
| 1+ | 1+ | X | X | 5 |
| 1+ | - | X | 1 | 6 |
| X | 1+ | X | 1 | 7 |

Table 1: Possible dependencies between $R_n$ and $R_m$.

There is still the possibility that dependencies involving more than two productions create a situation in which the parallel model yields a non-serializable behavior. The only situation in which this might occur are in cycles of disablings, analyzed in situation 8.

**Situation 8:** *There is a cycle of disablings among productions allocated to distinct processors.*

First we analyze the special case in which the cycle is formed by two productions $R_n \in P_i$ and $R_m \in P_j$. According to definition 11, if there is a cycle of disabling between $R_n$ and $R_m$, there exist two WMEs $W_k$ and $W_l$ such that $S_{C(R_m)}[W_k] \neq S_{A(R_n)}[W_k]$ and $S_{C(R_n)}[W_l] \neq S_{A(R_m)}[W_l]$. Therefore $W_k$ is a shared WME for $P_j$, $W_l$ is a shared WME for $P_i$, and neither $R_m$ or $R_n$ can fire locally. The acquisition of the broadcasting network works as a synchronizing element preventing $R_n$ and $R_m$ from firing in parallel. The same reasoning can be extended to disabling cycles with any number of productions.

This concludes the proof. Since the results are serializable for any possible conflicting situation, we conclude that Axiom 1 is a sufficient condition of ownership and that the results produced by the model proposed are serializable.

□

# 4 Production Partitioning Algorithm

The problem of partitioning a Production System into disjoint production sets which are then mapped onto distinct processors has been studied by a number of researchers. Most partitioning algorithms are designed with the goal of reducing enabling, disabling and output dependencies among productions allocated to different processors [37]. Oflazer formulates partitioning as a minimization problem and concludes that the best suited architecture for Production Systems has a small number of powerful processors [35]. Oflazer also indicates that a limited amount of improvement in the PS speed can be obtained by an adequate assignment of productions to processors. Moldovan presents a detailed description of production dependencies and expresses the potential parallelism in a "parallelism matrix" and the cost of communication among productions in a "communication matrix" [32]. Xu and Hwang use a similar scheme with matrices of cost to construct a simulated annealing optimization of the production partition problem [44].

Although certain basic principles are maintained in all partitioning schemes, partition algorithms are tailored to specific architectures. We are concerned with two kinds of relationships among productions: productions that share antecedents, and productions that have conflicting actions. Assigning productions with common antecedents to the same processor reduces memory duplication, while assigning productions with conflicting actions to the same processor prevents traffic in the bus. Previous partition algorithms were greatly influenced by enabling and disabling dependencies among productions [32, 35, 44]. Our experience with production systems shows that grouping productions with common antecedents is much

more effective to reduce the communication cost. Moreover, in the production system programs that we examined, a production seldom creates a WME that was not tested on its antecedents. Therefore, productions that have a greater number of common antecedents are also most likely to have a greater number of enabling and disabling dependencies among them. Thus, our partition algorithm does not include these dependencies, but only shared antecedents and conflicting outputs.

We analyzed and experimented with several partitioning algorithms and found the following algorithm to be the most effective [4, 5]. The optimal partitioning of productions into disjoint sets is modeled as a minimum cut problem, which is NP-Complete [12]. The polynomial time approximate solution presented in this section has three goals: minimizing the duplication of working memory elements; reducing traffic in the bus; and balancing the amount of processing in each processor. In the architecture presented in section 3 these goals translate to: minimizing the number of global productions and reducing the number of local DNT production. As a consequence, the number of local INT productions is increased.

To represent the relationships among productions we define an undirected, fully connected graph PRG $= (P, E)$ called *Production Relationship Graph*. Each vertex in $P$ represents one of the productions in the system, and each weighted edge in $E$ is a combined measure of the production relationships. PRG has a weight function $w : E \rightarrow Z^+$, defined by equation 1.

$$w(E_{ij}) = w(E_{ji}) \quad = \quad (1 - \delta_{ij}) \sum_{l=0}^{n-1} \sum_{k=0}^{m-1} \psi_{li,kj} \; + \; (1 - \delta_{ij}) \sum_{l=0}^{p-1} \sum_{k=0}^{q-1} \gamma_{li,kj}, \tag{1}$$

where $n$ and $m$ are the number of antecedents and $p$ and $q$ are the number of consequents in productions $R_i$ and $R_j$, respectively, $\delta_{ij}$ is 1 if $i = j$ and 0 otherwise, and

$$\psi_{li,kj} \quad = \quad \begin{cases} 1 & \text{if antecedents } A_l \text{ of } R_i \text{ and } A_k \text{ of } R_j \text{ are of the same type.} \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{li,kj} \quad = \quad \begin{cases} 1 & \text{if consequent } W_l \text{ of } R_i \text{ conflicts with } W_k \text{ of } R_j \\ 0 & \text{otherwise} \end{cases}$$

Empirical studies with a parallel architecture simulator show that the main factor limiting further reduction is the time spent in the matching phase in the Rete network. Consequently, the load balancing must concentrate on the processing performed in the Rete network. Furthermore, most of the time in the Rete network is spent in $\beta$-node activities. Thus, the number of $\beta$-tests performed in the antecedents of a production is used as a measure of the workload associated with this production. To address the constraint of balancing the amount of processing among processors, we define the function $B : P_0, ..., P_{N-1} \rightarrow Z^+$, which computes the number of beta tests that are expected to be performed by processor $P_i$.

$$B(P_i) \quad = \quad \sum_{j=0}^{N-1} \beta(R_j)\, \varphi_{ij}, \tag{2}$$

where $\beta(R_j)$ is the number of beta tests performed for production $R_j$, and $\varphi_{ij}$ is 1 if $R_j$ is assigned to $P_i$, and 0 otherwise[6]. $N$ is the total number of productions in the system.

Let $S_i$ denote the set of productions assigned to processor $P_i$. When the algorithm starts, all subsets $S_i$ are empty and all productions are in the set $S$. The fitness of placing production $R_i$ in set $S_k$ is measured by the value of the function $F(R_i, S_k)$.

$$F(R_i, S_k) \quad = \quad \sum_{j=0}^{N-1} w(E_{ij})\, \eta_{jk}(1 - \delta_{ij}), \tag{3}$$

$$\eta_{jk} = \begin{cases} 2 & \text{if } R_j \in S_k \\ 1 & \text{if } R_j \in S \\ -1 & \text{if } R_j \in S_m \neq S_k, \end{cases}$$

The value of the fitness function indicates how the production represented by the vertex $R_i$ fits in the subset $S_k$. $F(R_i, S_k)$ computes a weighted sum of the connections between vertex $R_i$ and all other vertices in PRG. A strong connection with a vertex that has been assigned to a set other than $S_k$ reduces the fitness of $R_i$ to $S_k$, while a strong connection with a vertex already in $S_k$ increases the fitness. A strong connection with a vertex that has not been assigned to any subset has an intermediate value because $S_k$ may be able to attract both vertices.

The strategy used in this partitioning algorithm consists of selecting the processor with the least number of estimated beta tests, and then finding the production best fitted to this processor. The productions strongly related to other productions in PRG are the first ones to be assigned to processors. The algorithm ends when there are no more productions in $S$.

**PARTITION**$(S, E, w, N, B, F)$
1 **while** $S \neq \emptyset$
2 **do** $S_k \leftarrow S_k \cup \{\ R_i\ /R_i \in S$ and
$B(P_k) = \min_k B(P_k)$ and
$F(R_i, S_k) = \max_i F(R_i, S_k)\}$

[6] $\beta(R_j)$ is an estimate of the number of beta tests performed because of the presence of production $R_j$. It is measured in previous runs of the same production system.

3 $S \leftarrow S - \{R_i\}$

# 5 Performance Evaluation

Performance evaluation can be accomplished through measurement, simulation, and analytic modeling [23]. Measurement consists of observing actual values for specified parameters in an existing system. Simulation consists in creating a model for the behavior of a system, writing a computer program that reproduces this behavior, feeding the simulator with an appropriate sample of the workload of the actual system, and computing selected parameters of interest. In analytic modeling a mathematical model of the system is created and its solution provides the performance evaluation [23]. In a related work, we used an analytical model to investigate the effect of using multiple functional units to update the Rete network within each processor [7].

In this research we use an event driven simulator to evaluate the speedup of the architecture proposed. The input of the simulator consists of production system programs written in OPS5 syntax. For syntax and lexical analysis, the tools *yyacc* and *yylex* were used[7].

## 5.1 Benchmarking

A well known weakness of production system machine research is the lack of a comprehensive and broadly used set of benchmarks for evaluation of performance. In the process of searching for benchmarks to evaluate this novel architecture, we contacted many researchers with the same problem: a new idea to be evaluated in need of a suitable set of benchmark programs. Most of the benchmarks obtained were toy programs with a small number of productions in which the researcher can only change the size of the database. A benchmark in which the number of productions and the database size can be independently changed would allow researchers to study various aspects of new architectures. Section 5.1.1 presents a new benchmark that has such characteristics. It is a modification of the well known Traveling Salesperson Problem that we call a *Contemporaneous TSP* (CTSP) [8]. Another benchmark that we wrote is a solution to the "Confusion of Patents Problem". The following sections describe CTSP in detail and briefly present some other benchmarks used to test the architecture.

[7]The front-end conversion of the OPS5 syntax into internal data structure was built by Anurag Acharya at Carnegie Mellon University for PPL [3, 2].

### 5.1.1 A Contemporaneous TSP

In this modified version of the TSP, the cities are grouped into "countries". The tour has to be constructed such that the salesperson enters each country only once. The location and borders of the countries must allow the construction of a tour observing this restriction. The problem is formally stated as follows:

> An instance of CTSP is represented by $(K, C, c, \mu_c, \sigma_c, O, d)$. $K = \{C_1, C_2, ..., C_n\}$ is a "continent" formed by "countries". Each country $C_i = \{c_{i,\tau(1)}, c_{i,\tau(2)}, ...., c_{i,\tau(m(i))}\}$ contains $m(i)$ "cities". The number of cities per country $m(i)$ is normally distributed with average $\mu_c$ and standard deviation $\sigma_c$. The ordering $O =< C_{\pi(1)}, C_{\pi(2)}, ..., C_{\pi(n)} >$ specifies the order in which the countries shall be visited. The function $d(c_{i,\tau(k)}, c_{j,\tau(l)}) \in Z^+$ specifies the distance between any two cities in the continent. The problem consists of finding an ordering of cities $< c_{i,\tau(1)}, c_{i,\tau(2)}, ..., c_{i,\tau(m(i))} >$ within each country $C_i$ that minimizes the cost of the global tour:

$$\sum_{i=1}^{n} \sum_{j=1}^{m(i)-1} d(c_{i,\tau(j)}, c_{i,\tau(j+1)}) + \sum_{i=1}^{n-1} d(c_{i,\tau(m(i))}, c_{i+1,\tau(1)}) + d(c_{n,\tau(m(n))}, c_{1,\tau(1)}). \quad (4)$$

This formulation of TSP is called "contemporaneous" because it reflects some aspects of modern day life. In the current global economy, travelpersons are likely to have greater needs than the traditional salesperson driving from town to town. Consider a music star in a worldwide tour carrying along a huge crew and sophisticated equipment: the singer will visit many different locations in each continent; the cost of flying back and forth between continents is much higher than movements within a continent and depends on the locations of departure and arrival. Other situations involving sophisticated traveling requirements include the planning of airline routes and national political campaigns in large countries such as USA, Brazil and India. Applications in which data locality allows the creation of clusters include: insurance database management, banking industry, a national health care information network, and a national criminal offense information network[8].

### 5.1.2 A Production System Solution for CTSP

The formulation presented above for the CTSP is generic enough to allow its application in many fields: there is no restriction in what the words continent, country, city, and distance might represent. To facilitate the construction of a Production System solution that is useful for testing new PS architectures, we used a simpler version of CTSP with the following restrictions:

[8] In the 1994 "Brady Bill", the USA Congress mandated the construction of such a network for background verification for the purchase of fire weapons.

- The problem is symmetric, i.e., $d(c_i, c_j) = d(c_j, c_i)$ for any $i$ and $j$.
- A continent is a two-dimensional Euclidian space.
- A country is a contiguous, rectangular shape within this space.
- The number of cities in each state follows a normal distribution with average $\mu_c$ and standard deviation $\sigma_c$.
- The city locations are uniformly distributed within each country.
- There is a common boundary between two countries that are consecutive in the ordering $O$.

Our PS solution for CTSP has a set of productions for each country and a set of productions for each country boundary. The data set is constructed in such a way that the distances among cities located within each country are stored in WMEs with different types. Given a country $C_i$, the country that precedes $C_i$ in the order $O$ is denominated $P(C_i)$; the country that succeeds $C_i$ in the order $O$ is denominated $S(C_i)$. It is not necessary to store in the data base the distance between every two cities in the continent. For a city $c_j$ in a country $C_i$, the only relevant distances are the distance to the cities within $C_i$, to the cities in $P(C_i)$, and to the cities in $S(C_i)$. The following list illustrates WMEs typically used in our solution to CTSP:

```
(GERMANY_city ^name GERMANY_01 ^status not_in_trip)
(FRANCE_city ^name FRANCE_10 ^status in_trip)
(GERMANY_distance ^from GERMANY_04 ^to GERMANY_07 ^value 135)
(FRANCE_GERMANY_distance ^from FRANCE_14 ^to GERMANY_03 ^value 357)
(GERMANY_POLAND_distance ^from GERMANY_01 ^to POLAND_05 ^value 55)
```

Our solution has seventeen *local* productions per country and twelve productions per country boundary. This organization allows the researcher to vary the number of productions by creating continents with different number of countries. The size of the data base is determined by the number of countries and the average number of cities per country. The variance between the amount of data processed by each cluster of production is given by $\sigma_c$.

The heuristic used in the PS solution of the problem involves the computation of two extra locations for each country $C_i$: the geometric center of the borders with $P(C_i)$ and with $S(C_i)$. Because we impose the restriction that countries have rectangular shapes in a two-dimensional Euclidian space, the border between two subsequent countries in the tour is always a segment of a straight line. The *border center*

$b(C_i, C_j)$ between countries $C_i$ and $C_j$ is the center of the line segment that forms the boundary. The heuristic used to construct the internal tour in a country $C_i$ is described below:

- The first city $c_k$ in the internal tour of a country $C_i$ is the city with minimum distance $d(b(C_i, P(C_i)), c_k)$.
- While the internal tour of country $C_i$ is not complete, select a city $c_l \in C_i$ such that $d(c_k, c_l) - d(c_l, b(C_i, S(C_i)))$ is minimum. $c_k$ is the last city inserted in the tour.
- Whenever the internal tours of two adjacent countries $C_i$ and $C_j$ are completed, the last city visited in $C_i$ is connected to the first city visited in $C_j$.
- Whenever there is a segment of tour formed by four cities $c_i \rightarrow c_j \rightarrow c_k \rightarrow c_l$ such that $d(c_i, c_j) + d(c_k, c_l) > d(c_i, c_k) + d(c_j, c_l)$, change this segment of tour to $c_i \rightarrow c_k \rightarrow c_j \rightarrow c_l$.

This heuristic rationale is to add to the internal tour the cities that are close to the last city included in the tour and far from the border in which the internal tour shall end. There is a limited local optimization of the constructed tour. We developed a C program that allows researchers to specify continent maps and to experiment with different numbers of countries, $\mu_c$, and $\sigma_c$.

This simplified CTSP offers many advantages for production system benchmarking: the number of productions in the program can be varied by changing the number of countries; the ratio of global to local data is controlled by the average number of cities in each country; the balance in the size of local data clusters is specified by $\sigma_c$; and the specification of the continent "map" is very simple making it easy for a researcher to generate a new instantiations of the benchmark. This benchmarking facility is available through anonymous ftp to: `pine.ece.utexas.edu` in `/a/pine/home/pine/ftp/pub/parprosys`. In the measurements presented in section 6, instances of the CSTP appear as `south`, `south2`, `moun` and `moun2`. In `moun` and `south` a single set of productions performs the optimization in all country borders, while in `south2` and `moun2` an specialized set of productions is used for the optimization of each country border. Table 2 shows the relation between the number of countries in each benchmark $C$, the average number of cities in each country $\mu_c$ and important parameters in the benchmarks generated by the facility.

| Measure | `south, moun` | `south2, moun2` |
|---|---|---|
| # of productions | $20\,C + 1$ | $30\,C + 1$ |
| # WME types | $8\,C + 8$ | $15\,C + 1$ |
| # WMEs in initial database | $C\,(2\,\mu_c^2 + 2\,\mu_c + 3)$ | $C\,(2\,\mu_c^2 + 2\,\mu_c + 3)$ |

Table 2: Static measures for the CTSP benchmark according to $C$ and $\mu_c$

### 5.1.3 Confusion of Patents Problem

We constructed a solution for the formulation of the *Confusion of Patents Problem* presented in [10, 22] The problem presents five patents, five inventors, five cities, and ten constraints. Using these constraints we must decide who invented what and where. In our solution, all 125 possible combinations and 10 constraints are present in the initial database; 67 productions use the constraints to eliminate combinations that are not possible; 19 productions select the right combinations and print the solution.

Because this solution has only four different types of WMEs, most of the productions either change or test the same kinds of WME. As a consequence, productions have strong interdependency, resulting in a production system poorly suited for clustering. Even in a machine with a moderate number of processors, most of the actions need to be broadcast on the network. The main source of parallelism is the concurrent execution of different portions of the Rete network. Performance measures to this solution of the confusion of patents problem are reported under the name `patents`.

### 5.1.4 The Hotel Operation Problem

Originally written by Steve Kuo at the University of Southern California, `hotel` is a production system that models the operation of a hotel. It is a relatively large and varied production system (80 productions, 65 WME types) with 17 non-exclusive contexts. Because each production in `hotel` is related with the activities that actually take place in a hotel, the amount of speedup obtained depends on the balance of work among each one of these activities. For example, if a hotel is specified with a large number of tables in the restaurant and very few rooms, the productions that take care of the restaurant tables will have a much larger load than the productions that cleanup the rooms. This work unbalance is transferred to parallel architectures that partition the program at the production level.

### 5.1.5 The Game of Life

This is an implementation for Conway's game of life, as constructed by Anurag Acharya. After our modifications, `life` has forty productions. Twenty five of these productions are in the context that computes the value of each cell for the next generation and potentially can be fired in parallel. The other fifteen productions are used for sequencing and printing and can be only slightly accelerated by Rete network parallelism.

### 5.1.6 The Line Labeling Problem

Different versions of the line labeling problem (Waltz and Toru-Waltz) have being used for performance evaluation [27, 28, 34, 37]. Our version was originally written by Toru Ishida (Columbia Univ.), and successively modified by Dan Neiman (Univ. of Massachusetts), Anurag Acharya (Carnegie-Mellon Univ.) and José Amaral (Univ. of Texas). The current version has two non-overlapping stages of execution, each one with four productions. Because the system is partitioned at the production level, the amount of parallelism is limited to four fold. Such a low limit in speedup occurs because this is a simple "toy" problem with only ten productions, not adequate for the architecture proposed. The line labeling problem is identified as `waltz2` in our set of benchmark.

Table 3 shows static measures — number of productions, number of distinct WME types, average number of antecedents per production, average number of consequents per productions — for the benchmarks used to estimate performance in the multiple functional unit Rete network. `south` and `south2` are CTSPs with four countries and ten cities per country; `moun` and `moun2` are CTSPs with ten countries and 15 cities per country; `life`, `patents`, `waltz2`, and `hotel` are the benchmarks discussed in sections 5.1.3 to 5.1.6.

| Bench. | # Prod | Ant./prod | Cons./prod | # WME types |
|---|---|---|---|---|
| `life` | 40 | 6.1 | 1.3 | 5 |
| `hotel` | 80 | 4.1 | 2.0 | 62 |
| `patents` | 86 | 5.2 | 1.2 | 4 |
| `south` | 91 | 4.7 | 2.8 | 40 |
| `south2` | 121 | 4.7 | 2.7 | 61 |
| `moun` | 211 | 4.7 | 2.8 | 88 |
| `moun2` | 301 | 4.7 | 2.7 | 151 |
| `waltz2` | 10 | 2.7 | 8.0 | 7 |

Table 3: Static Measures for Benchmarks Used.

# 6 Performance Measurements

The benchmarks described in section 5.1 were used to evaluate the performance of the proposed architecture. First we measure the amount of speedup over an architecture with global synchronization and without overlapping between matching and selecting-acting within a processor. Then we investigate the effectiveness of the use of associative memories. Finally we obtain estimates for the size of associative memories needed for each one of the benchmarks and for the level of activity in the bus.

Notice that this section measures performance improvement obtained from two distinct ideas: section 6.1 measures the improvement solely due to elimination of over-synchronization and section 6.2 measures the improvement solely due to use of associative memories. However, because there is some interaction between these improvements, their product is only a rough estimate of the combined benefit of these ideas.

## 6.1 Parallel Firing Speedup

To measure the advantages of parallel production firing and of the internal parallelism in each processor, we define a globally synchronized architecture that is very similar to the one proposed in this paper, except that it performs global conflict set resolution to implement the OPS5 recency strategy. This synchronized architecture is also very similar to the one suggested by Gupta, Forgy, and Newell [15]. In this architecture, each processor reports the best local instantiation to be fired to the bus controller. The bus controller selects the instantiation whose time tag indicates it to be the latest one to become fireable. This added decision capability in the bus controller implements the recency strategy to solve the conflict set. The processor selected to fire a production broadcasts all changes in the bus. A processor only selects a new candidate to fire when the matching in the Rete network is complete. The bus controller waits until all processors report a new candidate to fire. This mechanism reproduces the global synchronization and conflict set generation/resolution present in many of the previously proposed architectures. In order to have a fair comparison, we considered that the synchronized architecture uses an associative memory to store and solve the local conflict sets, and that the bus controller chooses the "winner" in one time step.

Since the synchronized architecture also uses associative memory to store and search the local conflict sets, the comparisons of Figures 5 and 6 do not reflect the advantages of using such memories in our architecture. We delay this analysis until Section 6.2.

Figure 5 shows the speedup curves for the benchmarks `life`, `hotel`, `patents`, and `waltz2`. In this and the next section, we will observe a significant difference in performance and memory requirements between this group of benchmarks and the ones based on CTSP (`south`, `south2`, `moun`, and `moun2`). This is due to a gap in complexity between the two groups of benchmarks: the CTSP programs have higher data locality, larger number of productions, and larger data sets. Due to these characteristics, CTSP programs reflect more closely the characteristics encountered in production system applications in industry. The curve names starting with "`s`" indicate measures in the synchronized architecture; the curve names starting with "`a`" indicate measures in the architecture proposed in this paper. All speedups are measured against a single processor synchronized architecture. For the benchmarks presented in Figure 5, there is not much distinction between the two architectures when they have a single processor. This indicates that the parallelism between the matching phase and the selecting/execution phase does not result in much speed

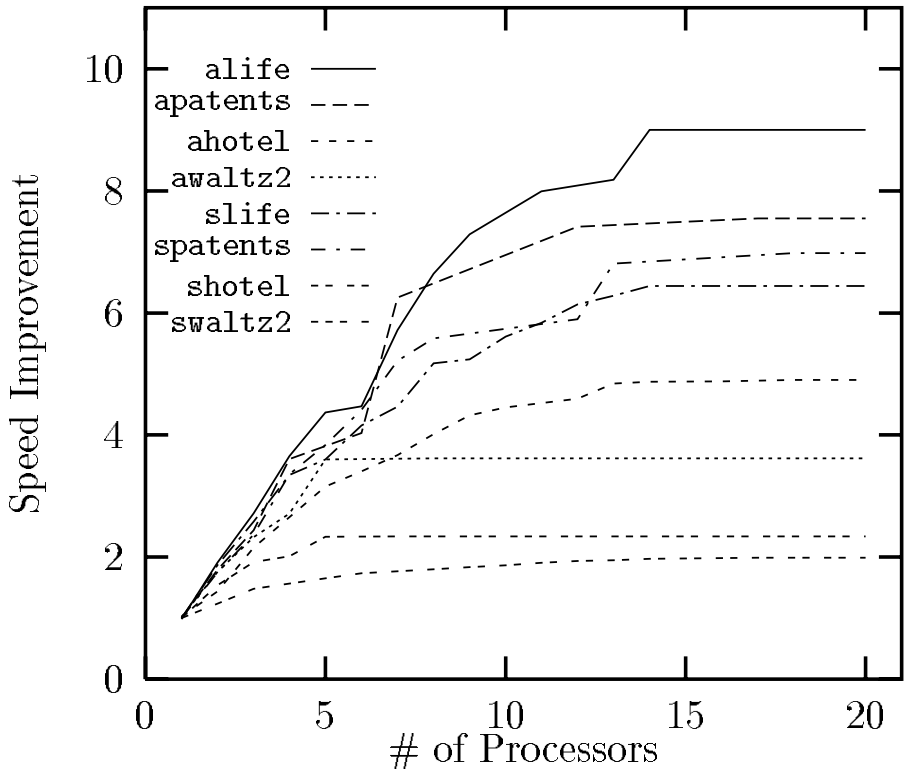


Figure 5: Speed improvement measures comparing "a" curves representing the new architecture that eliminates oversynchronization with the "s" curves of an idealized synchronous architecture that solves conflict set in one time step. Both systems use associative memories.

improvement for these benchmarks. Yet, even with these "toy problems", the parallel firing of productions and the elimination of the global synchronization provides significant speedup.

Figure 6 shows the comparative performance for the CTSP benchmarks. Here, significant speedup is observed over the synchronized architecture even for the single processor configuration. This measures the amount of speed that is gained due to the parallelism between matching and selecting/firing. The apparent superlinear speedup in the curves of Figure 6 reflects the fact that these curves are showing the combined speedup due to two different factors: intra and interprocessor parallelism. To obtain the speedup due exclusively to parallel production firing, the reader should divide the values in the "a" curves by the values in the same curve for a single processor machine. These results confirm our initial conjecture that the elimination of the global synchronization in a production system allows the construction of machines with significant speedup.

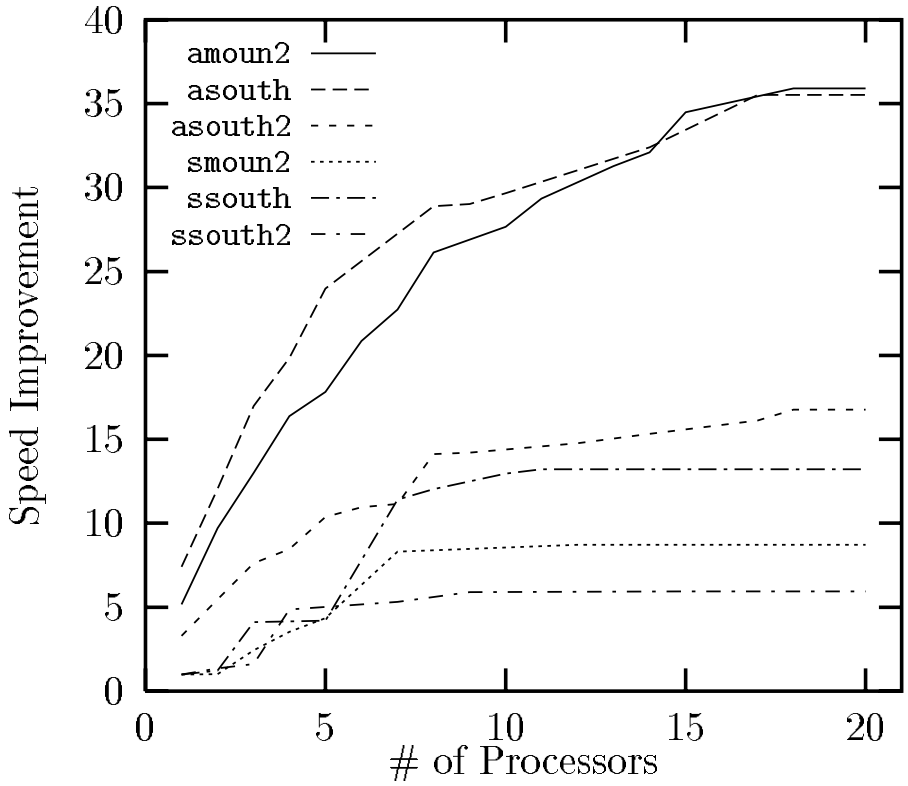


Figure 6: Speed improvement measures comparing "a" curves representing the new architecture that eliminates oversynchronization with the "s" curves of an idealized synchronous architecture that solves conflict set in one time step. Both systems use associative memories.

Another way to compare the two architectures is to measure how much speedup the proposed architecture has over the synchronized one with the same number of processors. Measurements were made for machines with one through twenty processors. Table 4 shows the mean and the variance for the speedups obtained with each configuration. It also shows the maximum and minimum speedup obtained with any number of processors. Because our architecture implements "eager" production firing without generating a global conflict set, in rare cases, some extra production execution might cause it to be slower than the synchronized architecture (see the minimum speedup for `patents`). The gap in performance between the CTSP and the other benchmarks in Table 4 indicates that the proposed architecture is very effective on extracting parallelism of PS programs that possess data locality.

| Benchmark | Speedup | | | |
|---|---|---|---|---|
| | mean | $\sigma_s$ | max | min |
| `life` | 1.14 | 0.35 | 1.28 | 1.02 |
| `hotel` | 1.89 | 1.56 | 2.50 | 1.14 |
| `patents` | 1.02 | 0.12 | 1.26 | 0.87 |
| `south` | 5.84 | 9.03 | 10.18 | 2.16 |
| `south2` | 3.53 | 3.95 | 5.79 | 1.77 |
| `moun2` | 4.90 | 5.10 | 8.07 | 2.87 |
| `waltz2` | 1.40 | 0.62 | 1.56 | 1.14 |

Table 4: Speedup over synchronized architecture using the same number of processors.

### 6.2 Effectiveness of Associative Memories

An associative memory or *content addressable memory* (CAM) is an storage device that retrieves data upon receiving a partial specification of its contents. We adopt Wade's terminology and call a traditional memory accessed by addresses a *reference addressable memory* (RAM) [43]. CAMs are most beneficial for systems in which storage devices are often searched for a cell with a given pattern. The most well known applications of the CAM mechanism are the tag matching in a cache memory and the data checking in a snooping cache or directory. When a CAM receives a request for a piece of data, it searches all positions of the memory and reports the contents of the records that match the specified pattern. Obvious advantages of a CAM over a RAM are the possibility of parallel matching when enough hardware is available to implement it, the liberation of the processor during memory searches, and reduced traffic between processor and memory [41].

In section 3 we stated that the design of the architecture is based on the premise that the use of CAMs significantly improves the processing speed. In this section we address questions that come to the mind of an inquisitive computer architect when analyzing the architecture. First, assume a machine configuration in which all memory components are CAM: what would be the impact of replacing one of these CAMs for a RAM? Second, consider a machine in which all memories are RAM: how much speedup would be gained if one of these memory components were to be replaced for a CAM?

To evaluate the speedup obtained by the use of CAMs, we implemented options in the simulator that allow us to specify whether each one of the individual memory components — AFIM, FIM, and PMM — is a CAM or a RAM. If a component is specified as a RAM, the simulator counts the number of accesses performed until the searched data item is found. This number is multiplied by the RAM access time to

find the time for that particular access. If a component is specified as a CAM, every access takes the same amount of time.

The effectiveness of a CAM in the architecture depends on the amount of data stored in the memory, the frequency of access, and whether its accesses are in the critical path of execution. Thus, the amount of speedup obtained by a given combination of CAM/RAM memories depends on the production system program that the machine is executing. For a production system program that maintains a large number of productions in the conflict set, the use of CAM for AFIM and FIM might result in a considerable speed improvement. If the conflict set is small, the use of CAM for these memories only improves the speed slightly.

To set up experiments to measure these speedups, we defined two quantities: $Speedup(M, B)$ and $Slowdown(M, B)$. $Speedup(M, B)$ is the amount of speedup that results when the memory component $M$ is replaced for a CAM in a machine that was originally formed only by RAMs. $M$ designates one of the memory components — PMM, AFIM, or FIM — and $B$ is a benchmark program. While $Speedup(M, B)$ in this section measures the amount of speed gained because of the use of CAMs, the speedup measured on section 6.1 was relating the asynchronous firing of production with a machine that fires productions synchronously but also uses CAMs. Because the base machine to compute the speedup in this section and in section 6.1 are different, these two set of measurements are not to be compared. Equation 5 shows how the speedup of PMM is measured.

$$Speedup(PMM, B) = \frac{Time(PMM_r, FIM_r, AFIM_r, B)}{Time(PMM_c, FIM_r, AFIM_r, B)}, \tag{5}$$

where $M_r$ indicates that the memory component $M$ is RAM and $M_c$ indicates that the memory component $M$ is CAM. $Time(PMM_r, FIM_r, AFIM_r, B)$ is the amount of time taken to execute the benchmark $B$ with the architecture configuration specified.

Considering a machine that uses only CAMs, $Slowdown(M, B)$ measures the reduction in speed that would occur if the memory component $M$ were to be replaced for a RAM. Equation 6 shows the measurement of the slowdown that results from the transformation of PMM from a CAM to a RAM.

$$Slowdown(PMM, B) = \frac{Time(PMM_r, FIM_c, AFIM_c, B)}{Time(PMM_c, FIM_c, AFIM_c, B)}. \tag{6}$$

For a given benchmark program the amount of speedup obtained by using CAM memories varies with the number of processors used in the architecture. Table 5 presents the average speedup for machines with one up to twenty processors. In practical designs, CAMs might be slower than RAMs: either because they

are constructed with older technology, or because they need to use more silicon area for the comparator circuits. To account for these factors we introduce a *technology factor* $T$ that indicate how much slower a basic operation such as the reading or writing of a single data element was considered in this comparison. Table 5 shows measures for a machine with CAMs with the same speed as the RAMs ($T = 1$) and for a machine with CAMs that are four times slower ($T = 4$) than the RAMs. Observe that there is no significant difference in speedup between the two measures, indicating the advantage of the use of CAMs, even if they are slower than RAMs.

| Benchmark | T | PMM | | FIM | | AFIM | | All |
|---|---|---|---|---|---|---|---|---|
| | | Speedup | Slowdown | Speedup | Slowdown | Speedup | Slowdown | Speedup |
| `hotel` | 1 | 3.03 | 29.25 | 0.99 | 1.56 | 1.56 | 13.48 | 45.51 |
| `hotel` | 4 | 3.01 | 30.11 | 0.99 | 1.57 | 1.48 | 13.56 | 45.32 |
| `life` | 1 | 2.82 | 2.12 | 1.29 | 1.02 | 1.59 | 1.16 | 3.35 |
| `life` | 4 | 2.80 | 2.14 | 1.29 | 1.02 | 1.56 | 1.16 | 3.35 |
| `moun2` | 1 | 3.25 | 4.85 | 1.00 | 1.02 | 1.79 | 2.50 | 8.50 |
| `moun2` | 4 | 3.29 | 4.94 | 1.00 | 1.02 | 1.73 | 2.54 | 8.49 |
| `patents` | 1 | 1.87 | 1.56 | 1.03 | 1.01 | 1.41 | 1.17 | 2.28 |
| `patents` | 4 | 1.87 | 1.55 | 1.03 | 1.01 | 1.40 | 1.16 | 2.26 |
| `south2` | 1 | 3.37 | 10.04 | 1.01 | 1.08 | 1.43 | 4.32 | 14.80 |
| `south2` | 4 | 3.34 | 10.17 | 1.01 | 1.11 | 1.48 | 4.41 | 14.93 |
| `waltz2` | 1 | 1.81 | 1.41 | 1.03 | 1.01 | 1.94 | 1.58 | 2.96 |
| `waltz2` | 4 | 1.81 | 1.45 | 1.03 | 1.01 | 1.87 | 1.59 | 2.95 |

Table 5: Speedup due to use of CAMs[9].

Table 5 shows the speedup and the slowdown due to each piece of associative memory for each one of the benchmarks presented in section 5.1. The last column shows the speedup that compares a configuration with all three memories associative against one in which all three memories are RAM. Table 5 shows that replacement of just one memory for a CAM results in quite low speedup. This limited speedup is result of the slow operation of the RAMs in the machine. Only when all three memories are made CAMs, the processing speed shows considerable improvement. The numbers in the slowdown columns show that the use of RAM in PMM or AFIM alone might cause significant reduction in speed. Both experiments show that the use of CAM for FIM is not very important. Overall, these results confirm our initial conjecture that the use of CAMs can provide considerable speedup in production system architectures.

[9]Each number is an average of 20 values, obtained for systems with 1 through 20 processors.

| Benchmark | PMM | | FIM | | AFIM | | FIM(synchronized) | |
|---|---|---|---|---|---|---|---|---|
| | Max | Ave | Max | Ave | Max | Ave | Max | Ave |
| `hotel` | 3200 | 1436 | 395 | 216 | 1030 | 699 | 3580 | 1178 |
| `life` | 2877 | 2643 | 690 | 584 | 3313 | 1472 | 23030 | 8787 |
| `moun2` | 27899 | 23303 | 2580 | 727 | 15634 | 3042 | 313400 | 46747 |
| `patents` | 776 | 739 | 605 | 179 | 1549 | 449 | 1410 | 426 |
| `south2` | 4788 | 2822 | 350 | 95 | 1159 | 611 | 47205 | 8414 |
| `waltz2` | 3573 | 1109 | 1250 | 870 | 2797 | 1688 | 5785 | 3299 |

Table 6: Maximum and average "crest" for memory size (bytes).

### 6.3 Associative Memory Size

The next question that the inquisitive computer architect must ask is: how large do these associative memories need to be? The simulator has an option to report the "crest"[10] of each memory component in any given run. Table 6 shows the maximum and the average crest over machines with up to twenty processors. The average crest is the average of the largest memory needed for each machine configuration. The maximum crest indicates the minimum memory size needed to run that specific benchmark. Observe that for some memory/benchmark the average crest is several times smaller than the maximum crest (see AFIM in `moun2` and PMM in `waltz2`). If memory size becomes a concern in the construction of the machine, a RAM can be used to contain overflow. The absence of a direct correlation between the size of the memory crest and the speedup and slowdown shown in table 5 reflects the fact that the processing speed is not solely dependent on the amount of data stored in each memory: it also depends on the frequency and time of access of these memories.

The speed comparison with the synchronized architecture presented in section 6.1 considered that both architectures used associative memory to store and search the conflict set. The average and the maximum crests of the associative memories for the synchronized architecture are presented in the rightmost columns of Table 6. Observe that for most of the significant benchmarks, the synchronized architecture needs a much larger memory. For the CSTPs benchmarks (`moun2` and `south2`) the maximum crest in the synchronized architecture was ten times larger than in the architecture proposed in this paper. This evidences that the "eager firing" mechanism also reduces the demand for memory.

[10]The *crest* of a memory component is the maximum amount of data stored in that memory component in any processor of the machine for a given benchmark and a specified number of processors.

| Benchmark | Bus Utilization(%) | | |
|---|---|---|---|
| | 4 proc. | 8 proc. | 16 proc. |
| `hotel` | 10.9 | 20.9 | 23.7 |
| `life` | 0.83 | 1.38 | 2.02 |
| `moun2` | 2.25 | 3.83 | 4.71 |
| `patents` | 0.68 | 0.89 | 1.08 |
| `south2` | 4.97 | 8.31 | 9.72 |
| `waltz2` | 1.36 | 1.79 | 1.76 |

Table 7: Percentage of time that the bus is busy.

### 6.4 Use of Bus

A legitimate concern about any bus-based parallel architecture is the limitation of a bus as a broadcasting network. In sections 2 and 3 we conjectured that bus bandwidth is not a limitation in the architecture proposed. Table 7 presents the measurements for the percentage of time that the bus is busy for machines with 4, 8 and 16 processors, assuming that bus bandwidth is the same as that of local memory. These measures include the arbitration time and the token broadcasting time. Observe that technological limitations would have to render the bus much slower than the memories before the bus speed becomes a concern in this architecture.

## 7 Concluding Remarks

We proposed a new architecture for production systems that eliminates global synchronization and the generation of a global conflict set. The increased importance of associative search for maintaining fireable instantiation tables in this setting is underscored by the big performance gains obtained by using modest amounts of associative memory. Note that a single physical CAM can be logically partitioned into PMM, FIM and AFIM, and the "crests" in each partition are not expected to occur in the same processor and at the same time. Thus, only a few kilobytes of associative memory is sufficient for most of the benchmarks considered.

A number of issues remain for future research in this area. With the improved speed in production selection and firing due to the CAMs, the matching in the Rete network is again a bottleneck. We have developed an analytical model to investigate the utilization of multiple functional units in the Rete network of each processor. The predictions indicate that a small number of functional units provide significant

improvement in the Rete network speed [7]. One can now study the system-level effect of a faster Rete network for the architecture proposed in this paper.

Acharya and Tambe have showed the usefulness of handling collections of WMEs instead of single WMEs during the match phase [1]. The manipulation of collections in the architecture presented in this paper would further reduce the amount of traffic in the bus. However, more theoretical studies are necessary before collection oriented production systems are built. For example, the handling of self-disabling productions in collection oriented systems needs to be studied with care.

This research assumed the use of serializability as a correctness criterion. Our experience with PS benchmarks indicates that programmers often rely on knowledge about conflict set resolution strategies when writing PS programs. This is mostly evidenced by the omission of important antecedents in productions that are enabled but never selected to fire by an specific strategy. For problems like CTSP, writing a serializable correct PS was fairly straightforward. Now that our study has indicated that serializable systems offer great speed improvements, it is desirable to develop programming aid tools to help in the specification and verification of a wider range of serializable PS programs.

## 8 Acknowledgements

We are very thankful to Anurag Acharya for letting us use the front-end of his parallel compiler [2], for being so helpful with many questions, and for providing some of the benchmarks that we used. We would also like to acknowledge Howard Owens for tracking a difficult bug in the implementation of the simulator, and Dan Miranker for fruitful discussions.

## References

[1] A. Acharya and M. Tambe. Collection-oriented match: Scaling up the data in production systems. Tech. Rep. CMU-CS-92-218, Carnegie-Mellon University, Pittsburgh, December 1992.

[2] A. Acharya, M. Tambe, and A. Gupta. Implementation of production systems on message-passing computers. In *IEEE Trans. on Parallel and Distributed Systems*, volume 3, pages 477–487, July 1992.

[3] Anurag Acharya. Design of PPL: A parallel production language. School of Computer Science, Carnegie Mellon University, Preliminary draft., 1993.

[4] J. N. Amaral. *A Parallel Architecture for Serializable Production Systems.* PhD thesis, The University of Texas at Austin, Austin, TX, December 1994. Electrical and Computer Engineering.

[5] J. N. Amaral and J. Ghosh. An associative memory architecture for concurrent production systems. In *Proc. 1994 IEEE International Conference on Systems, Man and Cybernetics*, pages 2219–2224, San Antonio, TX, October 1994.

[6] J. N. Amaral and J. Ghosh. Speeding up production systems: From concurrent matching to parallel rule firing. In L. N. Kanal, V. Kumar, H. Kitani, and C. Suttner, editors, *Parallel Processing for AI*, chapter 7, pages 139–160. Elsevier Science Publishers B.V., 1994.

[7] J. N. Amaral and J. Ghosh. Using queueing theory for analytical performance evaluation of a multiple functional unit rete network. In *XV Congress of the Brazilian Computer Society*, pages 611–625, July 1995.

[8] J. N. Amaral and J. Ghosh. Versatile benchmarking for concurrent production system architectures. In *XV Congress of the Brazilian Computer Society*, pages 599–610, July 1995.

[9] F. Barachini and N. Theuretzbacher. The challenge of real-time process control for production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 705–709, August 1988.

[10] R. E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1(1):27–120, 1970.

[11] C. L. Forgy. *On the Efficient Implementations of Production Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1979.

[12] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theor. Comput. Sci.*, 1:237–267, 1976.

[13] J.-L. Gaudiot and A. Sohn. Data-driven parallel production systems. *IEEE Transactions on Software Engineering*, 16:281–291, March 1990.

[14] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, March 1986.

[15] A. Gupta, C. Forgy, and A. Newell. High-speed implementations of rule-based systems. *ACM Transactions on Computer Systems*, 7:119–146, May 1989.

[16] A. Gupta, M. Tambe, D. Kalp, C. L. Forgy, and A. Newell. Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programing*, 17, 1988.

[17] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of ACM*, 27(4):797–821, October 1980.

[18] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability and Programmability.* McGraw-Hill, New York, 1993.

[19] T. Ishida. Optimizing rules in production system programs. In *Proceedings of National Conference on Artificial Intelligence*, pages 699–704, August 1988.

[20] T. Ishida and S. Stolfo. Towards the parallel execution of rules in production system programs. In *Proceedings of International Conference on Parallel Processing*, pages 568–575, 1985.

[21] T. Ishida, M. Yokoo, and L. Gasser. An organizational approach to adaptive production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 52–58, July 1990.

[22] P. C. Jackson. *Introduction to Artificial Intelligence.* Dover Pub., New York, 1985.

[23] K. Kant. *Introduction to Computer System Performance Evaluation.* McGraw-Hill, New York, 1993.

[24] M. A. Kelly and R. E. Seviora. An evaluation of DRete on CUPID for OPS5 matching. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 84–90, August 1989.

[25] M. A. Kelly and R. E. Seviora. A multiprocessor architecture for production system matching. In *Proceedings of National Conference on Artificial Intelligence*, pages 36–41, July 1989.

[26] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergammon Press, 1970.

[27] C.-M. Kuo, D. P. Miranker, and J. C. Browne. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13:424–441, December 1991.

[28] S. Kuo and D. Moldovan. Performance comparison of models for multiple rule firing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 42–47, August 1991.

[29] S. Kuo and D. Moldovan. The state of the art in parallel production systems. *Journal of Parallel and Distributed Computing*, 15:1–26, June 1992.

[30] H. S. Lee and M. I. Schor. Match algorithms for generalized Rete Networks. *Artificial Intelligence*, 54:249–274, April 1992.

[31] D. P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Pittman/Morgan-Kaufman, 1990.

[32] D. I. Moldovan. Rubic: A multiprocessor for rule-based systems. *IEEE Transactions on Systems, Man and Cybernetics*, 19:699–706, July/August 1989.

[33] P. Nayak, A. Gupta, and P. Rosenbloom. Comparison of the Rete and Treat production matchers for SOAR (a summary). In *Proceedings of National Conference on Artificial Intelligence*, pages 693–698, August 1988.

[34] D. E. Neiman. *Design and Control of Parallel Rule-Firing Production Systems*. PhD thesis, University of Massachusets, Amherst, MA, September 1992.

[35] K. Oflazer. Partitioning in parallel processing of production systems. In *Proceedings of International Conference on Parallel Processing*, pages 92–100, 1984.

[36] J. G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13:348–365, December 1991.

[37] J. G. Schmolze and S. Goel. A parallel asynchronous distributed production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 65–71, July 1990.

[38] J. G. Schmolze and W. Snyder. Using confluence to control parallel production systems. In *Second International Workshop on Parallel Processing for Artificial Intelligence (PPAI-93)*, August 1993.

[39] A. Sohn and J.-L. Gaudiot. A macro actor/token implementation of production systems on a data-flow multiprocessor. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 36–41, August 1991.

[40] S. Stolfo, H. Dewan, and O. Wolfson. The PARULEL parallel rule language. In *Proc. 1991 International Conference on Parallel Processing*, pages 36–45, 1991.

[41] A. Asthana Stolfo *et al.* A high bandwidth intelligent memory for supercomputers. In *Proceedings of Supercomputing Conference*, pages 517–524, May 1988.

[42] C. P. Thacker, D. G. Conroy, and L. C. Stewart. The alpha demonstration unit: A high-performance multiprocessor. *Communications of the ACM*, 36:55–67, February 1993.

[43] J. P. Wade. *An integrated content addressable memory system*. PhD thesis, Massachusetts Institute of Technology, May 1988.

[44] J. Xu and K. Hwang. Mapping rule-based systems onto multicomputers using simulated annealing. *Journal of Parallel and Distributed Computing*, 13:442–455, December 1991.