Methods and Effectiveness of Parallel Rule Firing

Toru Ishida
NTT Communications and Information Processing Laboratories
1-2356, Take, Yokosuka-shi, 238-03, Japan
ishida@nttkb.ntt.jp@relay.cs.net

Abstract
This paper provides implementation methods for parallel rule firing, where multiple production rules are fired simultaneously in each production cycle. The methods include the interference analysis that detects cases where a parallel firing result is different from those of sequential firings of the same rules in any order, the parallel firing algorithm that realizes the parallel firing on multiple processor architectures, and the parallel programming environment that provides language facilities and a parallel firing simulator. We have evaluated the effectiveness of parallel rule firing on several production system applications. Results show that the degree of concurrency can be increased by a factor of 2 to 9. The sources of parallelism are investigated based on the evaluation results.

1 Introduction
To improve the efficiency of production systems, several multiple processor architectures have been investigated [Stolfo et al., 1982; Forgy et al., 1984; Moldvan, 1986]. Two kinds of parallel algorithms were developed to more effectively utilize the parallel processing hardware.

One is parallel rule matching which aims to speed-up the matching process that consumes up to 90% of the total execution time. Decreasing the time to match rules makes it possible to compress each cycle of production system execution. Gupta et al. [1986] have parallelized the RETE match algorithm [Forgy, 1982], and have reported that the average concurrency in actual expert systems was improved 15.92-fold and execution speed was increased 8.25-fold. Miranker [1987] has also proposed the TREAT match algorithm, which was designed for fine-grain parallel processor systems.

The other type of parallel algorithm is parallel rule firing, which aims to reduce the total number of sequential production cycles by executing multiple matching rules simultaneously on a multiple processor system. The SOAR production system language [Laird et al., 1983] takes the parallel firing approach. In SOAR, each decision cycle consists of an elaboration phase and a decision procedure. The elaboration phase fires rules in parallel, and creates objects and preferences. Then the decision procedure fires rules sequentially examining the accumulated preferences and replacing objects. Gupta et al. [1986] have reported that SOAR's parallel firing mechanism increases performance of production systems when combined with parallel rule matching.

Two kinds of parallel algorithms were developed to more effectively utilize the parallel processing hardware. The first is parallel rule matching which aims to speed-up the matching process that consumes up to 90% of the total execution time. Decreasing the time to match rules makes it possible to compress each cycle of production system execution. Gupta et al. [1986] have parallelized the RETE match algorithm [Forgy, 1982], and have reported that the average concurrency in actual expert systems was improved 15.92-fold and execution speed was increased 8.25-fold. Miranker [1987] has also proposed the TREAT match algorithm, which was designed for fine-grain parallel processor systems.

The other type of parallel algorithm is parallel rule firing, which aims to reduce the total number of sequential production cycles by executing multiple matching rules simultaneously on a multiple processor system. The SOAR production system language [Laird et al., 1983] takes the parallel firing approach. In SOAR, each decision cycle consists of an elaboration phase and a decision procedure. The elaboration phase fires rules in parallel, and creates objects and preferences. Then the decision procedure fires rules sequentially examining the accumulated preferences and replacing objects. Gupta et al. [1986] have reported that SOAR's parallel firing mechanism increases performance of production systems when combined with parallel rule matching.

1 Introduction
To improve the efficiency of production systems, several multiple processor architectures have been investigated [Stolfo et al., 1982; Forgy et al., 1984; Moldvan, 1986]. Two kinds of parallel algorithms were developed to more effectively utilize the parallel processing hardware.

One is parallel rule matching which aims to speed-up the matching process that consumes up to 90% of the total execution time. Decreasing the time to match rules makes it possible to compress each cycle of production system execution. Gupta et al. [1986] have parallelized the RETE match algorithm [Forgy, 1982], and have reported that the average concurrency in actual expert systems was improved 15.92-fold and execution speed was increased 8.25-fold. Miranker [1987] has also proposed the TREAT match algorithm, which was designed for fine-grain parallel processor systems.

The other type of parallel algorithm is parallel rule firing, which aims to reduce the total number of sequential production cycles by executing multiple matching rules simultaneously on a multiple processor system. The SOAR production system language [Laird et al., 1983] takes the parallel firing approach. In SOAR, each decision cycle consists of an elaboration phase and a decision procedure. The elaboration phase fires rules in parallel, and creates objects and preferences. Then the decision procedure fires rules sequentially examining the accumulated preferences and replacing objects. Gupta et al. [1986] have reported that SOAR's parallel firing mechanism increases performance of production systems when combined with parallel rule matching.
database of assertions, called the working memory (WM). Assertions in the WM are called working memory elements (WMEs). Each rule consists of a conjunction of condition elements, called the left-hand side (LHS) of the rule, along with a set of actions called the right-hand side (RHS).

The RHS specifies information which is to be added to or deleted from the WM when the LHS is successfully matched with the contents of the WM. There are two kinds of condition elements: positive condition elements that are satisfied when there exists a matching WME, and negative condition elements that are satisfied when no matching WME is found. Pattern variables are consistently bound throughout the positive condition elements. Rules written in the OPS5 production system language are shown in Figure 1.

2.2 Execution Model of Parallel Firing

The production system interpreter repeatedly executes the Match-Select-Act cycle. In our parallel firing model, we do not assume that only one rule is chosen in the Select phase. Rather, we propose to fire multiple rules simultaneously on multiple processors. Production system programs are thus required to be written without any assumptions of particular conflict resolution strategies.

When firing multiple production rules in parallel, however, there exists the case that the result of parallel execution of rules is different from the results of sequential executions in any order of applying those rules. In this case, we say that there exists interference among multiple instantiation of rules. To avoid such an error, we detect interference in the Select phase. In the parallel rule firing model, a production cycle is executed as follows.

Match: For each rule, determine whether the LHS matches the current environment of the WM.

Select: Choose as many instantiations as possible as long as interference does not occur among selected instantiations.

Act: Fire rules according to the selected instantiations simultaneously.

2.3 A Data Dependency Graph

To analyze the interference among multiple instantiations of production rules, we introduce a data dependency graph of production systems, which is constructed from the following primitives.

A production node (a P-node), which represents a set of instantiations (P-nodes are shown as circles in all figures).

A working memory node (a W-node), which represents a set of working memory elements (W-nodes are shown as squares in all figures).

A directed edge from a P-node to a W-node, which represents the fact that a P-node modifies a W-node. More precisely, the edge indicates that a WME in a W-node is modified (added or deleted) by the corresponding rule of an instantiation in a P-node. When a rule adds (deletes) WMEs to (from) a W-node, the W-node is said to be ‘+’changed (‘-’changed), and the corresponding edge is labeled ‘+’ (‘-’).

A directed edge from a W-node to a P-node, which represents the fact that a P-node refers to a W-node. More precisely, the edge indicates that a WME in a W-node is referenced by the corresponding rule of the instantiation in a P-node. When a WME is referenced by a positive (negative) condition element of a rule, the W-node is said to be ‘+’referenced (‘-’referenced), and the corresponding edge is labeled ‘+’ (‘-’).

The interference analysis, presented in the next section, is effective for any size of P- or W-node. However, the larger the size is, the less information can be obtained. Since parallel rule firings are performed in a conservative manner, it would be better to associate P- and W-nodes with sets that are as small as possible.

3 Interference Analysis

This section details the methods of analyzing the interference among rule firings using a data dependency graph.

3.1 Building a Data Dependency Graph

First of all, we will describe the compile-time and the runtime analysis using the rules in Figure 1.

Compile-Time Analysis

Figure 2 displays the data dependency graph constructed at compile-time. A P-node is associated with a set of instantiations of each rule. In Figure 2(a), a W-node is associated with a set of WMEs in each class. In Figure 2(b), by contrast, a W-node is associated with a set of WMEs represented by a matching pattern which appears in the source production system programs. The latter is a more general approach than the former: it can accommodate cases in which class names are represented by variables. Furthermore, by utilizing all information written in the source programs, more inherent parallelism in the program is extracted.
However, in the latter approach, a way of analyzing overlapping W-nodes is necessary. Edges should be drawn from any P-node to all W-nodes which are overlapped by an RHS pattern associated with the P-node. For example, in Figure 2(b), since W-nodes, (candidate-city "name <x> "state New-York") and (candidate-city "name <y>"), overlap, '-'labeled edges are drawn from the P-node, make-possible-trip, to both W-nodes.

Run-Time Analysis

The run-time analysis can produce more accurate data dependency graphs than those of the compile-time analysis. Suppose there exist the following three WMEs.

(candidate-city "name Buffalo "state New-York")
(candidate-city "name New-York "state New-York")
(symptom "animal frog "action croak "place Buffalo"

A data dependency graph of the three created instantiations is shown in Figure 3. In this case, a P-node represents one instantiation, and a W-node represents an instantiated matching pattern. Since all pattern variables have already been instantiated, each W-node basically indicates a unique WME.

3.2 Detecting Interference

We will formally describe the paired-rule and the all-rule conditions to detect the interference among rule firings.

Paired-Rule Conditions

The paired-rule conditions [Ishida et al., 1985] detects the interference between instantiations in paired P-nodes. There is a possibility of interference between the two instantiations in P-node PA and P-node PB, if there exists a W-node which satisfies any of the following conditions:

A1) '+'changed ('-'changed) by PA and '-'referenced ('+'referenced) by PB.
A2) '+'changed ('-'changed) by PB and '-'referenced ('+'referenced) by PA.
A3) '+'changed ('-'changed) by PA and '-'changed ('+'changed) by PB.3

All-Rule Conditions

If all P-nodes could be analyzed at once, however, we could obtain more accurate results by using the all-rule conditions. An example of how this might be done is shown in Figure 4(a). Here, all rules can be fired in parallel even though interference is detected by the paired-rule conditions. For instance, condition A1 is satisfied between

3In OPS5, condition A3 is not necessary.
A rule (class1) removes class2. Run-time analysis (Fig. 3)

A rule (class2) removes class3.

A rule (class3) removes class4.

A rule (class1) removes class2.

A rule (class2) removes class3.

A rule (class3) removes class4.

Figure 4: Interference among all instantiations

ruleA and ruleB, and between ruleB and ruleC. However, if there is no other rule, ruleA, ruleB, and ruleC can be fired in parallel, because in this case the result of parallel firing is the same as the sequential firing in the order ruleC — ruleB — ruleA. On the other hand, Figure 4(b) shows an example where parallel rule firing should not be employed. In this case, since ruleA, ruleB, and ruleC interfere with each other in a cyclic fashion, the result of parallel firing could differ from any result of the sequential firing of the same rules in any order.

More formally, the all-rule conditions are described as follows: Let \( P_1, P_2, P_3, \ldots, P_n \), where \( P_1 = P_n \), be a cyclic sequence of an arbitrary number of P-nodes. We say there is cyclic interference in P-nodes, if, for all i, there exists a W-node which is '+' changed ('-' changed) by \( P_i \) and '-' referenced ('+' referenced) by \( P_{i+1} \). Interference occurs between two instantiations in P-nodes \( P_A \) and \( P_B \), if any of the following conditions is satisfied:

1. There exists cyclic interference in P-nodes including \( P_A \) and \( P_B \).
2. There exists a W-node, which is '+' changed ('-' changed) by \( P_A \) and '-' changed ('+' changed) by \( P_B \).

In OPS5, condition B2 is not necessary.

Table 1: Results of the Interference Analysis

3.3 Accuracy of the Interference Analysis

Table 1 summarizes the results of various interference analysis on the rules shown in Figure 1. The variations are derived both from the preciseness of data dependency graphs (compile-time / run-time) and from the scope of interference detection (paired-rule / all-rule). The following points can be drawn from Table 1.

- Run-time analysis can permit more parallelism than compile-time analysis due to the small size of the P- and W-nodes. For example, run-time analysis concludes that the instantiations of make-possible-trip can be fired in parallel.
- The all-rule conditions can permit more parallelism than the paired-rule conditions. For example, the all-rule conditions conclude that the instantiations of make-possible-trip and make-weather-forecast can be fired in parallel.

Ohsianwo et al. [1987] have also pointed out that not all the effects of parallelism can be determined by compile time techniques: excessive interference is indicated by these techniques which are inherently conservative. On the other hand, overheads cannot be ignored in any run-time analysis. In the next section, we combine compile- and run-time analysis techniques to get a reasonably accurate and efficient selection algorithm.

4 Parallel Firing Algorithm

From the discussions in Section 3.3, run-time analysis using the all-rule conditions seems to be the best solution. However, the computational cost of checking the all-rule conditions is quite high, because condition B1 requires that all strongly connected regions be detected from a data dependency graph. Since this overhead cannot be ignored in the run-time analysis, we propose the efficient selection algorithm shown in Figure 5 by simplifying the all-rule conditions.
There are two advantages. First, the compile-time analysis can help the selection algorithm to reduce run-time efforts: if two rules were determined not to cause interference at compile-time, we can avoid a precise check at run-time. Second, the compile-time analysis can help the selection algorithm to perform more accurate interference detection: in the compile-time analysis, though a data dependence graph is comparatively rough, we can completely examine the all-rule conditions.

5 Parallel Programming Environment

5.1 Language Facilities

To fire multiple rules in parallel, it is essential that rules are written without assuming a particular conflict resolution strategy. However, a conflict resolution strategy sometimes enables programmers to simplify rules. For example, by assuming the MEA or LEX strategy [Forgy, 1981], programmers can simplify the LHS of the rule which is to be fired only when other rules cannot be fired. It is thus impractical to disable conflict resolution strategies. Rather, we take an alternative approach by introducing the following language facilities:

- A ruleset is introduced to form a group of rules. Distinct conflict resolution strategies can be defined independently for each ruleset.
- A new conflict resolution strategy, DON'T-CARE, is introduced along with MEA and LEX. Rules in a ruleset under the strategy DON'T-CARE are fired in parallel.

From our experience, the rules written for man-machine interfaces, cannot be fired in parallel. To cover this, we divide rules into multiple rulesets: rulesets for man-machine interfaces are executed under the strategy MEA, and other rulesets under the strategy DON'T-CARE.\(^5\) Perhaps the most important thing to note is that by introducing the above language facilities, programmers are only required to select an appropriate strategy for each ruleset, and do not have to consider the internal parallel mechanism.

However, a more complex case also exists, especially when programs are originally written for sequential execution, in which the two types of rules are mixed: control rules can be executed under the DON'T-CARE strategy. The following mechanism has been devised to accommodate cases of this kind.

- A focusing mechanism is introduced to transfer control from one ruleset to another. For example, suppose the rules are divided into two rulesets: CONTROL, for control rules, and HEURISTICS for heuristic rules. Then, the focus function enables the production system interpreter to perform the mixed execution of both rulesets.

\(^5\)Note that we can write positive/negative conditions and add/delete actions under the DON'T-CARE strategy.
Since higher priority is given to the former ruleset, the control rules to shift stages are fired only when the heuristic rules can no longer be fired.

5.2 Simulation Environment

To evaluate the effectiveness of parallel firing of production systems, a simulation environment has been developed. The environment consists of an Analyzer and a Simulator.

The Analyzer inputs rules written in an OPS5-like production system. It constructs a data dependency graph of given production rules, analyzes the graph and outputs the results which show what rules can be fired in parallel. The compile-time interference analysis, which is described in Section 3, is performed in the Analyzer.

The Simulator simulates the parallel firing of production systems using the Selection algorithm described in Section 4. The results of the compile-time analysis performed by the Analyzer are referenced during the simulation. The overhead of the run-time selection is less than 10% of the total execution time.

6 Evaluation Results

Parallel Executability of Production Systems

We have evaluated several production system applications, all of which permitted some degree of parallelism. Table 2 summarizes the results of the evaluation. The evaluation was made under the assumption that processes are dynamically allocated to an effectively infinite number of PEs. Concerning parallel rule firing, the followings can be observed from Table 2:

- The average number of firings per cycle is 5.11 to 7.57.
- WM changes per cycle is 5.92 to 24.57, which is approximately 2 to 9 times more than in the sequential firings.

Stolfo [1984] proposed various parallel algorithms for production systems. If the algorithm performs the entire pattern matching repeatedly for each production cycle, we can expect speed to be improved by a factor of 5.11 to 7.57. On the other hand, if the RETE match algorithm is parallelized, the degree of concurrency can be increased by a factor of 2 to 9, because the number of WM changes (thus tokens) to be processed in parallel is increased.

We stress that this number shows only the effect of parallel firing, i.e. it does not include the effects of parallel matching, which may compress the production cycle itself. Thus more speed enhancement than shown in Table 2 can be expected from the total effect of parallel execution.

Sources of Parallel Executability

Parallelism in production system programs heavily depends on the nature of the problem addressed. Some production system programs, such as the *Monkey and Bananas* program, do not permit parallel firings at all. The potential parallelism in production system programs can be classified as follows:

- **Rule parallelism:** Multiple rules are fired in parallel without communication. For example, in the *Waltz labeling program* [Winston, 1977], many constraints can be applied independently. In the *Circuit design program* [Ishikawa et al., 1987], which is under development at NTT laboratories, many rules for optimizing a circuit can be applied in parallel. The rule parallelism increases as the number of independent rules becomes larger.

- **Pipeline parallelism:** Multiple rules are fired in parallel, passing data in a pipeline fashion. For example, in *Manhattan Mapper* [Lerner et al., 1983], which has been developed at Columbia University to provide travel schedules in Manhattan, the length of the pipe is six: some rules are fired to generate candidate paths, and then other rules evaluate them. In the Waltz labeling program, though there is no clear pipe, the data modified by some rules are further modified by other rules. The pipeline parallelism increases as the length...
7 Conclusion

We defined a parallel firing model of production systems, and proposed practical implementation methods for the model: the interference analysis, the parallel firing algorithm and the parallel programming environment.

The evaluation results on several production system applications show that the degree of concurrency can be increased by a factor of 2 to 9 by introducing parallel rule firing. Since the reported speed-up from parallel rule matching is 8.25 times [Gupta et al., 1986], we can say that thus parallel rule firing technique introduces another valuable source of parallelism in production systems.

Acknowledgment

The author wishes to thank Kunio Murakami, Ryohe Nakano and Salvatore J. Stolfo for their support to this work at NTT Laboratories and Columbia University. The author also thank Daniel P. Miranker and Hiroshi Okuno for their helpful discussions, and Yuzou Ishikawa and Mark Lerner for providing production system programs used in this study.

References


