Serializability Improves Parallel Execution of Production System

José Nelson Amaral*
(*amaral@madona.pucrs.br*)
Departamento de Eletrônica
Pontifícia Universidade Católica do RGS
90619-900 - Porto Alegre, RS

Joydeep Ghosh
(*ghosh@pine.ece.utexas.edu*)
Dept. of Electr. and Comp. Engineering
The University of Texas at Austin
Austin, Texas 78712

Abstract

This paper presents a new production system architecture that uses serializability as a correctness criterion to select a set of productions to be executed in parallel. The use of serializability eliminates global synchronization. This architecture takes advantage of modern associative memory devices to allow parallel production firing, concurrent matching, and overlap among matching, selection, and firing of productions. A comprehensive event-driven simulator is used to evaluate the scaling properties of the new architecture and to compare it with a parallel architecture using global synchronization before every production firing. Our results indicate that the combination of serializability and associative memories can achieve substantial improvements in speed with a very modest increase in hardware cost.

1 Introduction

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 [4]. 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 [6], substantial efforts were made to improve this phase. Comprehensive surveys of the research towards speeding up production systems are found in the works of Kuo and Moldovan [10] and Amaral and Ghosh [3].

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 [16].

*Supported by a fellowship from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and by Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) - Brazil.

The commutativity criterion proposed by Ishida and Stolfo [8] 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 [17] studied the use of confluence to control a parallel production system. They suggest the use of term rewriting systems 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.

On surveying measurements published by other authors [13, 6], 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 [1].

The need to improve other phases of production execution besides the match cycle is now evident [3]. 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. This architecture follows an early recommendation of Gupta and Forgy [7], i.e., that a parallel production system machine be constructed with a small number of relatively powerful processors.

2 Architectural Model

The architectural model proposed in this paper consists of a moderate number of identical processors interconnected through a Broadcasting Interconnection Network (BIN). Each of the processors has the internal organization shown in Figure 1. An I/O processor attached to the BIN initially loads the productions and the initial database in the processors. At compile time each production is uniquely assigned to a processor according to a partitioning algorithm that takes into consideration inter-production dependencies and workload balance [2]. 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 production system 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 the local processor are broadcast over the network.

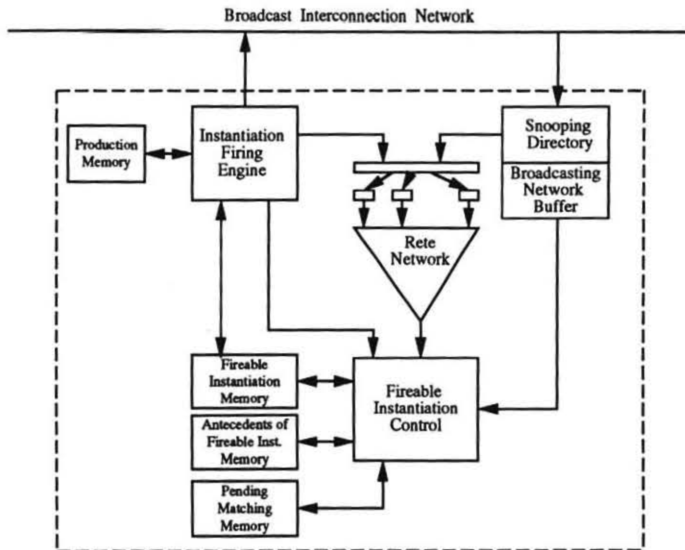

Figure 1: Processing Element Model

All tokens propagated over the BIN consist of deletion, addition or modification of a WME. Such operations might enable or disable a local production. Upon processing a token, the Fireable Instantiation Control (FIC) has to do the following: perform an associative search in the Antecedents of Fireable Instantiation Memory (AFIM) to verify which previously enabled productions are now disabled; remove such productions from the Fireable Instantiation Memory (FIM), and remove all their antecedents from AFIM; and place the incoming token in the input queue of the Rete Network. Notice that because the productions that are no longer fireable were removed from FIM, a *partially informed* selection can proceed and select a new production to be fired among the ones that remained in FIM.

This capability to fire a new production before the changes generated in the previous production firing are fully propagated through the Rete Network results in low overhead for token removing[1], and allows the maintenance of the beta memories of the original Rete algorithm [4]. This combination of the advantages of Rete and Treat is made possible by the storage of negated conditions in the representation of fireable instantiations of productions stored in FIM.

The compiler classifies the productions as local or remote: a local production modifies only WMEs that are exclusively stored in local memories; a remote production changes pieces of memory that are stored in other processors. Local productions are further classified in Independent of Network Transactions (INT) or Dependent on Network Transactions (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.

To select a production to fire, the Instantiation Firing Engine (IFE) performs an associative search in FIM to find the most recently enabled production. If the selected production is remote, the IFE places a request for ownership of the BIN. Upon receiving BIN ownership, the IFE waits until all outstanding tokens from previous broadcastings are processed by FIC. The IFE access FIM to verify whether the selected production is still fireable. If it is, IFE proceeds to execute its actions, propagating tokens that change shared WMEs in BIN and sending tokens that modify only local WMEs to FIC and Rete.

The Snooping Directory (SD) is an associative memory that contains a list of all WME types that are tested by antecedents of the productions assigned to the local processor. SD "snoops" BIN and captures only tokens that modify WMEs relevant to the processor. If there is a local production being executed, the token cannot be immediately processed. It is stored in the Broadcasting Network Buffer (BNB), and is processed as soon as the local production processing finishes.

The Pending Matching Memory (PMM) is necessary to store tokens that are in the Rete Network. Whenever a change to the conflict set[2] is generated in the Rete Network, FIC performs an associative search in PMM to verify if a later modification invalidates such change. This mechanism prevents races between IFE and Rete.

The main steps in the machine operation are presented below in an algorithmic form. The

[1] Low overhead in token removing is the most salient advantage of the Treat algorithm [11].

[2] "Conflict set" is the set of all productions enabled to be fired at any given time.

steps of the algorithm are performed by different structures of the processing element. A formal proof that the architectural model presented in this section produces correct results according to the serializability criterion is given in [1].

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 captured from BIN
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** wait BIN ownership
14.   **execute** tokens captured from BIN
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[3]. This is not a true recency criterion because the Rete Network 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. If such productions were to start firing while a remote processor is in the middle of a production execution, the intermin-

[3]A single associative search in FIM produces the entry with the most recent time stamp.

gling 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 16 is activated immediately upon releasing the BIN.

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 [18]. During startup each processor is assigned an arbitrary priority number from 0 to $N-1$. $N-1$ 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.

2.1 Correctness of the Processing Model

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 that together characterize its* **type**, $T[W_k]$[4].

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 the antecedents of a production R_i test WMEs of type $T[W_k]$, then W_k* **belongs** *to the antecedents of R_i, it is notated by $W_k \in A(R_i)$*[5].

[4]Two WMEs of the same type are distinguished only by the values associated with their attributes.

[5]A WME might belong to the antecedents of more than one production.

In the architectural model described in Section 2, 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 WMEs tested by its productions. This is stated in Axiom 1.

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

Definition 4 (Serializability Criterion of Correctness) *The parallel execution of a collection of production instantiations $\mathcal{I}$ is correct iff there exists at least one serial execution of $\mathcal{I}$ that produces the same results as the parallel execution.*

Theorem 1 states that the operation of architecture proposed in section 2 is correct according to the serializability criterion [16]. The proof for theorem 1 was produced by Amaral based on an analysis of all possible conflicting condition that might appear in the architecture [1]. This proof is not presented in this article due to the limitation of space.

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.*

In the following section we present a partition algorithm that determines which partitions shall be assigned to which processors based on the set of dependencies among the productions. This algorithm also takes into consideration the work balance among the processors. In section 4 we describe a set of benchmarks that is used for performance evaluation in sections 4.1 and 4.2.

3 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 [15]. Oflazer formulates partitioning as a minimization problem and concludes that the best suited architecture for Production Systems has a small number of powerful processors [14]. 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" [12]. Xu and Hwang use a similar scheme with matrices of cost to construct a simulated annealing optimization of the production partition problem [19].

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 [14, 12, 19]. 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 more 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 [1, 2]. The optimal partitioning of productions into disjoint sets is modeled as a minimum cut problem, which is NP-Complete [5]. 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 2 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, δ_{ij} is 1 if $i = j$ and 0 otherwise, and

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

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

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 β-node activities. Thus, the number of β-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.

$$B(P_i) = \sum_j \beta(R_j)\, \varphi_{ij}, \tag{2}$$

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

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 $F(R_i, S_k)$.

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

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

The 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.

[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.

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
$F(R_i, S_k) = \max_i F(R_i, S_k)\}$
3 $S \leftarrow S - \{R_i\}$

4 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. We used three benchmarks obtained from other researchers and developed a new benchmark in which the number of productions and the database size can be independently changed to allow researchers to study various aspects of new architectures. This new benchmarking, called Contemporaneous Traveling Salesperson Problem (CTSP), is presented in detail in [1].

Table 1 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 fifteen cities per country.

Bench.	# Prod.	Ant.	Cons.	WMEs
`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 1: Static Measures for Benchmarks Used.

`patents` is our solution to the constraint *Confusion of Patents Problem* presented in [9]. 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. The main source of parallelism is the concurrent execution of different portions of the Rete network. 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. `life` 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. Our version of the line labeling problem, `waltz2`, 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). It has two non-overlapping stages of execution, each one with four productions.

4.1 Parallel Firing Speedup

To measure the advantages of parallel production firing and The benchmarks described in section 4 were used to evaluate the performance of the proposed architecture. of the internal parallelism within 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 [7]. 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 2 and 3 do not reflect the advantages of using such memories in our architecture.

Figure 2 shows the speedup curves for the benchmarks `life`, `hotel`, `patents`, and `waltz2`.

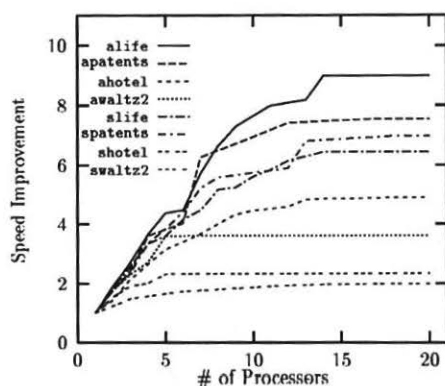

Figure 2: Speedup Curves

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. Because of 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 2, 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 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 3 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 2 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

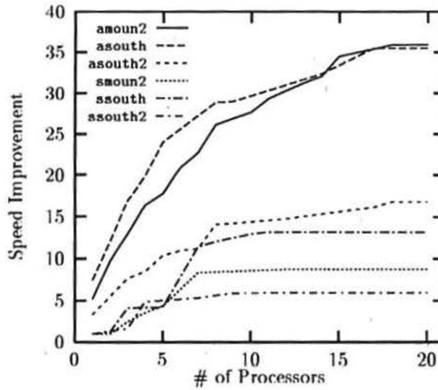

Figure 3: Speedup Curves

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.

4.2 Bus Utilization

A legitimate concern about any bus-based parallel architecture is the limitation of a bus as a broadcasting network. In section 2 we conjectured that bus bandwidth is not a limitation in the architecture proposed. Table 2 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 becames a concern in this architecture.

5 Concluding Remarks

We proposed a new production system architecture that eliminates global synchronization and the generation of a global conflict set. This elimination is possible because of the use of serializability as a correctness criterion in the selection of actions to be performed in parallel. The increased importance of associative search for maintaining fireable instantiation tables in this setting is evidenced by the big performance gains obtained by using modest amounts of associative memory.

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 2: Percentage of time that the bus is busy.

One of the drawbacks is the use of serializability as a correctness criterion is that the programmer must make sure that any possible sequential execution of enabled productions is correct. Our experience with PS benchmarks indicates that programmers often rely on knowledge about conflict set resolution strategies when writing PS programs. This often manifests itself in the omission of important antecedents in productions that are enabled but never selected to fire by a specific strategy. Now that our study has indicated that serializable systems offer great speed improvements, it is desirable to develop programming tools to help in the specification and verification of a wider range of serializable PS programs.

References

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

[2] 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*, San Antonio, TX, October 1994.

[3] 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.

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

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

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

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

[8] 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.

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

[10] 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.

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

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

[13] 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.

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

[15] J. Schmolze. A parallel asynchronous distributed production system. In *Proceedings of National Conference on Artificial Intelligence*, pages 65–71, 1990.

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

[17] 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.

[18] 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.

[19] 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.