Parallel Rule-Based Isotach Systems

Rashmi Srinivasa

Technical Report No. CS-99-04
February 5, 1999

Contact: rashmi@virginia.edu

Web: ftp://ftp.cs.virginia.edu/pub/techreports/CS-99-04.ps.Z

Abstract

The rule-based system is an important tool used to build expert systems. Much research has gone into trying to speed up rule-based systems, especially by using parallelism, but limited success has been observed so far. Most of the attempts have been in the direction of improving the Match-Recognize-Act (MRA) cycle [BROW85], which is the technique used in the sequential execution of rule-based systems.

We present a parallel algorithm (ISORULE) for the execution of rule-based programs, which is based on isotach time [REYN89], and which eliminates the MRA algorithm. We compare ISORULE to a conventional parallel technique based on MRA (PARAMRA). We describe the design and implementation of ISORULE, and analyze the ways in which ISORULE exploits parallelism. We describe a static model to analyze ISORULE performance, and use it to demonstrate that ISORULE exploits most of the statically determinable parallelism in a rule set. We detail our simulation of ISORULE and PARAMRA, and report results obtained by running synthesized as well as real rule-based programs on it. Our experiments with synthetic rule sets reveal order of magnitude performance improvement of ISORULE over PARAMRA. Since our initial analysis suggests that the performance improvement increases with the amount of concurrency in the rule sets, multiple orders of magnitude in performance improvement seem likely with larger rule sets exhibiting a low degree of conflict. Further, we analyze the results obtained from our initial investigation of real rule sets, and explain the more limited performance improvement we observed with these sets.

Contents

1 Abstract i

2 Contents ii

3 Introduction 1
3.1 Organization of Thesis 3

4 Rule-based Systems 5
4.1 Sequential Rule-based Systems 6
4.1.1 The MRA Cycle 6
4.1.2 Rete Match 8
4.2 Parallel MRA 9
4.2.1 Rule-level Match Parallelism 9
4.2.2 Node Parallelism 10
4.2.3 Intranode Parallelism 10
4.2.4 Action Parallelism 10
4.2.5 Data Parallelism 11
4.2.6 Overlapping Phases of the MRA Cycle 11
4.2.7 Multiple Rule Firing 11
4.3 Asynchronously Parallel Approaches 15
4.3.1 Offline Analysis 16
4.3.2 Locks Acquired by a Single Process 17
4.3.3 Locks Acquired by Multiple Processes 17
4.4 Chapter Summary 18

5 Isotach Rule-based Systems 20
5.1 Isotach Time and Atomicity 20
5.1.1 The Concept of Isotach Time 20
5.1.2 Atomicity 21
5.1.3 Flat Atomic Actions 22
5.1.4 Structured Atomic Actions 23
5.1.5 Sequential Consistency 24
5.1.6 Performance of Isotach Networks 25
5.2 Model of an Isotach Rule-based System 25
5.2.1 Assumptions 26
5.2.2 Partitioning 26
5.2.3 Types of Operations 28
5.2.4 Asynchronous Evaluation and Logical Firing Times 29
5.2.5 A Discussion of Correctness 30
5.2.6 Deadlock Freedom 32
5.2.7 Waiting Time 32
5.3 Parallelism 33
5.4 Chapter Summary 35

6 Implementation 37

6.1 Review of the Basic ISORULE Algorithm 37
6.2 Components of ISORULE 38
6.2.1 Rule Process 38
6.2.2 SIU Process 42
6.2.3 WME Process 43
6.2.4 Isotach Network 45
6.3 Optimizations 45
6.3.1 CANCELs and CONFIRMs 45
6.3.2 Speeding Up the Progress of a Firing Token 46
6.3.3 Throttling the Rule Process 46
6.4 A Conventional Parallel Paradigm 47
6.5 A Comparison of ISORULE to PARAMRA 48
6.6 A Comparison of ISORULE to Other Asynchronous Techniques 49
6.7 Chapter Summary 50

7 Performance Analysis 52
7.1 The Simulation 54
7.1.1 Workloads 54
7.1.2 Synthetic Workload 54
7.1.3 Real Workload 57
7.1.4 Simulation Parameters 58
7.2 A Static Model for ISORULE Performance 58
7.2.1 Dependence Analysis 59
7.2.2 A Static Model to Assess Parallelism 61
7.3 Synthesized Rule Sets 63
7.3.1 Analysis 63
7.3.2 Results from the Synthesized Rule Sets 65
7.3.3 Effect of Number of Processors on ISORULE Performance 70
7.3.4 Effect of Degree of Pipelining on ISORULE Performance 71
7.4 Real Rule sets 73
7.4.1 Sequential Nature of Rule-based Programs 73
7.4.2 The Monkey and Bananas Rule-based Program 73
7.4.3 The Tourney Rule-based Program 74
7.4.4 The Manners Rule-based Program 75
7.4.5 The Toru-Waltz Rule-based Program 76
7.5 Analysis of the Results 76
7.6 Chapter Summary 78

8 Summary and Conclusion 79
8.1 Future Work 81

9 Parameters to Simulation 83

10 References 85

Chapter 1
Introduction

The goals of Artificial Intelligence research include building expert systems and representing human cognition. To achieve these goals, different models have been developed for knowledge representation and application. One of the most widely used of these models is the **rule-based system** model. Many expert systems are expected to exhibit high performance in real-time domains or interactive domains, and hence, efficiency in rule-based systems is an important issue. Sequential executions of rule-based systems have been unable to meet these performance requirements. This thesis addresses the issue of speeding up the execution of rule-based systems through the use of an evolving technology called isotach timing [REYN89].

Several attempts have been made to speed up rule-based systems, especially by introducing parallelism into their execution paradigm; but these efforts at parallelism have had limited success so far. Most of the attempts have concentrated on improving the **Match-Recognize-Act** (MRA) cycle [BROW85], which is the technique used in the sequential execution of rule-based systems. Various types and granularity levels of parallelism have been explored in the endeavour to speed up different stages of the MRA algorithm. A few other approaches towards parallelizing rule-based system execution have eliminated the MRA cycle, and use an asynchronous paradigm instead. Existing parallel rule-based system paradigms experience one or more of the following problems.

- Failure to fully exploit the concurrency available
- Load imbalance
- Failure to share common tasks among processors

- High processor synchronization overhead
- High communication costs
- Memory contention
- Need for expensive runtime analysis
- Overly restrictive access to shared objects
- Cost of dealing with deadlock

In this thesis, we describe a parallel algorithm (**ISORULE**) for the execution of rule-based programs, which eliminates most of the problems listed above. The high-level design and implementation models for ISORULE are due to Craig Williams and Paul F. Reynolds. ISORULE is based on isotach networks [REYN89, WILL91], and does not use the MRA algorithm. Our simulation of ISORULE runs on an isotach network simulator due to Bronis R. de Supinski. This thesis makes the following contributions:

- We describe the design and implementation of ISORULE — a new technique for the parallel execution of rule-based systems — and analyze the ways in which ISORULE exploits parallelism.
- We detail the design and implementation of our simulation of the ISORULE system and of a conventional parallel MRA-based system (PARAMRA) for rule-based system execution.
- We use results obtained from the simulation to demonstrate that ISORULE performs significantly better than PARAMRA.
- We describe the design and implementation of a static model for the analysis of ISORULE performance.
- We use our static model and our simulation on synthesized rule sets to demonstrate

that ISORULE exploits most of the statically determinable parallelism in a rule set.

- We describe the introduction of a real Rete network algorithm [FORG82] and a parser for OPS5 [FORG81] rule-based programs into our simulation, and evaluate the performance of real rule-based programs on ISORULE.

Our experiments with synthetic rule sets show order of magnitude improvement in performance of ISORULE over PARAMRA. Since our initial analysis suggests that the performance improvement increases with the amount of concurrency in the rule sets, we believe that multiple orders of magnitude in performance improvement are possible with larger rule sets exhibiting a low degree of conflict. Other results demonstrate the effect of the number of processors and of pipelining on ISORULE performance. Also, we have begun an investigation of real rule-based programs with four programs written in the language OPS5 [FORG81]. Results from these rule sets are not as impressive as the order of magnitude improvement result we obtained from synthesized rule sets. We explore the characteristics of the rule sets and discover that the more limited performance improvement we observed is due to an embedded sequential control structure, and a small number of rules exhibiting a high degree of conflict.

Organization of Thesis

Chapter 2 introduces rule-based systems and describes the three components of a rule-based system: working memory, rules and inference engine. It explores existing sequential and parallel paradigms for execution of a rule-based system, and describes the MRA cycle. The various phases of the MRA cycle are discussed, including the Rete match algorithm. Chapter 2 further discusses different attempts at parallelization of rule-based

systems; these attempts differ in their levels of granularity. The advantages and disadvantages of each of these approaches are discussed.

Chapter 3 introduces our technique (ISORULE) for parallel rule-based system execution, which is based on isotach networks [REYN89, WILL91], and compares it with existing asynchronous and synchronous systems. Some background on isotach networks is provided, along with a discussion of the features which make them suitable for rule-based systems. This chapter presents the salient aspects of the ISORULE algorithm, and discusses the ways in which the system exploits the parallelism available to it from the rule set.

The implementation of the ISORULE system is discussed in detail in **Chapter 4**. The various components of the system, along with the functions they perform are detailed in the chapter, along with a description of optimizations that enhance ISORULE performance. A conventional parallel rule-based system (PARAMRA) is described, and compared with ISORULE.

In **Chapter 5** we describe a new method for statically evaluating the potential parallelism in a rule set. We predict and evaluate the performance of the ISORULE system compared to that of PARAMRA. Our simulation of ISORULE and PARAMRA is described, along with the workloads we used — both synthetic and real. The chapter presents the results obtained from the execution of the rule sets on ISORULE and PARAMRA, and analyzes the results. Finally, it elaborates rule-set-writing techniques which make the rule sets suitable for parallel rule firing.

Chapter 6 presents a summary and final conclusions, and proposes avenues for future research.

Chapter 2
Rule-based Systems

A rule-based system (also called a production system) is defined by a working memory, a set of rules in a **production memory**, and an inference engine. The **working memory** is a database of facts or assertions called working memory elements (WMEs). A working memory element has a class name, and a set of attributes with their corresponding values. An example of a WME (in the language OPS5[FORG81]) is:

```
(forecast ^where Boston ^when tomorrow ^weather rainy)
```

Here, `forecast` is the WME class name, `where`, `when` and `weather` are its attributes, and `Boston`, `tomorrow` and `rainy` are the corresponding values of the attributes.

A **rule** (or a **production**) consists of a conjunction of condition elements (called the **antecedent**), and a set of action elements (called the **consequent**). Firing of a rule involves execution of the action elements when the condition elements are satisfied. **Condition elements** can be positive (which are satisfied when a matching WME exists) or negative (which are satisfied when no matching WME exists). **Action elements** add new WMEs, or/and modify existing WMEs, or/and delete existing WMEs. Variables can occur in the place of values, and these variables are consistently bound through all of the elements of a rule. An OPS5 rule looks like this:

```
(p make-possible-trip
      (city ^name <x> ^state Massachusetts)
      -(forecast ^where <x> ^when tomorrow ^weather rainy)
      ->
      (make possible-trip ^where <x> ^when tomorrow))
```

The rule `make-possible-trip` has a positive condition element of class `city`, a nega-

tive condition element of class `forecast`, and an action element which adds a new WME of class `possible-trip` to the working memory. A rule instance with all its condition elements satisfied and all variables bound, is called an **instantiation**.

The **inference engine** (or interpreter) is the control mechanism which drives the rule-based system. It determines, for a certain state of working memory, which rules are eligible to fire, decides which to fire, and executes the rule(s). A variety of problem-solving strategies can be built into an inference engine. The two main reasoning strategies are **forward-chaining** and **backward-chaining** [BROW85]. In forward-chaining (also called **data-directed inference**), the series of inferences proceeds from antecedents to the corresponding consequents. The backward-chaining (also called **goal-directed inference**) process starts from a specification of the desired consequents, and proceeds by trying to prove antecedents that will justify concluding the consequent. Many production systems use both types of reasoning, and one can be emulated by the other. An inference engine that directly supports backward-chaining must have a large amount of control built into it, and therefore, backward-chaining production systems suffer from reduced flexibility. In this thesis, we consider forward-chaining production systems.

2.1 Sequential Rule-based Systems

The inference engine in a sequential rule-based system executes an algorithm called the **Match-Resolve-Act** (MRA) cycle [BROW85]. We describe the MRA cycle executed by a forward-chaining inference engine.

2.1.1 The MRA Cycle

The MRA cycle is made up of three phases which are repeatedly executed by the

rule-based system interpreter. In the **Match** phase, the facts in the working memory are matched against the condition elements of the rules, and a set of satisfied rule instantiations is determined. This set is called the **conflict set**.

The **Resolve** phase consists of selecting one of the rule instantiations in the conflict set to fire. This selection is carried out based on a conflict resolution strategy. Two of these strategies are MEA and LEX [FORG81]. LEX uses four steps to successively eliminate rule instantiations from the conflict set so that the one remaining instantiation can be fired. The first step is *refraction*, which means that all instantiations previously selected and fired are deleted from the conflict set. The second step of LEX discards all but the instantiations whose condition elements matched the most recently changed WMEs. The third step selects the instantiation(s) with the maximum number of tests in the condition elements of that rule. And finally, step four arbitrarily selects one of the remaining instantiations for firing. If there is a unique instantiation left in the conflict set at the end of any step, that instantiation is selected, and no further steps are necessary. The MEA strategy adds one extra test just after the refraction step of LEX. This test selects the instantiation(s) whose *first* condition element matches the most recent WME(s). This strategy is useful to handle subgoaling, that is, when the first condition element of a rule is always a *goal* element. Another way to specify control over rule selection is by using *metarules* [DAVI80]. Metarules are rules which determine how to apply other rules, and can be written in the same language as the normal rules, or in a separate control language. LEX and MEA can both be coded as metarules.

The selected rule instantiation is fired in the **Act** phase, and the working memory is modified accordingly. The MRA cycle is repeated until one of two things happens: either

the conflict set is empty, that is, there are no more rule instantiations to fire; or an explicit halt is executed as a result of firing a rule instantiation.

It has been observed that Match constitutes the most expensive phase of the MRA cycle. Literature puts the percentage of MRA cycle time that Match takes, from more than 90% [FORG79] in earlier systems, to about 50% [KUOS91] in more recent ones. The **Rete** algorithm [FORG82] has become the standard match algorithm for rule-based systems.

2.1.2 Rete Match

The Match phase of the MRA cycle involves matching a large number of patterns (represented by the condition elements of the rules) against a large collection of facts (WMEs). When one realizes that a typical working memory consists of hundreds of classes each with ten to a hundred attributes [FORG82], it is easy to see why the Match phase is so time-consuming.

The Rete match algorithm requires compilation of the condition element patterns into a data-flow network called the **Rete network**. The match involves no iteration over WMEs because state is saved between cycles. Each time a new WME enters the working memory, this state is updated. The Rete network also avoids iteration over the rules by acting as an index for the rules. Common test evaluations are shared between rules instead of being repeated for each rule.

Faster sequential match algorithms have been invented, like TREAT [MIRA87] and YES/RETE [HIGH89]. TREAT implements faster *deletions* from the working memory, at the cost of slower *additions* to the working memory. If deletions are more frequent than additions, TREAT obtains better performance than Rete. YES/RETE implements

faster *updates* to the working memory, and also increases the amount of sharing in the Rete network.

2.2 Parallel MRA

Attempts have been made to speed up the execution of rule-based systems, by parallelizing parts of the basic sequential MRA algorithm. These attempts vary in the granularity of parallelism as well as on the portion of the MRA algorithm which is divided up among the processors. We discuss sources of parallelism in rule-based programs with respect to the classic MRA control structure. However, several of these sources of parallelism are applicable to other control structures. Alternatives to the MRA control structure are discussed in Section 2.3.

2.2.1 Rule-level Match Parallelism

In this approach, the **set of rules** in the production memory **is partitioned**, and the subsets are allotted to the available processors. Each processor independently performs the Match phase on the rules in its subset. One processor performs the Resolve phase and selects a firable rule instantiation from the conflict set obtained collectively by all the processors doing the match. The advantage of this strategy is that no interprocessor communication is required for performing the match. But there are several disadvantages, one being that of load balance: different rules may need different amounts of time to be matched, and so some of the processors might be idle while the others are working away at their Match phases. Another cause of processor idleness is the **small cycle problem** [GUPT86]: each working memory change affects very few rules, and so there is very little work for the match processors to do in each iteration of the MRA cycle. Yet another disadvantage is

loss of sharing in the Rete network: since each processor has a separate Rete network for its own subset of the rule set, tests common to rules belonging to two different processors are not shared.

2.2.2 Node Parallelism

This strategy exploits finer grain parallelism by **partitioning the set of nodes** in the Rete network, and allotting subsets to several processors. The advantage over the rule-parallelism approach is that common tests are shared, since there is a single Rete network. But on the minus side, more interprocess communication is required during the Match phase. Two examples of systems which have implemented or simulated this technique are PSM-E [GUPT88] (an implementation) and PSM-M [ACHA89] (a simulation).

2.2.3 Intranode Parallelism

Going on to even finer grain parallelism, intranode parallelism allows multiple activations of the **same** Rete network node to be evaluated in parallel. This is in addition to allowing **different** Rete network nodes to be evaluated in parallel as in the node parallelism approach. An obvious disadvantage is that there is strong memory contention at the working memory, since several processors will try to access the same WMEs. PESA I [SCHM90] and DRete [KUOS90] are systems which use intranode parallelism in the Match phase.

2.2.4 Action Parallelism

A rule instantiation firing causes WME changes. These WME changes are processed concurrently in the action parallelism technique. Since very few WME changes are caused per firing (about 2.4 for OPS5 programs [GUPT89]), potential parallelism is low.

Action parallelism can be used in conjunction with the intranode parallelism strategy.

2.2.5 Data Parallelism

This method relies on extremely fine grain parallelism. The processing required for each node activation in the Rete network is done in parallel. This implies very high synchronization and scheduling costs since the method is so fine grain. It works fairly well with specialized hardware like DADO [STOL82], and other parallel machines like NON-VON [SHAW85] and the Connection Machine [WALT87].

2.2.6 Overlapping Phases of the MRA Cycle

Moving on to the other phases in the MRA cycle, one can achieve an improvement in performance by overlapping the Resolve and the Act phases [GUPT89]. This is done by following these steps:

- Guess which rule instantiation is going to fire next. (For instance, the second-best rule instantiation from the Resolve phase of the previous cycle.)
- Evaluate potential WME changes by that rule instantiation.
- After the actual Resolve phase, if the guess was right, just change the working memory; if the guess was wrong, reevaluate the changes to be made to working memory.

As long as the ratio of good guesses to bad guesses is reasonably high, performance of the system improves.

2.2.7 Multiple Rule Firing

An important issue in parallel rule firing is **conflict** among rules. The firing of one

rule instantiation can make another rule instantiation invalid. A non-conflicting set of rule instantiations can be selected from the conflict set by building a **data dependence graph**. Rule data dependence has been explored in [DIXI87], [ISHI90], [KUOS91], [MIRA90] and [SCHM90].

If a WME is added by a rule, the WME class is said to be "plus-changed" by that rule. If a WME is deleted by a rule, the WME class is "minus-changed" by that rule. If a WME occurs in a positive condition element of a rule, that WME class is "plus-referenced" by the rule. And if a WME occurs in a negative condition element of a rule, the WME class is said to be "minus-referenced" by that rule.

Potential conflict exists between two rules when there is a WME class that is

- plus-changed by one and minus-referenced by the other; or
- minus-changed by one and plus-referenced by the other; or
- plus-changed by one and minus-changed by the other.

These are the **pairwise conditions** for potential conflict or interference.

More parallelism can be captured by using **cyclic conditions** instead of the pairwise conditions. Here, the constraint is that a set of rule instantiations cannot be executed in parallel if the set has a **data dependence cycle**. For instance, take three rules *ruleA, ruleB* and *ruleC*. Consider the two cases shown in figures 2.1 and 2.2 [ISHI94]. In figure 2.1, the three rules can be fired in parallel because there is a sequential execution (*ruleC* => *ruleB* => *ruleA*) which is equivalent to the parallel execution. But in figure 2.2, there is a cyclic dependence, and there is no sequential execution which produces the same result as any parallel execution.

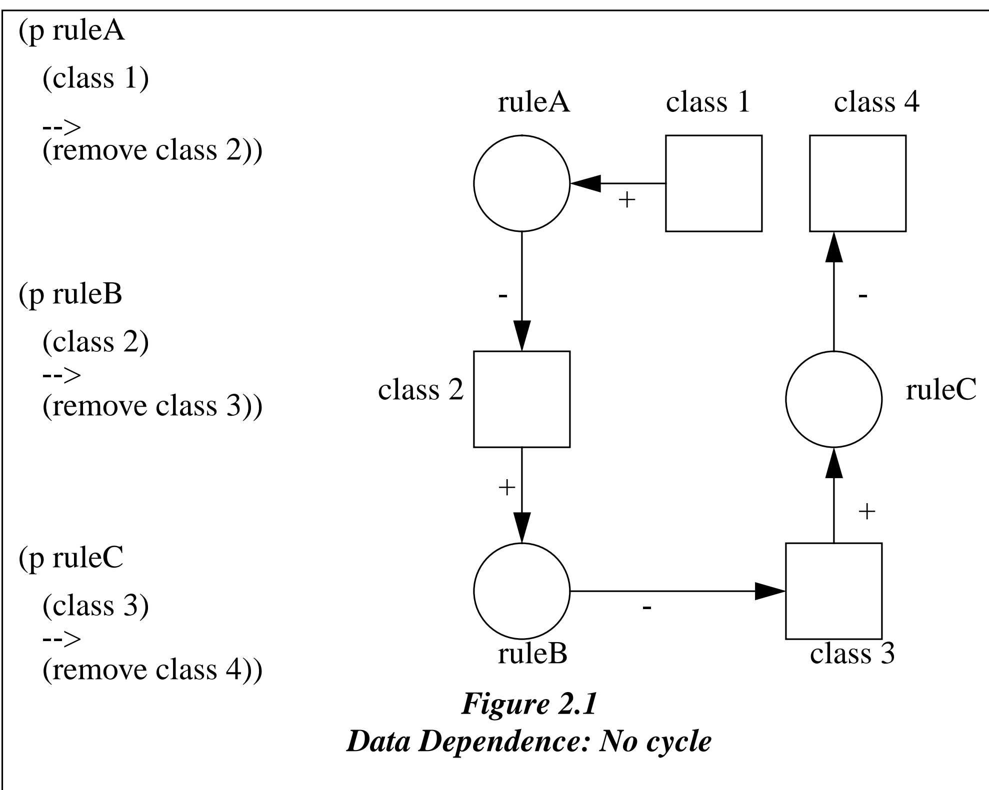


Figure 2.1
Data Dependence: No cycle

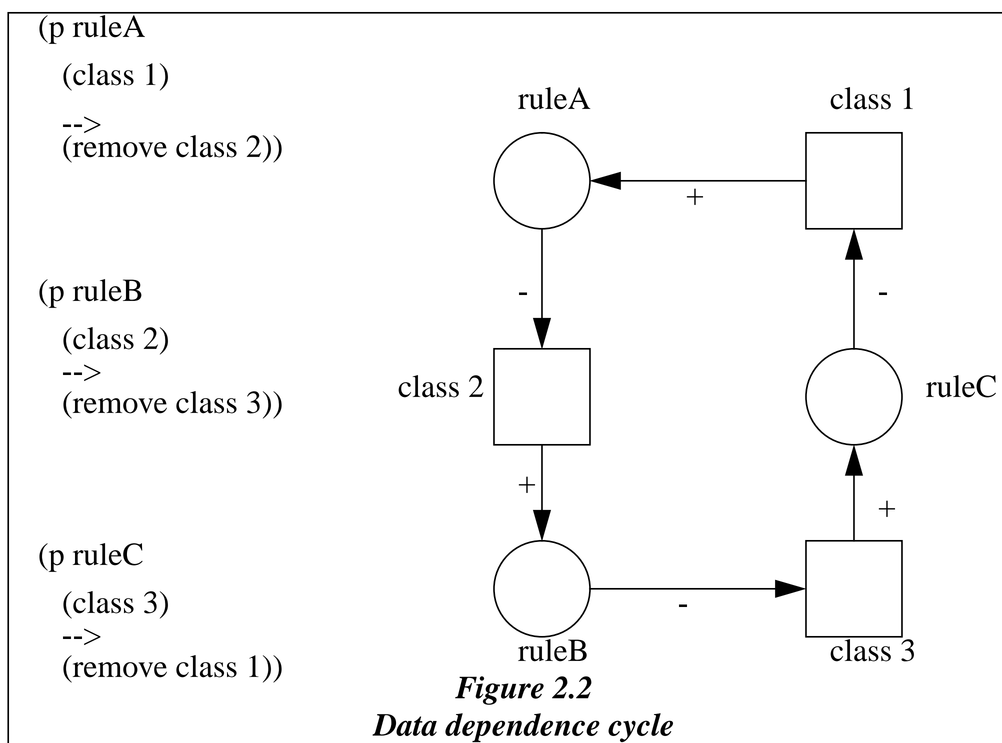


Figure 2.2
Data dependence cycle

The cost of computing cyclic dependences is, however, higher than that of computing pairwise dependences, because the constraint for detecting cyclic dependence requires that all strongly connected regions be detected from a data dependence graph.

Multiple rule firing alleviates the small cycle problem by providing more working memory changes per cycle, thus keeping more match processes busy. However, it comes with the cost of having to do **interference analysis** to ensure **serializability**. A parallel execution is *serializable* if there exists an equivalent serially correct sequential execution. Serializability is the most widely used correctness criterion for concurrency control in databases [BERN81, PAPA86].

Interference analysis (by using either the pairwise or the cyclic conditions) can be done either at compile-time or at run-time. While **compile-time analysis** is less expensive computationally, **run-time analysis** has the advantage that all variables are bound at run-time, and it is possible to detect dependences among *rule instantiations* rather than just among *rules*. A mixture of compile-time and run-time analyses is what is commonly used [ISHI90, KUOS91, MIRA90].

Parallelism gained from firing multiple rule instantiations concurrently stems from the following three sources:

- *Rule Parallelism*: If there is a set of rules in which no rule conflicts with (invalidates) another rule in that set, the rules in this set can be fired in parallel without synchronization among the PEs that fire the rules. The larger the set of non-conflicting rules, the greater the rule parallelism offered by the rule-based system program.
- *Pipeline Parallelism*: In pipeline parallelism, data is pipelined through rules. As an

example, let rule R_1 fire its first instantiation which enables rule R_2 to fire its first instantiation. *While R_2 is firing its first instantiation*, R_1 can go ahead and fire its *second* instantiation which enables R_2 to fire *its* second instantiation, and so on. This is an example of a rule pipeline of length two, and is shown in figure 2.3. Once the pipe is full, the rules forming the pipe can be fired concurrently. The longer the pipe, the greater the pipeline parallelism.

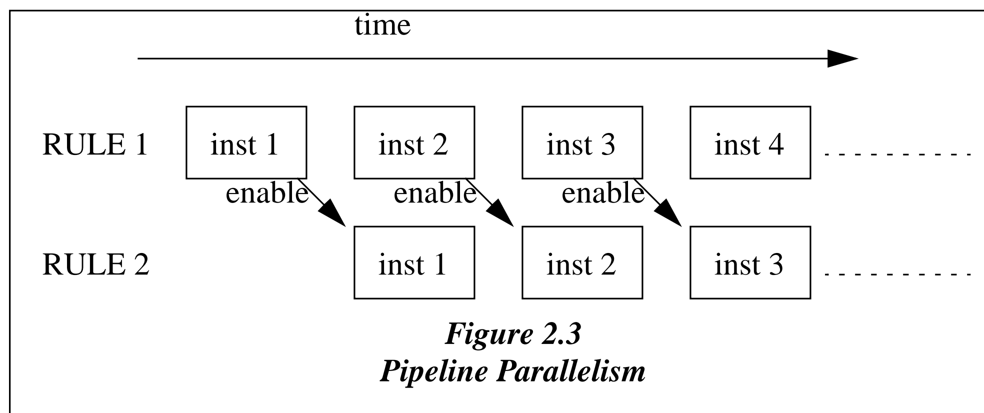


Figure 2.3
Pipeline Parallelism

- *Instantiation Parallelism*: More than one instantiation of a rule may be valid at a given time, and these instantiations can be fired in parallel.

Systems that use multiple rule firing include IRIS [ROMA89], Ishida's simulated system [ISHI85], PARS [SCHM90], RUBIC [MOLD89], CREL [KUOC91] and PARULEL [STOL91]

2.3 Asynchronously Parallel Approaches

We know of three parallel rule-based system schemes which have eliminated the MRA cycle, allowing multiple instantiations to fire simultaneously. These systems use an alternative control structure to ensure that the results obtained by the parallel execution

could have been obtained by some sequential execution. The algorithms that have been proposed to guarantee this property of serializability are described below. Several of the sources of parallelism described in Section 2.2, most notably rule parallelism, pipeline parallelism and instantiation parallelism (Section 2.2.7), can be exploited by asynchronous approaches.

2.3.1 Offline Analysis

Schmolze and Nieman have proposed an algorithm which uses offline analysis to aid in guaranteeing serializability. [SCHM92]

They define a "clash" between two rule instantiations as the property that one of the instantiations plus-changes a WME and the other minus-changes the same WME. Also, a rule instantiation "disables" another rule instantiation if the first plus-changes and the second minus-references the same WME, or if the first minus-changes and the second plus-references the same WME.

To ensure serializability, no two coexecuting instantiations must clash, and there should be no cycle of disables. Offline analysis synthesizes LISP functions that check for clashes and disables between two instantiations.

In their system, there are several demon processes and a scheduler process. The scheduler process schedules instantiations from the conflict set and places them in a shared queue. A demon process removes one of the scheduled instantiations from the shared queue (there is a mechanism which prevents multiple demon processes from selecting the same instantiation), and adds it to the list of executing instantiations (by executing a many-reader single-writer critical code section). The demon process tests this instantiation for clashes/disables with the set of previously enqueued executing instantiations, and

then checks to see if the system has marked the instantiation *dead* in the meantime, due to it being disabled by the execution of an instantiation by some other process. If the instantiation is not dead, and no clashes/disables are detected, the instantiation is executed by the demon process and removed from the list of executing instantiations. Otherwise, the instantiation is removed from the list of executing instantiations, and returned to the conflict set if it is not dead. Each demon process repeats this procedure until the shared queue of scheduled instantiations is empty. The scheduler process keeps trying to schedule instantiations from the conflict set until the conflict set is empty, the shared queue is empty, and all the demon processes are idle, in which case it exits.

2.3.2 Locks Acquired by a Single Process

Another technique, also by Schmolze and Nieman [SCHM92], uses locks to perform synchronization. There is no checking for clashes or disables. The scheduler process tries to acquire locks for an instantiation. If the required locks can be obtained, the rule instantiation is scheduled. Each demon process chooses an instantiation out of the ones the scheduler has scheduled, and executes it. Since acquisition of locks is a sequential operation performed only by the scheduler process, there is no possibility of a deadlock. Also, a demon process doesn't have to perform any checks to make sure that it is safe to execute that instantiation.

2.3.3 Locks Acquired by Multiple Processes

Schmolze also proposed an asynchronous system PARS [SCHM88, SCHM90] in which different processors coordinate rule firings by acquiring locks. PARS modifies the phases of the MRA cycle, and adds two more phases. The five phases are: *match, select,*

disable, act and *enable*. The rule set is divided up among the processors, and the WMEs referenced by the rules allotted to a processor are stored at that processor. Each processor executes the PARS cycle asynchronously.

In the **match** phase, the rules are matched against the WMEs, and a local conflict set is formed. In the **select** phase, one of the instantiations from the local conflict set (which is not *disabled*) is selected. If no such instantiation exists, the process goes back to the *match* phase, else it proceeds to the *disable* phase. In the **disable** phase, the processor sends *disable* messages to all of the processors which have rules that are incompatible with this one. To prevent **deadlock** caused by cyclic *disable* messages, rules are prioritized, and *disable* messages are sent only to processes that have incompatible rules with a priority *higher* than this one. The processor then waits for receipt of messages confirming the disabling of all of the incompatible rules. In the interim, if this rule is invalidated, the processor has to abandon the execution of the rule and jump to the *enable* phase. Otherwise, the *act* phase is executed. In the **act** phase, the selected rule instantiation is fired, and the resulting WME changes are sent to the affected processors. The processor then waits for acknowledgments from all receiving processors, and sends reacknowledgments to all of them. In the **enable** phase, the rules which were disabled in the *disable* phase are enabled by sending *enable* messages.

2.4 Chapter Summary

A rule-based system is composed of a working memory, rules and an inference engine that drives the system. A rule is made up of condition elements and action elements. Sequential rule-based systems execute the MRA (Match-Resolve-Act) cycle. The condition elements are matched against WMEs in the *match* phase. The *resolve* phase uses

a conflict-resolution strategy to select a firable instantiation from the conflict set. The selection can be done by implicit strategies like MEA and LEX, or by embedded metarules. The instantiation is fired in the *act* phase. The Rete algorithm is the most popular match algorithm.

Existing parallel rule-based systems have explored different types of parallelism: rule-level, node, intranode, action, data, overlapping phases and multiple rule firing. We describe these parallelism sources with respect to the classic MRA control structure, but several of these sources are applicable to alternative control structures too. Data dependence analysis of the rules is used in multiple rule firing systems. The only previously published asynchronously parallel approaches use either offline analysis or locks to synchronize multiple rule firing. Section 4.6 provides a qualitative comparison of these asynchronous approaches to the technique we present.

The approach we present falls under the asynchronously parallel multiple rule-firing category, but differs from the other asynchronous techniques described in that it does not introduce locking, and eliminates the bottleneck of a central control process.

Chapter 3
Isotach Rule-based Systems

We present a parallel rule-based system algorithm (henceforth called **ISORULE**), which is based on **isotach networks** [REYN89, WILL91]. An isotach network uses an embedded logical time network to reduce the cost associated with the coordination of multiple processors in a parallel system. ISORULE does away with the MRA cycle, and coordinates processors by using a completely different paradigm. ISORULE differs from other asynchronous techniques in that it does not introduce locking, and eliminates the bottleneck of a central control process. Each rule in the ISORULE system executes atomically, and rules fire asynchronously. Rules can be selected for execution whenever the state of the working memory renders them executable.

3.1 Isotach Time and Atomicity

Isotach networks support **atomicity** — a property which, as we shall see, is quite useful to the efficient execution of parallel rule-based systems. With isotach time, a processor can control the time at which the messages that it sends are processed at the receiving processor.

3.1.1 The Concept of Isotach Time

Isotach time is an extension of Lamport's logical time [LAMP78]. Lamport's logical time system expresses potential causality by three rules: Event A happens before event B $(A \rightarrow B)$, and A is assigned a timestamp less than the one assigned to B if

- A takes place before B, on the same process;

- A is a message-send and B is the receipt of the same message; or
- there exists an event C such that $A \rightarrow C$ and $C \rightarrow B$.

A message is *received* when the SIU (switch interface unit) of the destination process removes the message from the network. Isotach logical times are lexicographically ordered n-tuples of integers, of which the first is called the *"pulse"*. There are two changes to Lamport's system:

- If $A \rightarrow B$, then the time assigned to A is less than **or equal to** the one assigned to B; and
- More importantly, isotach networks maintain the **isotach invariant**: a message is received exactly *d* pulses after it is sent, where *d* is the logical distance the message travels.

In this discussion, the isotach logical time n-tuple is of the form *(pulse, pid, rank)* where *pid* is the identifier of the process that issued the message, and *rank* is the issue rank of the message, that is, $rank = r$ if the message is the rth message issued by the process identified by *pid*. Since a processing element (PE) can control the logical time of receipt of any message it sends in an isotach system, an isotach network gives the power to enforce properties like atomicity and sequential consistency.

3.1.2 Atomicity

An atomic action is a group of operations issued by a process, where the set of operations appears to be executed indivisibly (without interleaving with other operations). Conventional systems usually use **locks** to enforce atomicity; but locks come with some disadvantages; there is lock maintenance overhead, overly restrictive access to shared

objects, and the possibility of deadlock and livelock. Atomic actions can be **fla**t or **structured**.

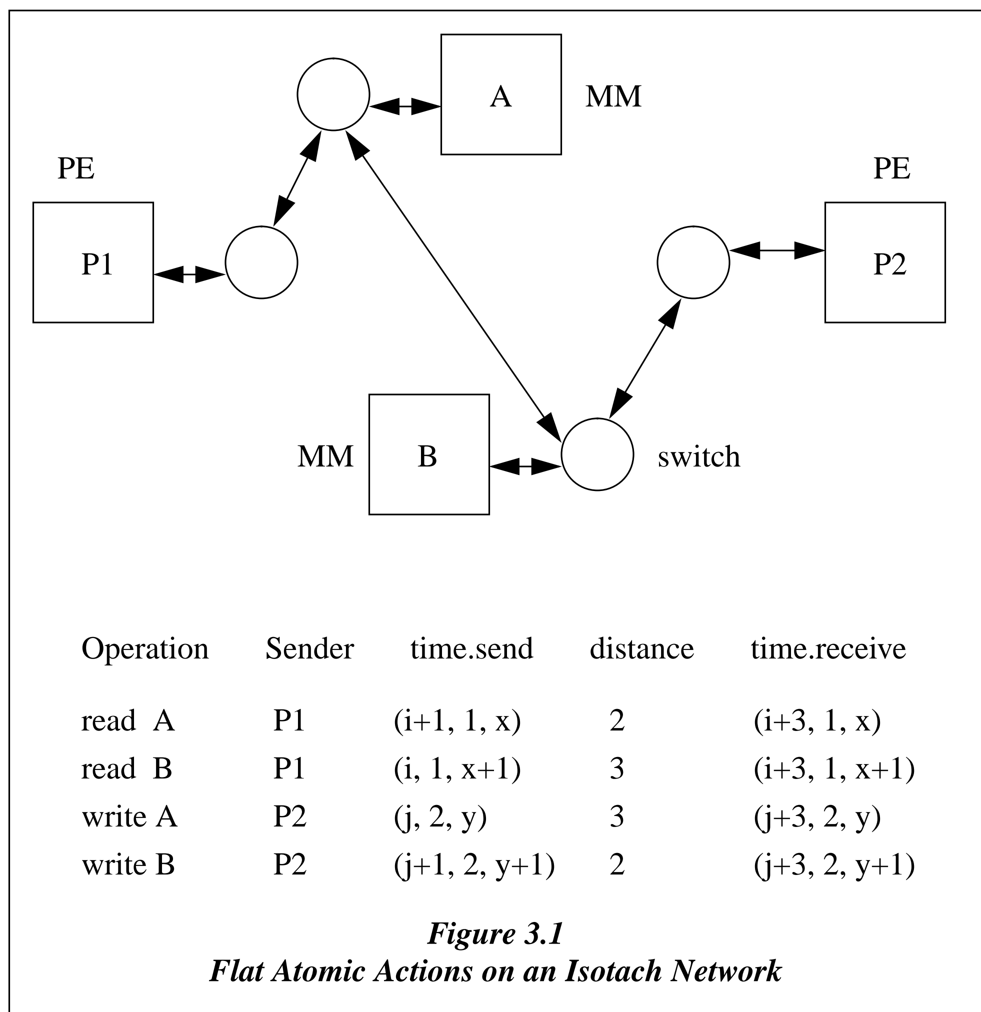
3.1.3 Flat Atomic Actions

Structured atomic actions can have **internal data dependences** among shared variables. Flat atomic actions can have no such dependences. Isotach systems can execute flat atomic actions without synchronizing with other processes, and can execute structured atomic actions without acquiring locks.

A processor in an isotach system executes a flat atomic action by sending all operations in the atomic action so that they are received at the destinations in the same pulse. An *equidistant network* is a network in which the distance (number of switches) between any two PEs is the same. For an equidistant network, the condition for execution of a flat atomic action would reduce to: Send all operations in the flat atomic action in the same pulse.

Consider the non-equidistant network shown in figure 3.1. *A* and *B* are shared variables in two memory modules, *P1* and *P2* are processing elements, and the circles are switches. *P1* needs to *read* the variables *A* and *B* atomically, and *P2* needs to *write A* and *B* atomically. In a conventional system, one of the PEs would lock the two variables (or two sets of variables), and perform its operations, during which time the other processor would be unable to do anything. Once the locks were released by this processor, the other processor would go ahead and acquire locks, and perform **its** operations. Provisions would have to be made to avoid deadlock. But in an isotach system, *P1* and *P2* could execute their atomic actions asynchronously as follows: *P1* sends the *read* on *A* one time pulse after it sends the *read* on *B*, so that both the *read*s arrive at their destinations in the same pulse. *P2*

sends its *write* on *B* one pulse after it sends the *write* on *A*, again to ensure that both the *write*s arrive at their destinations in the same pulse. Since the isotach system maintains the isotach invariant, both operations in each atomic action are received in the same pulse. If all four operations happen to reach their destinations in the same time pulse, the executions will still be atomic because the operations in the same pulse will be executed in order of *pid* of the sender.



Operation	Sender	time.send	distance	time.receive
read A	P1	(i+1, 1, x)	2	(i+3, 1, x)
read B	P1	(i, 1, x+1)	3	(i+3, 1, x+1)
write A	P2	(j, 2, y)	3	(j+3, 2, y)
write B	P2	(j+1, 2, y+1)	2	(j+3, 2, y+1)

Figure 3.1
Flat Atomic Actions on an Isotach Network

3.1.4 Structured Atomic Actions

The data-dependences among operations in structured atomic actions require an

extension to the basic isotach atomicity technique described in the previous section. This extension is implemented in form of **split operations** [WILL93]. A split operation performs an access in two steps: **scheduling** the access, and **transferring** the value. If a process has incomplete knowledge about an access due to an unsatisfied data dependency, it can reserve a slot in the variable's history, which ensures that the access will appear to be executed in the same time pulse as the other operations in the structured atomic action. An **unsubstantiated *write*** or a ***sched*** (a *write* which has been scheduled but not executed) holds up *read*s scheduled up to the next *write*; but other *read*s and *write*s are not delayed.

A processing element in an isotach system executes a structured atomic action by issuing a set of split operations scheduling all of the *read*s and *write*s making up the atomic action, so that they're received in the same logical time pulse at each variable accessed. As the values to be assigned are determined, the second step of each access — the ***assign*** — is performed. On the other hand, if the process determines that it is no longer able to perform the *write*, it can ***cancel*** the scheduled *write.* So the set of operations in the structured atomic action effectively reserves a consistent time slice over the access-histories of the concerned variables, thus guaranteeing atomicity.

3.1.5 Sequential Consistency

Another property provided by isotach systems is *sequential consistency.* A sequentially consistent execution is one in which the overall order of execution of operations is consistent with the order of execution implied by each individual process' sequential program [LAMP79]. In conventional systems, sequential consistency would be enforced by disallowing pipelining; so a process would have to wait for information telling it of the execution of its outstanding operation before it can issue its next operation. An isotach

network imposes no such restrictions on pipelining. To enforce sequential consistency in an isotach system, all that a processor has to do is timestamp each *send* operation so that it is received in a pulse greater than or equal to the pulse in which the preceding operation was received.

3.1.6 Performance of Isotach Networks

An isotach network has lower *raw power* than a comparable conventional network This means that assuming that there are no atomicity or sequential consistency constraints, an isotach network would be slower than a conventional network because the former has to bear the cost of maintaining isotach logical time. However, a simulation study [REYN92] comparing the performance of isotach networks to that of conventional networks has revealed that isotach networks outperform conventional networks when confronted with a workload imposing atomicity and sequential consistency constraints. The improvement in both throughput and delay has been observed to be more than ten-fold in some cases.

3.2 Model of an Isotach Rule-based System

The importance of **atomicity** in rule-based systems lies in the fact that correct execution of a rule demands the satisfaction of the condition elements of the rule *at the time the action elements are executed*. So a rule is a structured atomic action where the condition elements are *read*s to some variables, and the action elements are *write*s based on the values returned by the *read*s. There is a data-dependence because the action elements depend on the truth of the condition elements. ISORULE does not currently handle metarules; but if it did, the order of execution of rules would become important, and the

property of **sequential consistency** would be useful to guarantee this ordering. Recall that metarules impose an ordering on the execution of normal rules, and the same order has to be perceived at all of the processors in the system.

In ISORULE, rules fire asynchronously, depending on the underlying isotach network to guarantee atomicity and hence correct execution. Processing elements don't synchronize with other PEs before they schedule a rule instantiation firing. A firing can be scheduled whenever the rule is eligible, that is, whenever there is a firable instantiation of the rule. Eliminating the MRA cycle removes the loss of rule-parallelism that is inherent in the MRA algorithm. This section gives an overview of the ISORULE algorithm.

3.2.1 Assumptions

We assume in the course of this discussion that the PEs are connected in an equidistant network of *delta* stages. This assumption can be removed. We also assume that the rule set is static, that is, rules are not added, deleted or modified in the course of execution of the rule set. This assumption too, as we shall see, is not necessary. An assumption that cannot be removed as yet, is the assumption that there are no metarules. And finally, we assume that any serializable execution of the rules is a correct execution.

3.2.2 Partitioning

The set of rules is partitioned among the processors. These are some of the terms used in the discussion that follows:

- The **read set** for a PE is the set of WME classes referenced by the condition elements of each of that PE's rules.
- The **write set** for a PE is the set of WME classes referenced by the action elements

of each of that PE's rules.

- The **reader set** for a WME class is the set of PEs whose read set includes that class.
- The **writer set** for a WME class is the set of PEs whose write set includes that class.

Assuming a static rule set, these sets are statically determinable and don't change during the course of the execution. But with dynamic rule sets, these sets could change while the rule set was being executed. Isotach networks can provide the basis for building systems which handle dynamic rule sets as well, by using a technique similar to delta cache protocols [WILL93]. We assume a static rule set here.

Each PE in the ISORULE system is given a copy of all of the WMEs belonging to its read and write sets. Each PE is also given a copy of the reader set for each of the WME classes in its write set. That is, each PE knows which PEs will read the WME classes that it changes in the course of a rule firing. This completes the **preprocessing** phase of the ISORULE algorithm.

ISORULE has to ensure that the set of WMEs in each PE's local memory is up-to-date. This is similar to the problem of maintaining cache coherence, but simpler. The reader and writer sets are analogous to the set of PEs in a cache block's directory. In cache coherence protocols, these sets can change during execution. Delta cache protocols have been used to solve the cache coherence problem by using isotach networks, and the ISORULE technique is similar to the delta cache technique. The simpler nature of the rule-based system problem lies in the fact that since the reader and writer sets in a rule-based system are static, a PE changing a WME knows the destinations to which it has to multicast the change. If the sets were dynamic, the PE would have to send the change to a home node that tracks the reader or writer set.

3.2.3 Types of Operations

The operations used in the ISORULE system are based on the split operations — ***sched***, ***assign*** and ***cancel*** — discussed in Section 3.1.4. A ***SCHED*** is the logically synchronous multicast message that a processor issues to schedule the WME changes caused by a rule firing. In this case, the WME changes to be made are actually known at the time of sending the SCHED, so it is not really an *unsubstantiated* write. The receiver of a *SCHED* does not make the WME changes scheduled by the multicast, but instead, waits for a corresponding ***CANCEL*** or ***CONFIRM*** from the issuer of the *SCHED*. Figure 3.2 shows multicasts of *SCHED*s, *CANCEL*s and *CONFIRM*s.

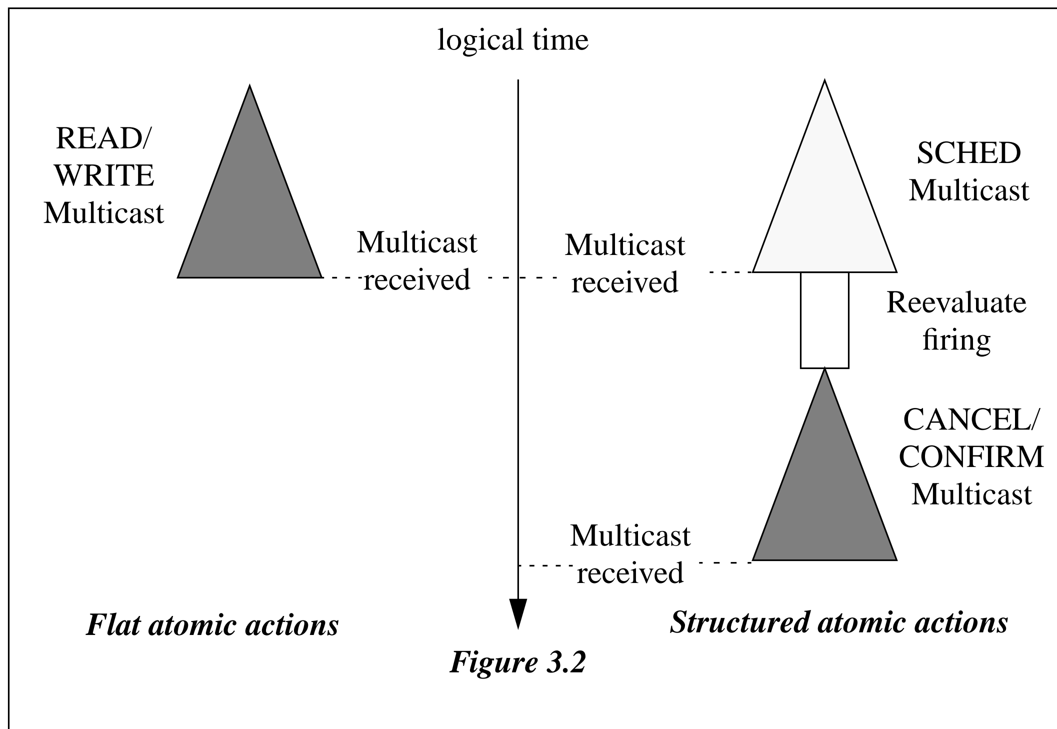


Figure 3.2

On receiving a *CANCEL*, a process deletes the corresponding *SCHED*. The *CONFIRM*, a variation of the *assign* type of split operation (Section 3.1.4), is used to signal the receiver that the previously scheduled WME changes are valid.

3.2.4 Asynchronous Evaluation and Logical Firing Times

Each PE evaluates (performs match on) the rules in its partition asynchronously. The logical firing time(LFT) of a rule firing is chosen to be the logical time at which the SCHED for that rule firing is received by the PEs in the firing PE's reader set for the selected rule. More formally, if *P* is the *pid* of the process scheduling the firing, *t* is the *pulse* in which process *P* multicasts the *SCHED*, the *SCHED* is the rth multicast by process *P*, and *delta* is the distance between any two PEs in the equidistant network, then LFT = (*t+delta, P, r). * Recall that with isotach systems, a PE can send out a multicast of operations and have all of the operations received at the same logical time at all destinations. Therefore, all receivers agree on the value of the LFT. Note that with a non-equidistant network, LFT could still be the logical time of receipt of the SCHED, and the isotach network would ensure that this LFT was the same at all receiving PEs (that is, the SCHED multicast was received by all destinations in the same logical time pulse).

Each PE in the ISORULE system executes the following algorithm:

- The PE tentatively schedules a rule firing by sending the SCHEDs for a rule firing so that they arrive in the same pulse at all PEs in the reader set for that rule.
- The PE re-checks the validity of the rule firing at the LFT for that firing, by reevaluating the rule.
- If the instantiation is still valid, the PE confirms the scheduled changes by sending CONFIRM messages, else it cancels them by sending out CANCELs to the PEs in its reader set.

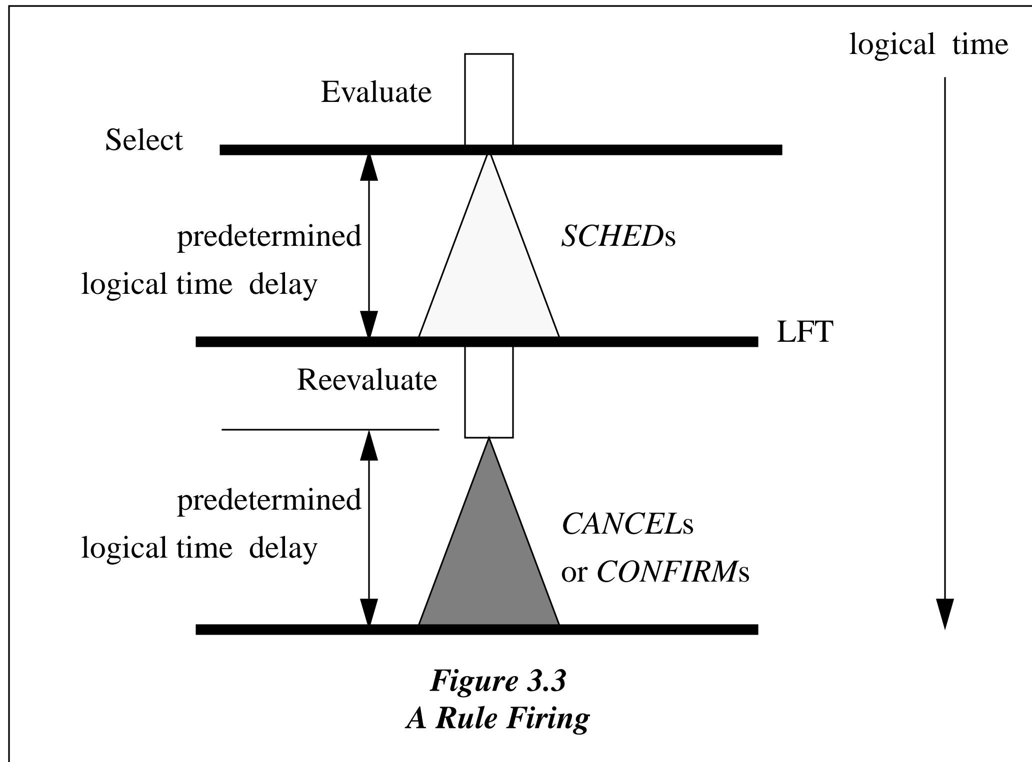


Figure 3.3
A Rule Firing

Figure 3.3 shows this procedure of evaluation, scheduling, reevaluation and confirmation or cancellation for a rule firing. The waiting time (the rectangles marked *Evaluate* and *Reevaluate*) is indeterminate, but bounded.

3.2.5 A Discussion of Correctness

Figure 3.4 shows the ISORULE system execution for two rules R_1 and R_2, where the firing of R_2 invalidates R_1, and the firing of R_1 has to be cancelled. Here, processor PE_2 schedules an instantiation of rule R_2 to fire, and sends out a SCHED multicast. Before the SCHED can get to processor PE_1, PE_1 schedules an instantiation of rule R_1 and multicasts the SCHED for it. The SCHED for R_2 arrives at PE_1 at the logical time LFT(R_2). Note that PE_1 should not cancel R_1 at this point because R_2 might get cancelled. Now PE_2 completes reevaluation and sends out a CONFIRM multicast. On receiving the CONFIRM

and on processing the WME changes caused by R_2, PE_1 knows that R_1 has to be cancelled, as it has been invalidated by R_2. So PE_1 sends out a CANCEL multicast, and abandons execution of the R_1 instantiation. Since the LFTs for all rule firings are agreed upon by all processors, the observed firing order of rules is the same at all processors.

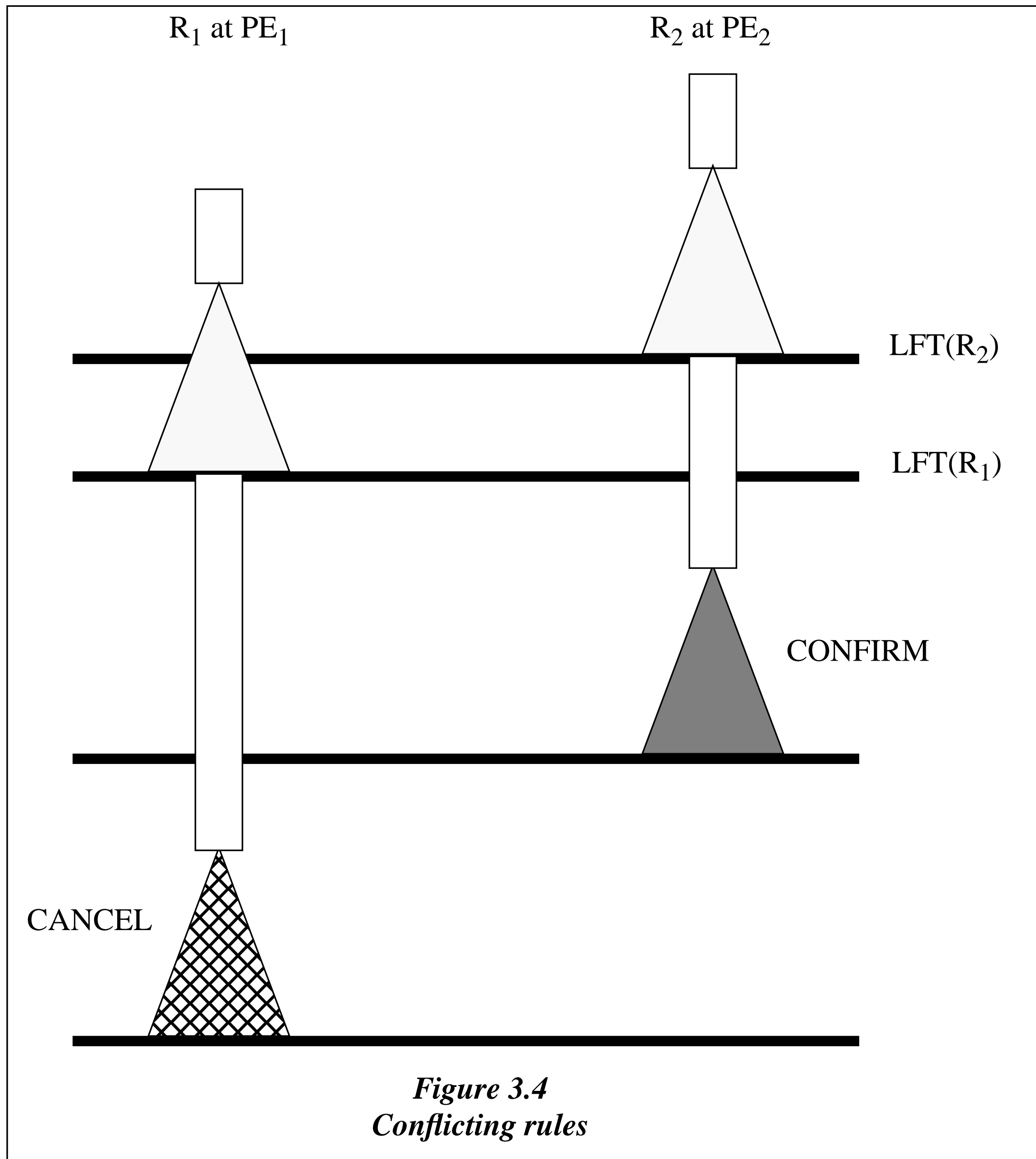


Figure 3.4
Conflicting rules

Any SCHEDs arriving *before* the LFT of a rule instantiation RI_1 are taken into account in the reevaluation of RI_1 at its LFT. That is, RI_1 cannot be reevaluated until these

SCHEDs have been resolved (either cancelled or confirmed). Any SCHEDs arriving *after* the LFT of RI_1 cannot make the firing of RI_1 incorrect because the rule firings represented by these SCHEDs all have LFTs *later* than that of RI_1, and hence these firings come *later* than RI_1 in the order of rule firings. It is the LFT alone that determines the firing order of rule instantiations. Since the LFTs are consistent across all processors, all processors agree on the same firing order.

3.2.6 Deadlock Freedom

A rule firing is **outstanding** if the rule process has issued the SCHEDs scheduling the firing, but has not yet sent out CANCELs or CONFIRMs for the firing. All outstanding rule firings are totally ordered by their LFTs, and all processors agree on the values of these LFTs. The rule firing with the lowest of these LFTs can always make progress. Hence the ISORULE system always progresses, and there is no possibility of deadlock.

3.2.7 Waiting Time

A rule firing RF_1 has a finite waiting time between its LFT and the time at which the decision is made to either cancel or confirm RF_1. In this waiting time, the processor is waiting for all SCHEDs with LFT *earlier* than that of RF_1 to be cancelled or confirmed. In the worst case, RF_1 may have to wait for all other outstanding rule firings at all PEs to be cancelled or confirmed before a decision can be made to cancel or confirm RF_1. In other words, instantiation RF_1 potentially conflicts with instantiation RF_2 which potentially conflicts with instantiation RF_3, and so on, until instantiation RF_{n-1} conflicts with instantiation RF_n, where *n* is the maximum number of outstanding rule firings across all proces-

sors. If there are p processors in the system, and a maximum of o outstanding rule firings are allowed per processor, then the maximum number of outstanding rule firings across all processors is $n = p \times o$

If the average evaluation time per rule firing is e_t, and l_t is the average latency time of the network to deliver a SCHED/CANCEL/CONFIRM corresponding to a single rule firing, then this bound on the waiting time is:

$$\text{Worst-case waiting time} = n \times (e_t + l_t)$$

physical time units.

3.3 Parallelism

Previous attempts to speed up rule-based systems by parallelization have had limited success. In the parallel-match MRA approach, only one rule is fired per cycle. This rule firing will affect very few rules owing to the small cycle problem. This means that most processors will be idle a significant portion of the time. Trying to speed up rule-based systems by using node parallelism in the Rete network increases synchronization overhead. On the other hand, trying to speed them up by firing multiple rules in the Act phase increases the length of the Resolve phase of the cycle because a set of non-conflicting rules has to be selected for firing.

Production systems based on the MRA cycle suffer from the effects of the small cycle problem because only one rule fires every cycle, and very few rules are affected by this firing, leaving a large percentage of the processors idle in the match phase. In ISORULE, no eligible rule is blocked by artificial contrivances of the algorithm like firing only one rule even though several may be eligible to fire. Hence, ISORULE diminishes the

effect of the small cycle problem. The coarser grain of parallelism (rule-level rather than node-level) avoids the increased synchronization overhead problem inherent in node-level parallel systems. At the same time, pipelining allows more parallelism than simple rule-level parallelism would. Finally, even though multiple rules are fired, no analysis to determine a set of non-conflicting rules is required. The only parallelizing overheads in the ISORULE system are the cost of cancelling a scheduled firing due to detection of conflict at the logical firing time, and the network latency cost of enforcing the isotach invariant. Thus we expect the ISORULE system to exploit all or most of the parallelism available in a rule-based application. Parallelism in ISORULE is exploited in two ways:

- **Parallelism across processors:** The process of performing match is divided among the PEs, and is done independently, without any need for synchronization. Each PE also schedules rules for firing asynchronously, whenever they are eligible. This provides one facet of the parallelism exhibited by ISORULE.
- **Parallelism due to pipelining:** The other facet of parallelism in the ISORULE system is obtained by the fact that each PE can have more than one outstanding scheduled firing at a time. That is, a PE does not have to wait until a firing is completed before it schedules a new firing.

The overall improvement in performance for ISORULE is a multiplicative result of the speed-ups gained by these two facets of parallelism. Figure 3.5 shows these two aspects of parallelism. Each processor pipelines rule firings. The first rule firing by P2 is cancelled on reevaluation; all other firings are confirmed. In general, we expect CANCELling to be infrequent as compared to CONFIRMing because of the observed fact that each working memory change affects very few rules [GUPT86], and so there is very little conflict among

rules.

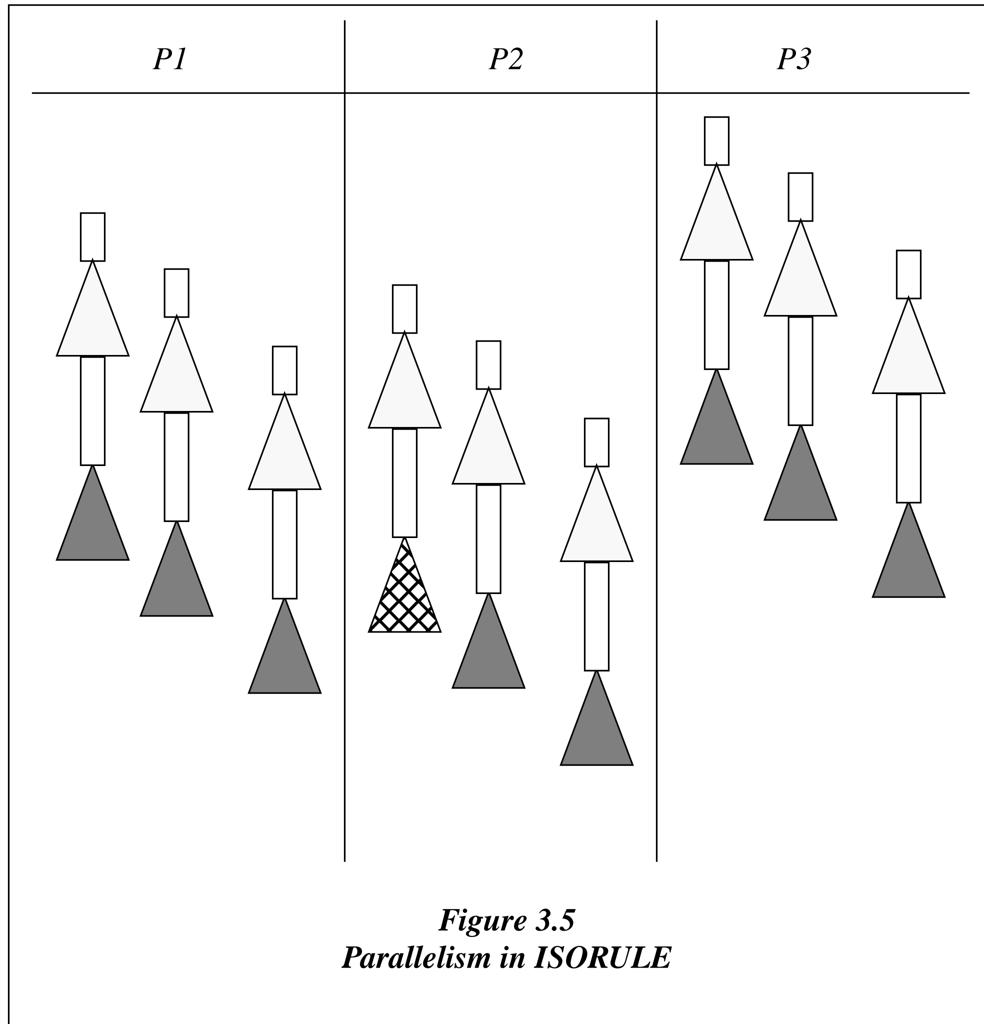


Figure 3.5
Parallelism in ISORULE

3.4 Chapter Summary

Our parallel rule-based system algorithm (ISORULE) eliminates the MRA cycle, and synchronizes among processors without using locks. Rules fire asynchronously and can be scheduled whenever they are eligible. ISORULE is based on isotach networks.

Isotach networks support atomicity which is essential to rule-based systems since each rule firing is one atomic action. Isotach networks are based on isotach logical time and maintain the *isotach invariant*. A PE can control the logical time of receipt of any

message it sends in an isotach system, and can hence execute atomic actions. Flat as well as structured atomic actions can be executed by a PE in an isotach system. A rule is a structured atomic action, and is hence executed with the help of a variant of *split operations* in the ISORULE system. A simulation study has shown that confronted with a workload imposing atomicity constraints, isotach networks outperform conventional networks by up to an order of magnitude [REYN92].

In ISORULE, the rule set is partitioned among the PEs, and a PE uses a SCHED multicast to schedule a rule firing. After *delta* pulses, when the SCHED has been received at all its destinations (Logical Firing Time LFT), the scheduling PE reevaluates the rule instantiation, and either CANCELs it (if the instantiation is no longer valid) or CONFIRMs it. ISORULE is deadlock-free, there is always progress, and the waiting time between the LFT and the time of sending out a CANCEL or a CONFIRM is bounded. The only parallelizing overheads in ISORULE are the cost of cancellation and the network latency due to enforcing the isotach invariant. The final improvement in performance of ISORULE is the product of the parallelism across processors, and the parallelism due to pipelining.

Chapter 4
Implementation

ISORULE was implemented as three co-executing processes at each processor, relying on the isotach network to maintain isotach logical time. This chapter describes the functions performed and data structures used by the three processes, and presents some optimizations that were implemented to enhance the performance of the ISORULE system. A conventional parallel rule-based system paradigm based on the MRA cycle (PARAMRA) was also implemented, so that its performance could be compared to that of ISORULE.

4.1 Review of the Basic ISORULE Algorithm

We restate our assumptions. The PEs form an equidistant network of *delta* stages: each PE is the same distance *delta* from each other PE. The rule set is static; that is, addition, deletion and modification of the rules in the production memory are not allowed. Recall from chapter 3 that these two assumptions are not necessary. The other assumptions are that there are no metarules, and that any serializable execution is correct.

The ISORULE algorithm partitions the set of rules among the PEs so that the rules can be processed in parallel. A process schedules a rule firing by sending out a multicast of SCHEDs to all of the PEs in the reader mask for that rule. It reevaluates the rule instantiation exactly *delta* pulses later to see if it is still valid at the LFT. The reevaluation includes all changes to working memory made or scheduled to be made up to LFT. If the instantiation is still valid, the PE confirms the rule firing by multicasting CONFIRMs to the PEs in the reader mask for the rule, else it cancels the firing by sending them CANCEL

multicasts.

4.2 Components of ISORULE

The tasks that need to be performed at any PE in an ISORULE system can be divided into three concurrently executable sub-tasks which communicate through buffers. The first of these sub-tasks comprises matching rules against WMEs, scheduling rule firings and reevaluating the firings at LFT. These are the functions that are more closely related to the rule set itself, and we assign a **rule process** to perform them. The second sub-task involves interfacing with the isotach network and multicasting and receiving messages, and we call the process handling this the **SIU process**. The third of the sub-tasks requires presentation of WME changes to the rule process in the correct order, and we call the process assigned to this the **WME process**. Messages are sent through an **isotach network** that maintains isotach logical time across all the PEs.

Ideally, each of the three processes would run on a separate co-processor. The interaction among the three processes is shown in figure 4.1. The **write buffer**, the **operation buffer** and the **token buffer** provide the communication among the three processes.

4.2.1 Rule Process

We assume that each process can have a maximum number of outstanding rule firings at any given time. A large number of outstanding instantiations at a processor means more pipeline parallelism. But it also means more potential for cancellation, and for buffer overflow.

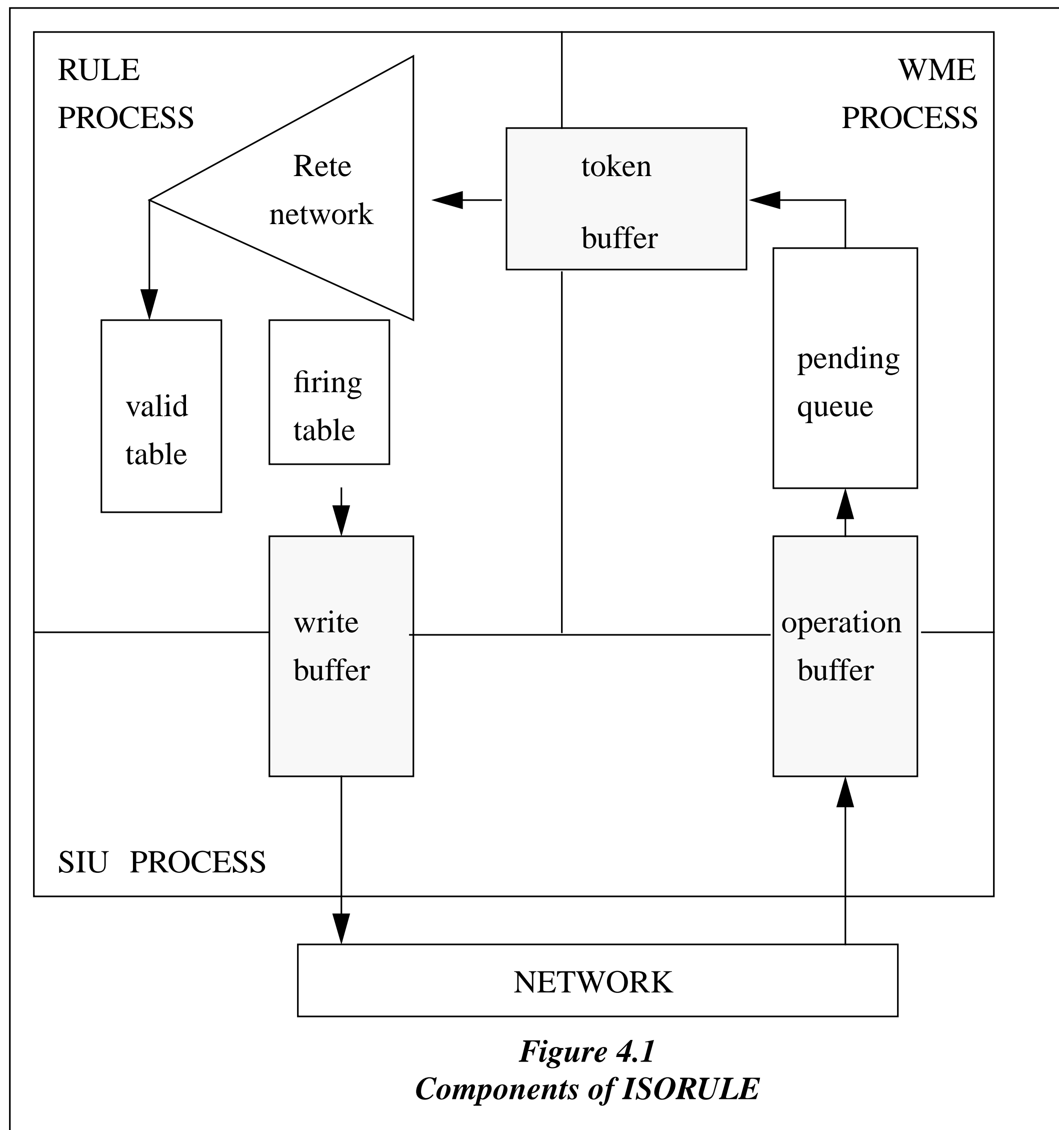


Figure 4.1
Components of ISORULE

The rule process maintains the data structures shown in figure 4.2:

- *rete*: the Rete network for the set of rules in this PE's partition;
- *valid table*: the set of valid instantiations output by the Rete network;
- *firing table*: a table of outstanding rule firings with a limit on the number of entries.

The rule process is responsible for scheduling rule firings. If the rule process has fewer outstanding rule firings than the maximum allowed, it tries to schedule a new firing.

If any new non-outstanding instantiations exist in the valid table, it picks one. The criterion for this selection could be anything, and meta-rules could be applied here. The rule process then allocates an entry in the firing table for this selected instantiation, marking it valid. The index of this entry in the firing table, called the *firing number*, uniquely identifies this instantiation among the outstanding instantiations at this PE. The rule process records the firing number in the valid table entry and marks this entry outstanding. It then enqueues a SCHED in its write buffer for this instantiation.

The rule process also handles tokens passed to it by the WME process. Each token can be:

- a **WME token**, representing a change to working memory resulting from a successful rule firing; or
- a **firing token**, signalling the LFT (time to reevaluate an outstanding instantiation).

When the rule process finds a WME token at the head of the token buffer, it pushes the token through the Rete network. If the change to working memory caused by this token results in the invalidation of an outstanding instantiation, the process cancels the firing of the instantiation and marks the corresponding entry in the firing table invalid so that when the firing token for the instantiation comes in, the process will know that the firing has already been cancelled.

When the rule process finds a firing token at the head of the token buffer (which signals the LFT for the rule firing represented by that token), the rule process determines if the instantiation is still valid by checking the invalidation status of the corresponding entry in the firing table. If the entry in the firing table indicates that the firing is still valid, the rule process enqueues a message confirming the firing. Otherwise, the instantiation has

already been cancelled.

The main algorithm followed by the rule process is shown in figure 4.2.

```
Data structures:

rete
valid table
      is_outstanding;  /* TRUE if this instantiation is outstanding */
      firing_number;   /* index into firing table if is_outstanding is TRUE */
firing table           /* indexed by firing_number */
      inst;            /* pointer to rule instantiation */
      invalidated;     /* TRUE if this instantiation has become invalid */

Main algorithm:

initialize_rete();
repeat until end of program
      if token buffer is not empty
         remove token from head of token buffer;
         if token is a WME token
            rete_push(token);              /* push WME token through rete */
         else                            /* it's a firing token signalling LFT */
            reevaluate_firing(token);      /* LFT: reevaluate the instantiation */
      else if firing table is not full   /* below limit max_outstanding */
         inst = select from valid_table; /* choose instantiation that isn't outstanding */
         if inst exists
            schedule_firing(inst);         /* schedule instantiation to be fired */
```

Figure 4.2
Rule process algorithm

The sub-procedures invoked in the main algorithm are shown in figure 4.3.

```
initialize_rete()
      push initial WME tokens through rete, updating valid table;

rete_push(WME token)
      push WME token through rete, updating valid table;
      for each outstanding instantiation invalidated in valid table
         enqueue CANCEL in write buffer;           /* cancel firing */
```

```
        set firing table[].invalidated to TRUE; /* mark instantiation invalid */


reevaluate_firing(firing token)
      get firing number f of the instantiation from firing token;
      if firing table[f].invalidated is FALSE
         enqueue CONFIRM in write buffer;        /* confirm firing */
      remove entry f from firing table;


schedule_firing(inst)
      f = allocate entry for inst in firing table;
      firing table[f].invalidated = FALSE;
      enqueue SCHED in write_buffer;           /* schedule inst */
      set
         firing_number = f and
         is_outstanding = TRUE
      in valid table, for the entry for inst;
```

Figure 4.3
Rule process algorithm: sub-procedures

4.2.2 SIU Process

The SIU process at each PE holds a copy of the reader mask for each rule assigned to that PE. Recall that the reader mask tells which PEs must be notified when that rule is scheduled, cancelled or confirmed. When the rule process issues an operation (SCHED, CANCEL or CONFIRM), the SIU process looks up the reader mask for the corresponding rule, and multicasts the operation to all the PEs in the reader set. Also, when a SCHED for a rule firing is multicast in pulse p, the SIU process generates a firing token for that firing exactly *delta* pulses after p. That is, the SIU process generates the firing token for a rule firing at the LFT for that firing, and sends it to the WME process. The order in which the SCHED and the firing token are delivered to the WME process is crucial. The SCHED has to be appended **after** the firing token because if the order is reversed, deadlock is immedi-

ate: the firing token cannot get to the rule process because there is an unsubstantiated SCHED blocking it, and the only way the SCHED can be substantiated is if the rule process processes the firing token and either cancels or confirms the firing. The process by which a firing token reaches the rule process is explained in Section 4.2.3.

The SIU process also receives SCHED, CANCEL and CONFIRM messages from the network, and passes them on to the WME process. The algorithm for the SIU process is shown in figure 4.4.

```
/* incoming messages */
siu_in_process(in_operation)
      append in_operation to operation_buffer;


/* outgoing messages */
siu_out_process()
      get operation from head of write_buffer;
      create destination list for message from reader_mask;
      build multicast message and pass it to the network;
      /* generation of firing token */
      if operation was a SCHED
         after delta pulses
            generate firing token
            append firing token to operation_buffer before SCHED
```

Figure 4.4
SIU process algorithm

4.2.3 WME Process

It is the WME process' job to match incoming CANCEL and CONFIRM operations with the corresponding SCHEDs. The WME process stores an incoming SCHED in a FIFO queue of pending changes. When a CANCEL or a CONFIRM arrives, it searches the pending queue for the corresponding SCHED. (The corresponding SCHED is identi-

fied by the rule number and the firing number.) When the corresponding SCHED is found, either both operations are deleted (in the case of a CANCEL), or the SCHED is replaced with the WME tokens representing the rule firing (in the case of a CONFIRM).

The other responsibility of the WME process is to deliver WME tokens to the rule process in the order in which the changes were scheduled. This order is not necessarily the same as the order in which the CONFIRMs arrive. An incoming firing token is appended to the pending list. The first item on the pending list can be delivered to the rule process if and only if it is a **token** — either a firing token or a WME token. Matching a CANCEL or CONFIRM with a SCHED may enable delivery of one or more tokens. Each time a SCHED is matched with an incoming CANCEL or CONFIRM, the WME process moves all the tokens, if any, from the head of the pending list to the token buffer.

A firing token is moved from the pending list to the token buffer only after all of the SCHEDs received before it are either cancelled or confirmed. Therefore, the rule process sees a WME token if and only if the LFT of the rule firing represented by the WME token is **earlier than** the LFT of the local rule firing represented by the firing token.

The algorithm for the WME process is shown in figure 4.5.

```
remove operation from operation buffer;

if operation is a firing token or a sched, append it to pending list;
else  /* operation is a CANCEL or a CONFIRM */
      search pending list for corresponding SCHED;
      assert: corresponding SCHED will be found;
      if operation is a CANCEL
         delete operation and corresponding SCHED;
      else  /* operation is a CONFIRM */
         turn SCHED into WME tokens;
         leave in pending list at spot where corresponding operation
           was found;
```

```
while first item in pending list is a firing token or a WME token
      move item to token buffer;
```

Figure 4.5
WME process algorithm

4.2.4 Isotach Network

The isotach network accepts the messages given to it by the SIU process, and routes them to the proper destinations, handing them to the SIU processes of the receiving PEs. The network maintains isotach logical time while doing so. The individual messages in a multicast are delivered to their destinations in the same logical time pulse. Our implementation included an isotach network simulation [DESU94].

4.3 Optimizations

4.3.1 CANCELs and CONFIRMs

SCHEDs need to arrive at all their destinations in the same logical time pulse, and therefore SCHEDs reserve a consistent time slice across all processors for a rule firing. On the other hand, the time of arrival of CANCELs/CONFIRMs need not be the same at all destinations. In fact, the sooner they arrive the better, because CANCELs and CONFIRMs eliminate SCHEDs from the pending queue, allowing tokens to progress towards the rule process. For this reason, CANCELs and CONFIRMs destined for the sending PE itself (let us call them *self-CANCELs* and *self-CONFIRMs*), are directly placed in the operation buffer instead of being sent out into the network. Since a self-CANCEL can arrive at the operation buffer before the corresponding SCHED, the WME process algorithm needs to be modified so that the WME process retains a self-CANCEL until the corresponding

SCHED arrives.

4.3.2 Speeding Up the Progress of a Firing Token

A firing token may be held up in the pending list by operations which are ahead of it but which do not affect it, thus delaying the firing of the local rule represented by that firing token. Since it is desirable to fire rules as quickly as possible, we work a firing token forward through the operations ahead of it in the pending queue. Whenever an incoming firing token is appended to the pending list, the WME process tries to work it forward in the list by checking if the operation ahead of it potentially invalidates it. If there is no such invalidation, the firing token is moved ahead of the operation. If the operation *does* invalidate the firing token, the firing token cannot be moved any further, and it is left at that position in the pending list. Further, if the operation that was found to invalidate the firing token is a WME token (a confirmed rule firing), then the firing token is marked invalid, and moved to the head of the token buffer. On seeing this invalidated firing token, the rule process immediately knows that a CANCEL has to be sent out.

4.3.3 Throttling the Rule Process

If the rule process keeps emitting SCHEDs for new rules quickly, the buffers at the PEs receiving these SCHEDs may fill faster than they can empty, leading to an unstable condition. While this unstable condition can be remedied to some extent by keeping the number of outstanding instantiations low, a finer degree of control is achieved by throttling.

When the SIU process at a PE finds that the length of the token buffer at that PE, or the combined lengths of the operation buffer and pending list (also at that PE) has

exceeded some preset limit, it sends a THROTTLE message to all of the PEs that can send messages to it. When the SIU process receives a THROTTLE message, it registers the message by incrementing the throttle count associated with all rules writing to the PE that sent it the THROTTLE by one. Now when the rule process tries to schedule a rule instantiation, it checks the throttle count associated with that rule to see if any of the destination PEs for that rule have sent it THROTTLE messages. If the throttle count is greater than zero, that rule firing is not scheduled, and the rule process attempts to pick a different rule to schedule.

When the buffer lengths at a PE which has sent a THROTTLE go below a preset level, it is ready to receive new SCHEDs again, and so multicasts an UNTHROTTLE message. A PE receiving an UNTHROTTLE message decrements the throttle count associated with all rules writing to the PE that sent the UNTHROTTLE by one. When the throttle count for a rule goes down to zero, the PE knows that no PEs have an objection to that rule being scheduled for firing.

4.4 A Conventional Parallel Paradigm

A popular parallel paradigm used in the execution of rule-based systems is what we will call the **PARAMRA** paradigm. PARAMRA is based on the MRA cycle, but the *match* phase of the cycle is performed in parallel by all the processors. Hence it is a parallel-match MRA paradigm.

In the PARAMRA system, one processor is designated to perform conflict resolution. (Let us call this the "resolve-PE".) The set of rules is divided among all the other PEs (let us call these the "match-PEs"). The match-PEs perform the match on the rules assigned to them in the *match* phase. In the *resolve* phase, each of the match-PEs arbi-

trarily selects one valid instantiation from its pool as a candidate for firing, and sends it to the resolve-PE. If a match-PE cannot find a valid instantiation, it sends a *no_instantiation* message to the resolve-PE. When the resolve-PE has received messages from all the match-PEs, it arbitrarily selects one out of the candidate instantiations, and confirms that it is to be fired by sending it to all the match-PEs. If there are no candidates, it means that there no more firable rules in the entire system, and the program is terminated.

When a match-PE receives a confirmed rule firing from the resolve-PE, it fires the rule and updates its local copy of the working memory. This completes its *act* phase, and it now goes back to the *match* phase.

4.5 A Comparison of ISORULE to PARAMRA

Clearly, the synchronous nature of the PARAMRA paradigm prevents it from taking advantage of all of the parallelism in the rule set. The only parallelism PARAMRA avails of is *match parallelism*. *Match* **is** the most expensive phase of the MRA cycle and it makes sense to try to speed it up. But as *match* becomes cheaper and cheaper, the other phases of the MRA cycle become more and more significant. Even though there may be several rules eligible to fire in the *act* phase, the PARAMRA system fires only *one*.

ISORULE, on the other hand, fires rules asynchronously whenever they are eligible to fire. So if, at any instant, there are N rule instantiations eligible to fire, PARAMRA can fire only one, while ISORULE can fire all N concurrently, provided there are enough processors available to it. This analysis does not take into account the increased network latency of the isotach network (introduced because of the cost of maintaining isotach logical time), and the cost of rule cancellations in the ISORULE system. These factors will reduce the performance gain of ISORULE over PARAMRA, but we do not expect this

reduction to be high enough to cancel out the advantages offered by ISORULE over PARAMRA. We expect the number of cancellations in ISORULE to be low because rule-based system literature says the amount of conflict among rules in a rule-based system is very low [GUPT89]. Hence, we expect to see the ISORULE system outperform PARAMRA by taking advantage of more of the parallelism offered by a rule set.

Another advantage of ISORULE is its completely distributed control. The resolve-PE in the PARAMRA system is a bottleneck: message traffic to and from the resolve-PE increases as the number of processors is increased. ISORULE has no such bottleneck, and is therefore more scalable than PARAMRA.

4.6 A Comparison of ISORULE to Other Asynchronous Techniques

Asynchronous algorithms [SCHM92] which use either offline analysis (Section 2.3.1) or locks (Section 2.3.2) were described earlier. In contrast to these other asynchronous approaches, ISORULE does not require any kind of offline analysis to determine which rules to fire. Besides, there is no central scheduler process which might become a bottleneck, nor is there contention for a central resource like a shared queue. Moreover, overhead associated with acquiring and releasing locks is not present in ISORULE.

PARS [SCHM88, SCHM90] was described in Section 2.3.3 as the final asynchronous algorithm for rule-based system execution. Processors in PARS use *enable* and *disable* messages to lock out conflicting rules from executing, and have to wait for acknowledgments. Besides, the system incurs the overhead of having to deal with deadlock. ISORULE involves no enable/disable messages, waiting for acknowledgments, or deadlock detection/recovery. However, ISORULE does bear the cost of cancellations of

rule firings, and the cost of maintaining isotach logical time.

4.7 Chapter Summary

The rule set in the ISORULE system is partitioned among the PEs, and the WME classes read by each PE are stored in that PE's local memory. The reader and writer sets and the reader mask are calculated and stored. A process schedules a rule firing by sending out a multicast of SCHEDs to all the PEs in the reader mask for that rule. It reevaluates the rule instantiation exactly *delta* pulses later to see if it is still valid at the LFT. If the instantiation is still valid, the PE confirms the rule firing by multicasting CONFIRMs to the PEs in the reader mask for the rule, else it cancels the firing by sending them CANCEL multicasts. The ISORULE system is composed of four components:

- a **rule process** at each PE, that executes the rule-based program;
- a **WME process** at each PE, that ensures that the rule process gets the correct view of its read set;
- an **SIU process** at each PE, that interfaces between the rule process and the network on one hand, and between the network and the WME process on the other hand;
- an **isotach network**, that maintains isotach logical time.

Optimizations can be carried out to speed up a rule firing.

We chose to implement a popular parallel rule-based system paradigm (PARAMRA) which parallelizes the *match* phase of the MRA cycle. Only one rule instantiation is fired in a single cycle in PARAMRA, thus losing out on a lot of the available parallelism in the rule set. Since ISORULE fires rules asynchronously and whenever they are eligible to fire, we expect ISORULE to outperform PARAMRA. The cost in network

latency due to maintaining isotach logical time, as well as the cost of cancellation of rule firings in the ISORULE system will reduce the performance gain of ISORULE, but we do not expect these factors to outweigh the advantages ISORULE offers over PARAMRA.

Chapter 5
Performance Analysis

To recapitulate, ISORULE fires rules asynchronously whenever they are eligible to fire. If there are *N* rule instantiations eligible to fire, PARAMRA can fire only one, while ISORULE can fire all *N* concurrently, provided there are enough processors available to do so. Actually, the improvement is not quite *N* because of the cost of maintaining isotach logical time and the cost of cancellation of rule firings in ISORULE. Since cancellations are generally few due to very little conflict typically occurring in a rule set, we expect the performance improvement of ISORULE over PARAMRA to come very close to *N*.

Our efforts and contributions are outlined below and elaborated in the rest of this chapter:

- A significant amount of effort was put into a simulation of the ISORULE and the PARAMRA systems. Section 5.1 describes the simulation in detail, including the generation of synthesized rule sets, and the integration of a real Rete network algorithm and a parser for real rule sets.
- We devised a static dependency model to analyze the effects of rules in a rule set on one another, and to predict ISORULE performance based on this analysis. The design and implementation of this model is detailed in Section 5.2.
- We computed performance predictions on synthesized rule sets using our static model, simulated the rule sets, and compared our predictions with actual results. The experiments we performed, and the results we obtained are described in Section 5.3.
- We investigated the effects of pipelining and the number of processors on ISORULE

performance for synthesized rule sets; Section 5.3 reports on this investigation.

- We evaluated the performance of real rule-based programs written in OPS5 and executed using ISORULE and PARAMRA. We discuss the performance of these real rule-based programs in Section 5.4.
- Finally, we analyzed the performance results we obtained, and listed characteristics of rule sets that would allow ISORULE to exploit a large portion of the available parallelism.

The results of our performance study are listed below:

- Experiments with synthesized rule sets revealed that ISORULE outperforms PARAMRA by up to an order of magnitude. Our initial analysis suggests that the performance improvement increases with the amount of concurrency in the rule sets, and hence multiple orders of magnitude in performance improvement seem likely with larger rule sets exhibiting a low degree of conflict.
- We expected ISORULE to perform better with more pipelining up to a point, and then decline due to increased cancellations, messages and throttling. Tests confirmed this trend in the behaviour of ISORULE.
- Experiments on up to 32 processors revealed that ISORULE performance improved with an increase in the number of processors, up to the point where the number of processors equalled the number of rules.
- Tests with actual OPS5 rule sets did not yield the spectacular results obtained from the synthesized rule sets. We attribute the limited speed-ups to the limited potential parallelism of the OPS5 rule sets. ISORULE still exploited most of the limited parallelism offered by the OPS5 rule sets.

5.1 The Simulation

We simulated the ISORULE system on an isotach equidistant network, and the PARAMRA system on a conventional equidistant network. As discussed in chapter 3, the assumption of an *equidistant* network can be removed with slight changes to the ISORULE algorithm. A simulator for the underlying networks was provided by Bronis R. de Supinski [DESU94]. It was important to simulate the networks to such detail because we wanted to include the cost of maintaining isotach logical time in the ISORULE system.

The simulation proceeds in units of "*simulation cycles*", where one *simulation cycle* models the minimum time it takes for a network switch to move an item from one buffer to another. Hence simulation cycles represent the same *real-time* in both the ISORULE and the PARAMRA systems. All other activities in the simulation are assigned times that are scaled units of simulation cycles. If the same task is performed at a PE in both ISORULE and PARAMRA, the task is allotted the same number of simulation cycles in both systems.

5.1.1 Workloads

We need to provide our simulation with a workload comprising a working memory and a set of rules. We use two kinds of workloads. The first is a synthetic workload which was generated based on certain carefully selected parameters, and the second is a real workload which consists of a test suite of rule-based programs written in OPS5.

5.1.2 Synthetic Workload

The synthetic workload consists of a set of WME classes, and a set of rules referencing and modifying the members of these classes. The parameters used to generate this

workload are:

- the total number of WME classes,
- the total number of rules,
- a range for the number of condition elements in a rule,
- a range for the number of action elements in a rule, and
- parameters that control "hot" WME classes (classes that are accessed by a large number of rules).

Figure 5.1 shows an example of a synthesized rule set which contains seven WME classes. Rule 1 plus-references WME classes 3 and 4, and minus-references WME class 2. It also minus-modifies WME class 4 and plus-modifies WME class 2. In other words, rule 1 can fire if WMEs belonging to classes 3 and 4 exist, and there is no WME belonging to class 2. When rule 1 is fired, it adds a WME belonging to class2, and deletes a WME belonging to class 4.

```
WME classes:
   0, 1, 2, 3, 4, 5, 6

(rule 1
   (class 3)
-  (class 2)
   (class 4)
-->
-  (class 4)
   (class 2))

(rule 2
   (class 5)
   (class 2)
   (class 6)
-->
   (class 2))
```

```
(rule 3
   (class 0)
-  (class 1)
-->
-  (class 0))
```

Figure 5.1
An example synthesized rule set

We use a probabilistic model for the Rete algorithm which matches rules against WMEs. When a WME token is pushed through the Rete network, the Rete algorithm determines which of the existing rule instantiations have been invalidated by the WME token, and which new rule instantiations have been created by the WME token, and updates the set of instantiations accordingly. We model the pushing of WME tokens corresponding to the firing of a rule R_i through the Rete network in the following way:

- *Invalidation of existing instantiations:* Static analysis is first used to determine for every pair of rules R_i and R_j, whether R_i potentially invalidates R_j. This static analysis is performed by comparing the WME classes modified by R_i and those referenced by R_j. If static analysis determines that R_i *cannot* invalidate R_j, then firing R_i concurrently with R_j at run time does not affect any instantiation of R_j. On the other hand, if the static analysis reveals that R_i *can* invalidate R_j, an *invalidation probability* for rule R_j is used to decide for each instantiation of R_j if the instantiation remains valid. The invalidation probability is used to model the possible matching of *attributes* and *variable bindings* of the instantiations of R_i and R_j. For instance, static analysis with the synthesized rule set in figure 5.1 reveals that rule 2 potentially invalidates rule 1, since rule 2 adds a WME belonging to class 2, and a rule 1 firing depends on a WME belonging to class 2 *not* being present. However, the firing of a particular instantiation of rule 2 may *not actually invalidate* a particular instantiation

of rule 1, if the attributes and variable bindings of the WME modified and referenced by the two rule instantiations do not match at run-time.

- *Creation of new instantiations:* Static analysis is used to determine if R_i can enable R_j, for every pair of rules R_i and R_j. If R_i potentially enables R_j, a *validation probability* for R_j is used to decide if a new instantiation of R_i should be created.

The *invalidation* and *validation* probabilities are generated for each rule from a range specified by parameters to the simulation. The procedure just described simulates the result of a Rete match. Each rule starts out in the system with a certain number of firable instantiations, also generated from a range specified by parameters to the simulation. Note that an actual implementation of ISORULE would use an actual Rete network.

5.1.3 Real Workload

Four OPS5 programs were chosen to comprise the real workload: Monkey and Bananas, Tourney, Manners and Toru-Waltz. The first three of these programs solve toy problems. The rule sets are of different sizes and exhibit varying amounts of conflict among the rules. The parsing of the OPS5 programs, and the Rete match was performed by CParaOPS5 code [KALP88] written at Carnegie Mellon University. CParaOPS5 provides a parser which processes a rule-based program written in OPS5 and generates a C file containing the Rete network data for that program. We modifed the CParaOPS5 parser to generate multiple Rete networks, one for each processor, with the rules assigned to that processor. CParaOPS5 also provides a uniprocessor version of the code which reads the data from the C file generated by the parser, builds the Rete network, and executes the Rete match algorithm, updating the set of rule instantiations eligible for firing. We inte-

grated this part of the CParaOPS5 code with our simulation of ISORULE and PARAMRA. Finally, we integrated the portion of CParaOPS5 that modifies working memory in response to the execution of an action element of a rule.

5.1.4 Simulation Parameters

Apart from the parameters already mentioned, the simulation also uses the following parameters common to both ISORULE and PARAMRA:

- the number of PEs; and
- the costs, in simulation cycles, for the tasks performed by the rule process, WME process and SIU process. These are chosen based on the complexity of the tasks relative to one another.

In addition, these are the parameters that are specific to ISORULE:

- the thresholds for throttling and unthrottling [see Section 4.3.3];
- the maximum allowed length of a buffer (write buffer, operation buffer, token buffer, pending queue); and
- the maximum number of outstanding rule instantiations allowed at any time. (This parameter controls the degree of pipelining.)

All of the simulation parameters are elaborated in Appendix A.

5.2 A Static Model for ISORULE Performance

We devised a static dependency model to analyze the potential concurrency in a rule set, so that we could predict ISORULE performance. The model builds a data dependency graph, and uses a greedy algorithm to find an approximate solution to the NP-complete problem of finding a maximum independent set. The model is both conservative and

optimistic in its estimate of potential concurrency.

5.2.1 Dependence Analysis

If we can analyze the amount of parallelism inherent in a rule set, we can predict the performance of the ISORULE system on that rule set. To this end, we construct a **data dependence graph** representing the rule set. Interference analysis was discussed in Section 2.2.7. There, it was discussed as a method used by conventional multiple rule firing systems to determine a set of rule instantiations that can be safely fired in parallel. *We* use dependency analysis for the purpose of determining available parallelism in a rule set in order to enable prediction of ISORULE performance on that rule set. A set of rule instantiations can fire concurrently if the instantiations in the set do not interfere with one another. Such a set is called a set of **compatible** rule instantiations. A set of compatible rule instantiations can be selected from the conflict set by constructing a data dependence graph.

A data dependence graph is built by determining the data dependence between each pair of rule instantiations. Each node in this graph represents a rule instantiation, and an edge between node N_1 and node N_2 represents the data dependence relation between the rule instantiations represented by N_1 and N_2.

Let RI_i and RI_j be rule instantiations of the rules R_i and R_j respectively. Rule instantiation RI_i *inhibits* RI_j if the firing of RI_i changes working memory in a such a way that RI_j is no longer valid. Two example data dependence graphs are shown in figure 5.2. In figure 5.2-a, rule instantiation RI_1 removes the WME B that is referenced by rule instantiation RI_2, thus making RI_2 ineligible for firing. In figure 5.2-b, RI_1 inhibits RI_2

which inhibits RI_3 which inhibits RI_1; the three rule instantiations cannot fire simultaneously because there is no sequential result that is equivalent to the parallel result.

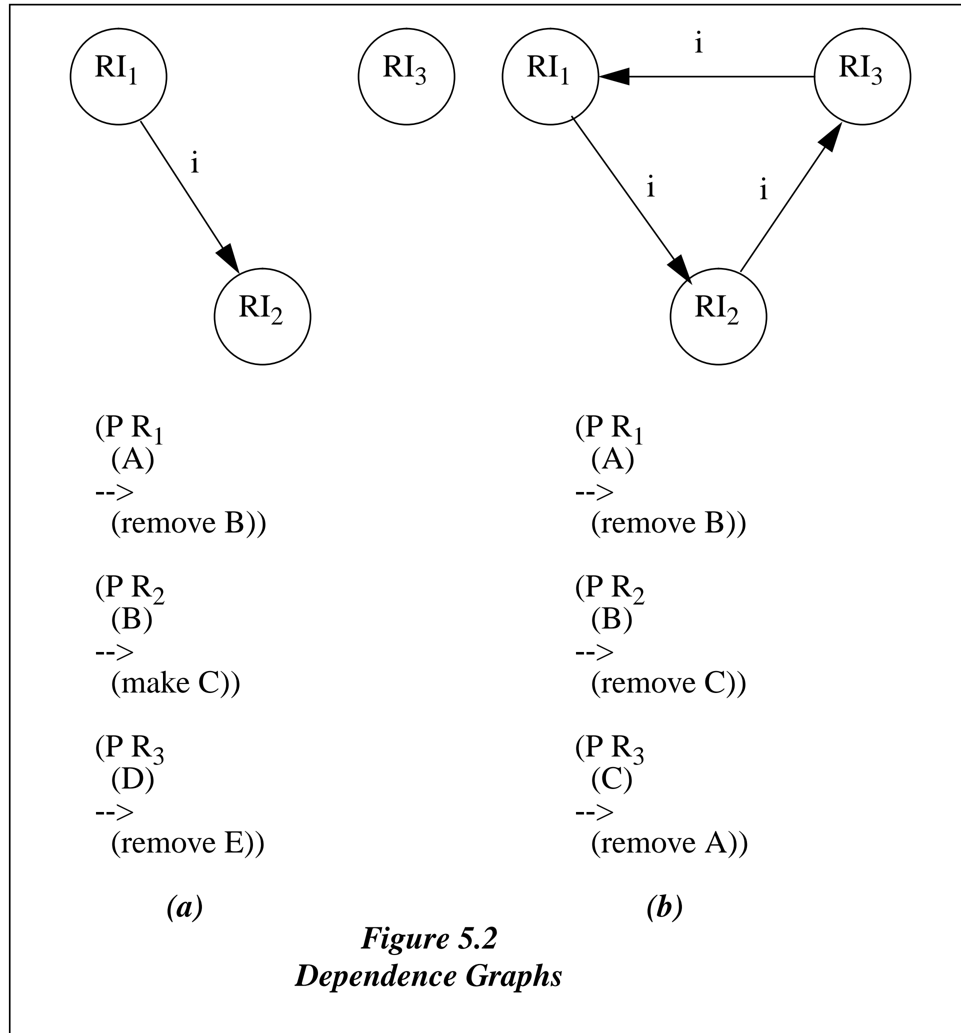


Figure 5.2
Dependence Graphs

Static analysis of the condition and action elements of all the rules can be used to build a data dependence graph for a rule set. While only partial data dependence analysis is possible with this approach (due to the presence of attributes and variables in production rules), it has the advantage that the analysis can be done off-line. At run time, all variables are bound, and it is possible to analyze the data dependences between rule *instantiations* instead of just between rules. But the highly computation-intensive nature of dynamic

analysis has led most researchers to propose a mixture of compile-time and run-time analyses [ISHI90, KUOS91, MIRA90].

5.2.2 A Static Model to Assess Parallelism

For each rule set, we build its dependency graph. The graph contains another type of edges in addition to inhibiting edges: *conflict* edges. If rule instantiation RI_i plus-references (minus-references) a WME, and rule instantiation RI_j minus-references (plus-references) the same WME, the two instantiations cannot be valid at the same time. We draw a *conflict edge* between the nodes representing rules R_i and R_j to indicate that instantiations of these two rules may not co-exist in the conflict set. Now, if we remove nodes (and the associated edges) from the data dependence graph such that we are left with the largest sub-graph in which there are no edges, we have the largest subset of rules that can occur in the conflict set such that no two rules in the subset conflict with each other. This gives us an estimate of the amount of rule parallelism there is in the rule set. Instantiation parallelism is, of course, not detected by this sort of static analysis.

The problem of finding the largest sub-graph with no inhibiting edges is a *maximum independent set problem*, which is known to be NP-complete [GARE79]. We use a greedy algorithm to arrive at an approximate solution to the problem. We start with the graph representing the entire rule set with inhibiting dependences, and remove one node (with all edges touching it) at a time until we end up with a graph that has no inhibiting edges. At each step, we choose the node to be removed by using the following greedy algorithm consisting of two rules:

- **Rule R1**: Remove the node with the largest number of inhibiting edges either

emanating from it or incident upon it. This removes the rule which is involved most in conflict. If there is more than one node satisfying this criterion, we use rule R2 to pick one of these candidate nodes.

- **Rule R2:** If R1 does not yield a unique node to remove, **randomly** select one of the candidates that are left, and remove it.

Figure 5.3 illustrates this process for an example rule set.

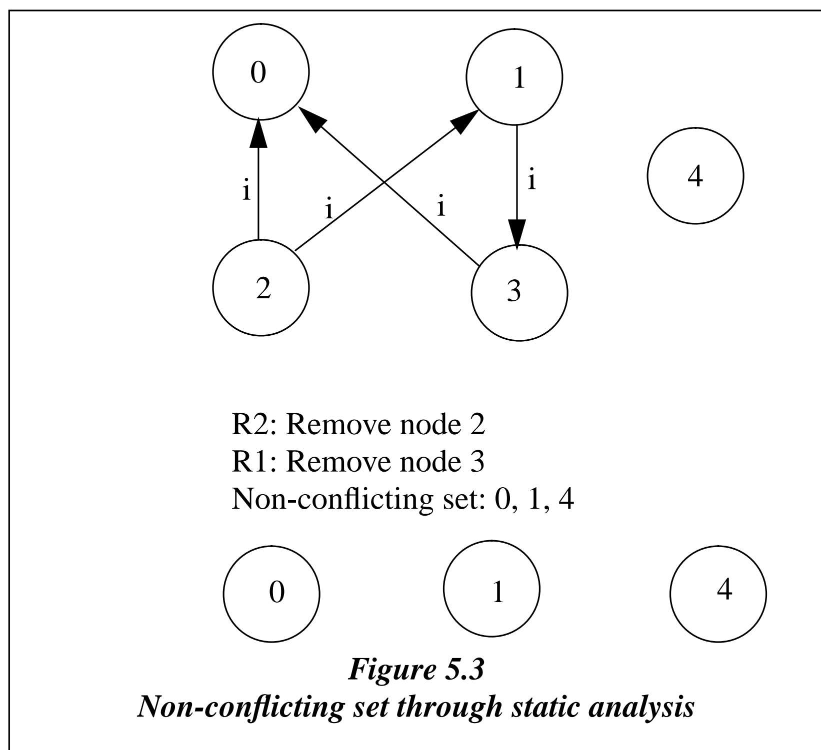


Figure 5.3
Non-conflicting set through static analysis

The estimate obtained from our static model may *understate* the amount of rule parallelism actually available, for any of the following reasons:

- We use a heuristic that may find a smaller-than-maximal non-conflicting set.
- The existence of an edge between the nodes representing two rules does not necessarily mean that their instantiations will actually conflict. A consideration of the attributes and variables may reveal that there is, in reality, no conflict. Our model

takes the conservative view and assumes that the two instantiations *will* conflict. We will discuss in Section 5.3.1 how this aspect of the understating problem can be eliminated from our analysis of ISORULE performance over synthesized rule sets.

- Favourable LFTs can enlarge the set of concurrently firable rules beyond the maximal non-conflicting set as defined above. For instance, in figure 5.3, if instantiations of rules 0 and 1 had LFTs earlier than the instantiation of rule 2, then rule 2 would not invalidate rules 0 and 1. Using *pairwise* instead of *cyclic* conditions (Section 2.2.7) is responsible for this aspect of the understating problem.

Our estimate of rule parallelism may also *overstate* the amount of rule parallelism available, for the following reason:

- Not all of the non-conflicting rules detected by our model may actually have instantiations simultaneously during execution of the rule set.

It would be advantageous if we could change our model so that it *only understates* or *only overstates* the amount of rule parallelism; we leave this task to future research.

5.3 Synthesized Rule Sets

We applied our static analysis model to synthesized rule sets and discovered that ISORULE exploits most of the statically determinable rule parallelism in a rule set. Our experiments revealed up to an order of magnitude performance improvement of ISORULE over PARAMRA.

5.3.1 Analysis

We varied the parameters controlling the characteristics of the synthetic rule sets, and obtained several rule sets of different sizes and exhibiting varying degrees of conflict.

We analyzed each of these synthesized rule sets statically with the approximation algorithm detailed earlier, producing an estimation of the maximum concurrency offered by each rule set. This allowed us to make a prediction of how much the ISORULE system would outperform the PARAMRA system, taking into account only rule parallelism and disallowing pipelining. If the static analysis revealed that the concurrency in the rule set (size of the largest set of non-conflicting rules) was N, this would mean that given this conflict set, the PARAMRA system would be able to fire only one rule, while the ISORULE system could fire all N rules (assuming there were enough processors to allot each rule to a different processor). We then ran the simulation with the rule set, placing the following constraints on the ISORULE system so that it matched our static dependence model more closely:

- We forced ISORULE to have at most one outstanding instantiation per rule at any time and disabled pipelining. Since our static dependence model does not detect instantiation parallelism, and does not consider multiple rules being executed simultaneously at a single processor, this constraint on ISORULE brings it closer to the model assumed by our static analysis.
- We set the *invalidation probability* to 1.0, which means that when static analysis has detected that a rule instantiation potentially invalidates another rule instantiation, we *actually invalidate* the second instantiation even though a consideration of attributes and variable bindings may reveal that the two instantiations do not actually conflict. Thus, the constrained version of ISORULE now matches the conservative analysis of interference by our dependence model, and eliminates the second form of *understatement* of the amount of rule parallelism by our static dependence model.

We call the constrained version of ISORULE ISO1, and the unconstrained version ISO2. We compared the number of rule firings obtained by the PARAMRA run and the ISO1 run over a selected number of simulation cycles, and checked this figure against the one obtained from static analysis. We then ran ISO2 over the same rule set for the same number of simulation cycles. We expected two results:

- If the result obtained from the ISO1 run is almost equal to the result predicted from static analysis, it means that the ISORULE system is exploiting most of the statically determinable parallelism in the rule set. However, remember that the figure obtained from static analysis may *understate* the amount of concurrency *actually* available. So it is possible for the actual improvement in performance displayed by ISORULE over PARAMRA to actually *exceed* the predicted improvement, as will be seen in the graphs 5.4-b, 5.5 and 5.6-b discussed in Section 5.3.2.
- Enabling pipelining should give the ISORULE system the benefit of a second kind of parallelism, and we would expect to see further improvement in its performance as compared to that of the PARAMRA system. Allowing multiple outstanding instantiations per rule will improve the performance of ISORULE if the rule-based program results in more than one instantiation being firable at a time, for one or more rules. In other words, we would expect the performance of *ISO2* to be better than that of *ISO1*.

5.3.2 Results from the Synthesized Rule Sets

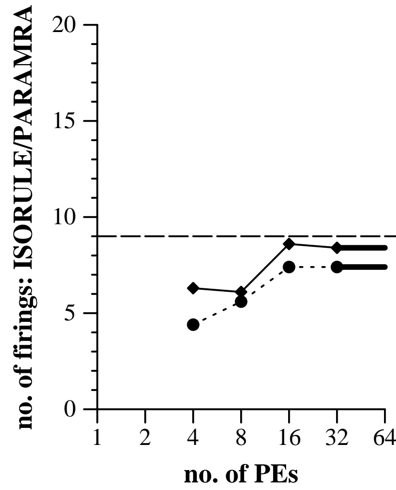
We took a **rule set size of 32**, varied the parameters controlling the rule set characteristics, and generated two groups of rule sets with the following characteristics: The first group exhibited *more conflict* and, on static analysis, yielded an average concurrent set

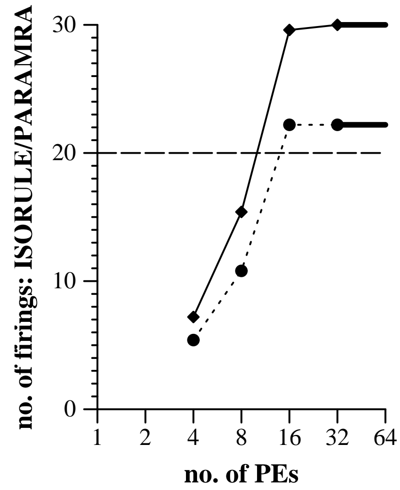
size which was *less than* half the size of the rule set (less than 16). The second group of rule sets exhibited *less conflict*, and static analysis for these rule sets yielded an average concurrent set size *greater than* half the size of the rule set (greater than 16).

We made several runs of ISORULE and PARAMRA on each of the rule sets in **group 1** for a fixed number of simulation cycles (long enough to stabilize the results), and calculated the average number of ISORULE firings per run, and the average number of PARAMRA firings per run, for 4, 8, 16 and 32 processors. We then *normalized* the number of ISORULE firings by the number of PARAMRA firings for the same number of runs. Figure 5.4-a shows the results from these runs. The dotted line is the ISO1 (Section 5.3.1) result normalized by the PARAMRA result. We compare ISO1 performance to the performance predicted by our model because the assumptions made in ISO1 are similar to the ones made in our static dependency model. The solid line represents the ISO2 (Section 5.3.1) result normalized by the PARAMRA result.

Figure 5.4-b shows the results from similar runs of the rule sets in **group 2**. Results for rule sets containing **16 rules** are shown in figure 5.5, and those for rule sets containing **8 rules** are in figure 5.6 Rule sets were not run with the number of processors *greater than* the number of rules because the extra processors would not be assigned any rules by either ISORULE or PARAMRA, and would remain idle throughout the execution. Each point on the graph represents the average over 24 runs.

.





CONCURRENT SET SIZE = 9

ISO1/PARAMRA (32 PEs) = 7.4

ISO2/PARAMRA (32 PEs) = 8.4

CONCURRENT SET SIZE = 20

ISO1/PARAMRA (32 PEs) = 22.2

ISO2/PARAMRA (32 PEs) = 29.9

(a) ***(b)***

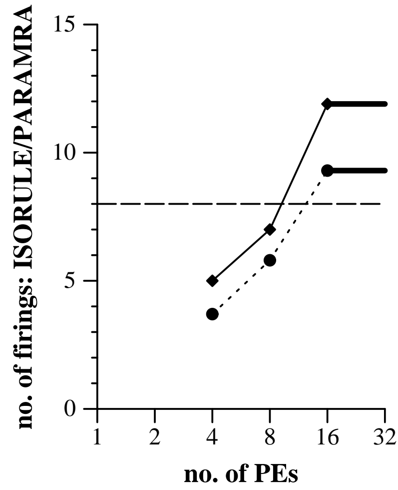
. . . . **: ISO1 / PARAMRA**

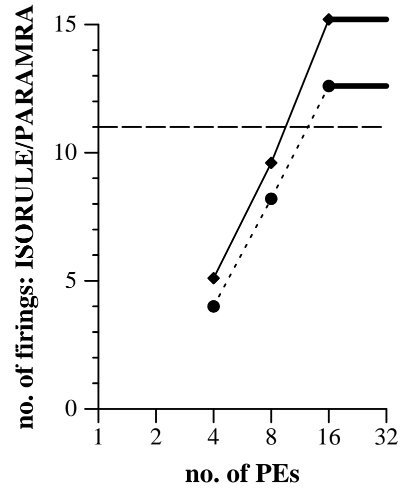
———— **: ISO2/PARAMRA**

— — — **: CONCURRENT SET SIZE FROM STATIC ANALYSIS**

━━━━ **: HYPOTHETICAL (NOT FROM EXPERIMENTS)**

Figure 5.4
Rule Sets with 32 Rules





CONCURRENT SET SIZE = 8

ISO1/PARAMRA (16 PEs) = 9.3

ISO2/PARAMRA (16 PEs) = 11.8

CONCURRENT SET SIZE = 11

ISO1/PARAMRA (16 PEs) = 12.6

ISO2/PARAMRA (16 PEs) = 15.1

: ISO1 / PARAMRA

: ISO2 / PARAMRA

: CONCURRENT SET SIZE FROM STATIC ANALYSIS

: HYPOTHETICAL (NOT FROM EXPERIMENTS)

(a) ***(b)***

Figure 5.5
Rule Sets with 16 Rules

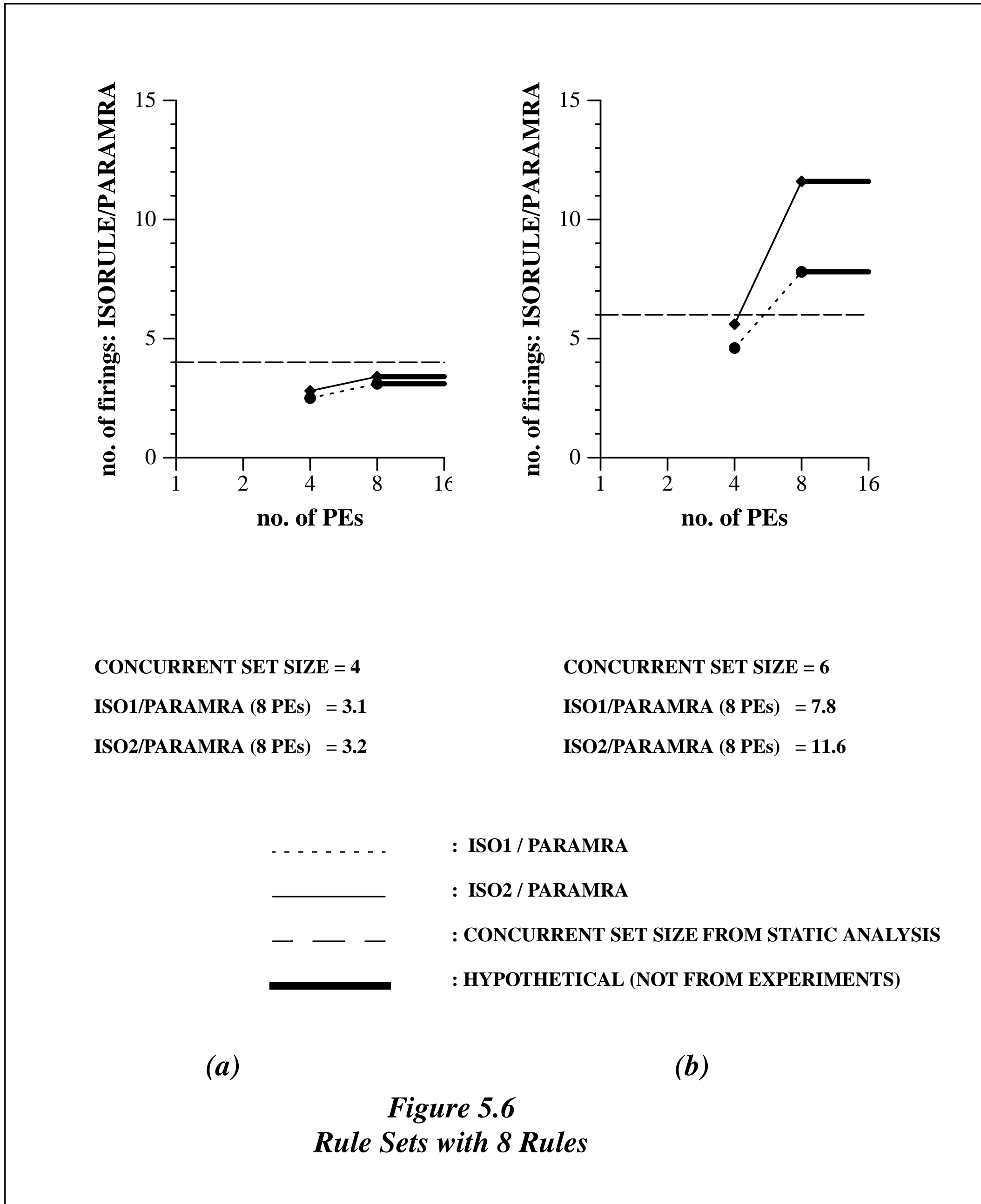


Figure 5.6
Rule Sets with 8 Rules

Figures 5.4, 5.5 and 5.6 show that the ISO1/PARAMRA results obtained (with the maximum number of processors) come close to the values predicted by our static dependence model. Hence the ISORULE system seems to exploit most of the statically determinable rule parallelism available in a rule set. ISO2 does better than ISO1, thus

confirming the second prediction we made in Section 5.3.1. All of the results for ISO2 (except the one for two rule sets with a lot of conflict: figures 5.4-a and 5.6-a) show an **order of magnitude** improvement in performance over PARAMRA.

5.3.3 Effect of Number of Processors on ISORULE Performance

We expect ISORULE performance to improve as we increase the number of processors used to execute a rule set. This improvement would stop once the number of processors became more than the number of rules. (For a system with *N* rules, if there were more than *N* processors, the extra processors would be idle throughout the execution, as neither ISORULE nor PARAMRA would assign any rules to them.) The improvement in ISORULE performance obtained by increasing the number of processors could be sub-linear due to reasons such as an increased percentage of cancellations of rule firings. On the other hand, PARAMRA performance could deteriorate with an increased number of processors because of increased contention for the resolve-PE.

We generated several rule sets with **16 rules**, executed several runs of each of them over 4, 8 and 16 processors, and averaged the results obtained. Figure 5.7-a shows the results from these runs. The graph in figure 5.7-b shows similar results for rule sets with **32 rules** executed over 4, 8, 16 and 32 processors. Pipelining was allowed for all cases, with a maximum pipe length of two. Each point on the graphs is an average over 24 runs.

We obtained results from our experiments showing the number of rule firings by ISORULE over a fixed number of simulation cycles. The results verified, for up to 32 processors, that ISORULE performance improves as the number of processors increases. We observed with the 32-rule cases that the percentage of cancellations increased from an average of 5% (with 4 processors), to an average of 21% (with 32 processors).

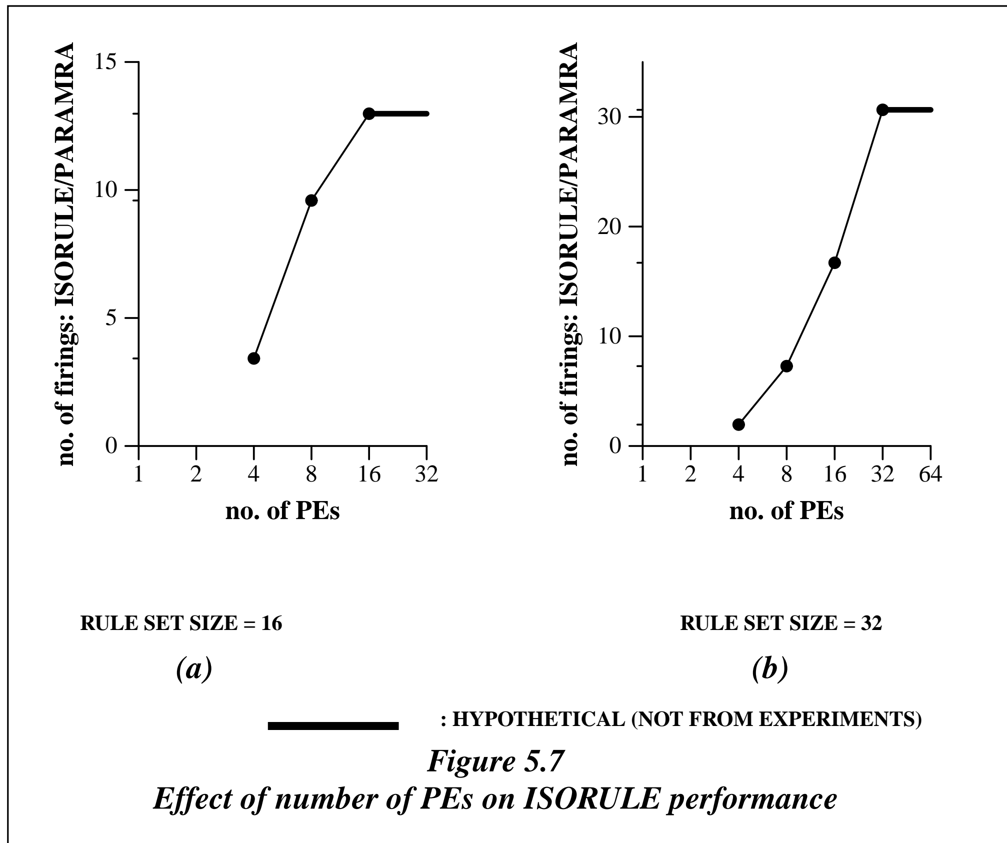


Figure 5.7
Effect of number of PEs on ISORULE performance

The effect of the deterioration in PARAMRA performance is very pronounced in figure 5.7-a, when the number of PEs goes from 4 to 8. This deterioration makes the $\frac{ISORULE}{PARAMRA}$ ratio improve by *more than* a factor of two.

5.3.4 Effect of Degree of Pipelining on ISORULE Performance

Increasing the amount of pipelining allowed (by increasing the maximum number of outstanding instantiations allowed) increases the number of concurrent firings if there are enough eligible instantiations to pipeline. A disadvantage of pipelining too many rule firings is that the percentage of firings **cancelled** can go up. The number of messages travelling through the network increases too. Another disadvantage is that the buffers at the

processors receiving the SCHEDs can fill up fast, and the sending process might have to be throttled. So the improvement in ISORULE performance due to an increase in the degree of pipelining may **deteriorate** as the degree of pipelining becomes higher. The results in figure 5.8 confirm these predictions.

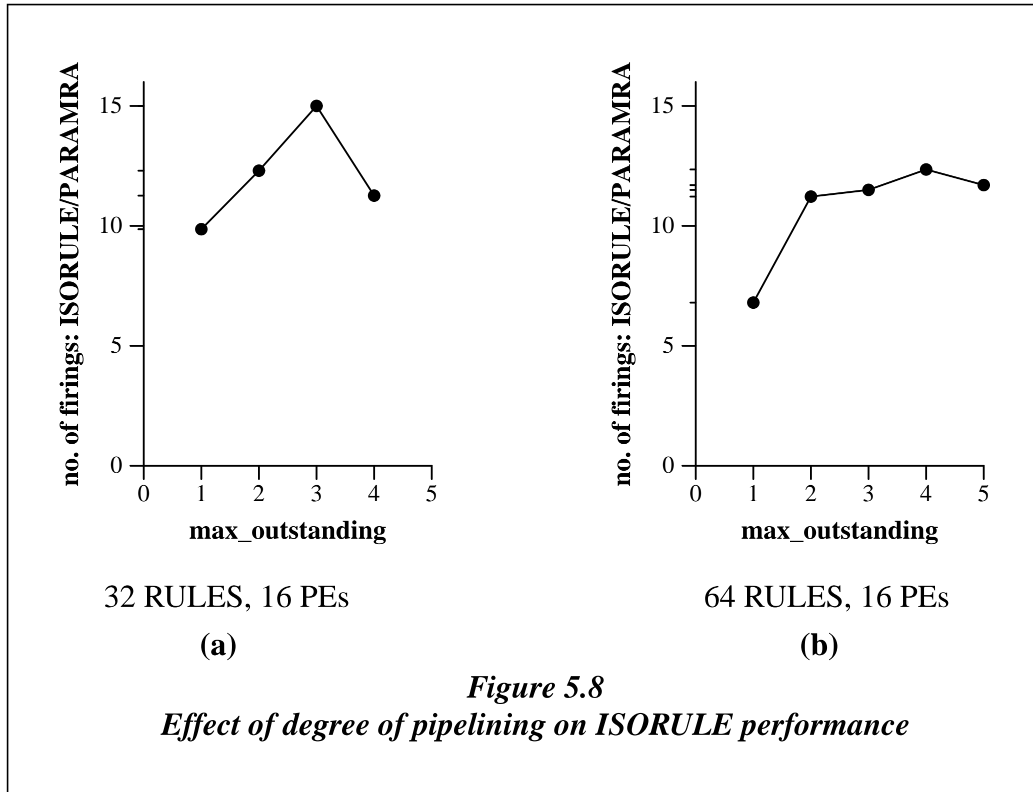


Figure 5.8
Effect of degree of pipelining on ISORULE performance

The results were obtained by varying the maximum allowable length of the pipe, for a rule set with 32 rules, and are shown in figure 5.8-a. Figure 5.8-b shows similar results for a rule set consisting of 64 rules. All runs are on a network of 16 PEs. In both cases, ISORULE performance improves with increasing pipelining up to a point, and then starts deteriorating. We discovered that the percentage of cancellations increases from an average of 6% with a pipe length of one, to an average of 18% for a pipelength of four.

5.4 Real Rule sets

Available parallelism in a rule-based system program depends on the problem that the program is trying to solve. Some programs (like *Monkey and Bananas*) do not allow any concurrent firings at all. The four rule-based system programs we tested are all written in OPS5, and permit varying degrees of concurrency. We analyzed these OPS5 rule sets by hand to obtain the amount of concurrency they provide.

5.4.1 Sequential Nature of Rule-based Programs

Several rule-based system programs are written with sequential control-flow in mind. For example, the Tourney rule-based program assumed that all possible instantiations of the rule *make-candidate* would be fired before firing the rule *make-choice*. We rewrote the Tourney program to remove the assumption of sequential execution. Whenever a rule assumed that all instantiations of another rule would be fired before it, we added condition elements to the first rule which made it ineligible to fire as long as the other rule was still firable. For instance, let us suppose a rule-based program requires all valid instantiations of rule R_1 to be fired before firing any instantiation of rule R_2. We could add extra condition elements to R_2 which allowed R_2 to fire only when the *condition elements for R_1 are not satisfied*, thus removing the sequential-execution assumption in the rule set.

5.4.2 The Monkey and Bananas Rule-based Program

This program consists of **18 rules**, and solves a toy problem. As mentioned in the introduction of this section, the Monkey and Bananas program permits *no parallelism* at all. Only one rule is eligible for firing at any given time, and execution of the program proceeds in a strictly sequential fashion. Running the ISORULE and PARAMRA systems

with four processors on this program, and comparing the number of simulation cycles each took to solve the problem gave the following result: the PARAMRA system outperformed the ISORULE system by a factor of **1.2**.

ISORULE cannot do better than PARAMRA because there are no parallel rule firings possible. The reason PARAMRA does *better* than ISORULE in this case is that the raw power of the isotach network (on which ISORULE is based) is *less* than the raw power of the conventional network (on which PARAMRA is based). Since only one rule instantiation can be fired at a time, the ISORULE system offers no advantage over the PARAMRA system.

5.4.3 The Tourney Rule-based Program

Tourney uses **16 rules** to solve the problem of scheduling a tournament, and exhibits *instantiation parallelism*. Two of the rules allow several eligible instantiations to fire concurrently. Running this program on ISORULE and PARAMRA gave the results shown in figure 5.9:

	$\frac{ISORULE}{PARAMRA}$ (pipe=1)	$\frac{ISORULE}{PARAMRA}$ (pipe=2)	$\frac{ISORULE}{PARAMRA}$ (pipe=3)
4 PEs	1.66	1.79	1.62
8 PEs	1.74	1.84	1.66

Figure 5.9
Normalized Results for Tourney

Figure 5.10 shows the unnormalized results (number of rule firings per 1000 simulation cycles) for Tourney.

	PARAMRA	*ISORULE* (pipe=1)	*ISORULE* (pipe=2)	*ISORULE* (pipe=3)

4 PEs	8.24	13.69	14.78	13.4
8 PEs	7.65	13.35	14.1	12.75

Figure 5.10
Unnormalized Results for Tourney

Increasing the length of the pipe (that is, the maximum number of outstanding firings allowed) improves ISORULE performance, but when the pipelength reaches three, performance deteriorates again. We attribute this deterioration to the disadvantages of over-pipelining outlined earlier. More outstanding rule firings means that there is a greater possibility of having to cancel rule firings. Increasing the number of processors does not help the Tourney program very much, as there is no rule parallelism in the rule set.

5.4.4 The Manners Rule-based Program

This program solves another toy problem, and employs **eleven rules** for the purpose. The rules are executed in a sequential order, but there is a pipe of length two in the rule set. (Here, *pipe* relates to the pipeline parallelism in **rule sets** discussed in Section 2.2.7, and not to the pipelining of firings in ISORULE.) Hence the available parallelism in the program is the pipeline parallelism of two. We ran Manners on ISORULE and PARAMRA, and computed the average number of rule firings per 1000 simulation cycles for each. The results are summarized in figure 5.11.

	PARAMRA	*ISORULE*	$\frac{ISORULE}{PARAMRA}$
4 PEs	11.6	16.9	1.45
8 PEs	8.6	17.1	1.9

Figure 5.11
Results for Manners

Increasing the number of processors doesn't make a significant difference to ISORULE performance because the length of the pipe is only two, allowing only two processors to be busy at the same time.

5.4.5 The Toru-Waltz Rule-based Program

Toru-Waltz is an OPS5 rule-based program with **9 rules** and an available rule parallelism of four. Figure 5.12 shows the results obtained by running the Toru-Waltz program on 4 and 8 processors. The ISORULE and PARAMRA results are in terms of rule firings per 1000 simulation cycles.

	PARAMRA	*ISORULE*	$\frac{ISORULE}{PARAMRA}$
4 PEs	8.25	16.6	2.01
8 PEs	4.7	18.7	3.97

Figure 5.12
Results for Toru-Waltz

Increasing the number of processors from four to eight does not improve the ISORULE performance much because Toru-Waltz does not allow a concurrency of more than four.

5.5 Analysis of the Results

The evaluation results from the synthesized rule sets show that the performance of the ISORULE system beats that of PARAMRA by an order of magnitude. However, the results from the OPS5 rule sets seem to belie these expectations. The reason for this discrepancy is that the OPS5 rule sets do not have enough potential parallelism. They are small rule sets, some of which solve toy problems, and which were designed with sequen-

tial rather than parallel systems in mind.

The characteristics of the OPS5 rule sets we used are unlike those of real-world rule sets as depicted in the literature. The small cycle problem in conventional MRA-driven rule-based systems arises from the fact that a rule firing affects a very small number of rule instantiations. The OPS5 rule sets, on the other hand, display a very high degree of conflict, causing the ISORULE system to cancel a large number of scheduled firings and also to send a large number of messages through the network since the reader set for a rule is so large. Our experiments revealed that the percentage of rule firing cancellations in ISORULE for these OPS5 rule sets ranges from 36% to 66%.

So the true effectiveness of our parallel rule firing strategy cannot be assessed by these four OPS5 rule sets alone. If rule sets are written with the following characteristics, we expect that ISORULE will exploit a large portion of the parallelism available:

- A large number of rules
- A small amount of conflict among the rules
- No sequential exploration of solution space
- No strict control embedded into the rule set via metarules

The confirmation of these expectations is left to future research.

5.6 Chapter Summary

This chapter presented a static model to analyze dependences among rules in a rule set. Results from both the synthesized rule sets and the real (OPS5) rule sets showed that ISORULE exploits most of the parallelism available in a rule set. Experiments with synthesized rule sets revealed that ISORULE outperforms PARAMRA by an order of magni-

tude. Since our initial analysis suggests that the performance improvement increases with the amount of concurrency in the rule sets, multiple orders of magnitude in performance improvement seem possible with larger rule sets exhibiting a low degree of conflict. The effect of the degree of pipelining in the ISORULE system on its performance was analyzed. These experiments revealed that ISORULE performs better with increasing degree of pipelining up to a point when it starts declining due to an increased number of cancellations, messages and throttling. Increasing the number of processors improves ISORULE performance, but the improvement can be sub-linear due to an increased number of messages having to be exchanged, or a larger percentage of cancellations. PARAMRA performance can deteriorate with an increased number of processors due to increased contention for the resolve-PE.

While results from the real (OPS5) rule sets were not as spectacular as the ones from the synthesized ones, ISORULE still exploited most of the parallelism offered by these rule sets. If rule sets are written so that they exhibit a high degree of concurrency, we expect that ISORULE will exploit a large portion of the parallelism available.

Chapter 6
Summary and Conclusion

ISORULE is an asynchronous parallel system for the execution of rule-based systems. In this thesis, we investigated various aspects of ISORULE behaviour and performance through an analytical model and confirmed our predictions through simulations.

ISORULE has the following advantages over existing parallel execution models for rule-based systems:

- No interprocessor communication is required for performing match, unlike systems that used node-level parallelism;
- Pipelining of rule firings allows more parallelism than simple rule-level parallelism would;
- ISORULE requires no dependence analysis for execution, and all rule instantiations that are eligible can be fired;
- Processor-idleness caused by the synchrony of conventional MRA-based systems is eliminated; thus, the *small-cycle problem* is not a *problem* for ISORULE;
- ISORULE does not introduce locking (and thus overly restrictive access);
- ISORULE has no potential for deadlock (since the rule firing with the lowest LFT can always make progress, and all processors agree on the values of the LFTs); and
- ISORULE requires only *two* multicast messages per rule firing, unlike PARS, which requires enable and disable messages, as well as acknowledgments.

The additional costs ISORULE imposes are:

- the network latency cost of enforcing the isotach invariant; and

- the cost of cancellation of rule firings.

Cancellations are expected to be infrequent, inferring from the observed fact that there is very little conflict among rules.

We devised a static dependency model to analyze the available amount of concurrency in a rule set, and to predict ISORULE performance based on this analysis. We then simulated the ISORULE system and a conventional parallel rule-based system (PARAMRA), modelling the underlying networks so as to include the cost incurred by ISORULE for maintaining isotach logical time. We discovered that ISORULE exploits most of the statically determinable rule parallelism in a rule set. The results from the test runs closely agreed with the predicted values from our model. Other results from our analysis were:

- We expected ISORULE to do better with more pipelining up to a point, and then decline due to increased cancellations, messages and throttling. Tests confirmed this trend in the behaviour of ISORULE.
- Another prediction was that with an increase in the number of processors, ISORULE performance would improve. This improvement could be sub-linear due to an increase in the percentage of cancellations of rule firings in ISORULE, or due to increased message traffic. PARAMRA performance could deteriorate with an increased number of processors due to increased message traffic and also due to increased contention for the resolve-PE. It was verified up to 32 processors through experiments that an increased number of processors boosts ISORULE performance.

Experiments with synthesized rule sets revealed that ISORULE outperforms PARAMRA by an order of magnitude. Since our initial analysis suggests that the perfor-

mance improvement increases with the amount of concurrency in the rule sets, multiple orders of magnitude in performance improvement seem likely with larger rule sets exhibiting a low degree of conflict. Tests with actual OPS5 rule sets did not yield the spectacular results obtained from the synthesized rule sets. This was because the OPS5 programs we used have very little potential for parallelism, and most of them are small rule sets that solve toy problems. Besides, the rule sets have been designed with sequential rather than parallel execution in mind. If rule sets are written with the following characteristics, we expect that ISORULE will exploit a large portion of the parallelism available:

- A large number of rules
- A small amount of conflict among the rules
- No sequential exploration of solution space
- No strict control embedded into the rule set via metarules

ISORULE *did* exploit most of the limited parallelism offered by the OPS5 rule sets.

Future Work

We have identified the following directions for future research:

- *Investigation of larger real rule sets:* An obvious area for future work with ISORULE is the investigation of larger real rule sets. If these rule sets exhibit the characteristics listed above, or if they can be rewritten to exhibit those characteristics, then we expect that ISORULE will exploit a large portion of the parallelism available.
- *Support for metarules:* ISORULE does not currently support metarules. Recall that metarules permit the imposition of special rules on the execution order of normal

rules in a rule set. We mentioned the sequential consistency guarantee offered by isotach networks as an aid toward supporting metarules. Modifying the ISORULE algorithm to make it handle metarules is a topic which needs to be further investigated.

- *Dynamic rule sets:* ISORULE makes the simplifying assumption that the rule sets are static. Consistency techniques developed for delta cache protocols, which are also based on isotach networks (see Section 3.2.2), provide the basis for extending ISORULE to dynamic rule sets.

Appendix A
Parameters to Simulation

We provide a list of the parameters we used for our simulation. Some of the parameters are specific to the synthetic workload; others are used with both the synthetic and the real workloads. The costs of various tasks are expressed in simulation cycles.

Parameter	**Range**	**Comments**
Number of PEs	4-32	both
Number of WME classes	10-400	synthetic workload
Number of rules	4-128	synthetic workload
Number of hot WME classes	1-20	synthetic workload
Probability of choosing a hot WME class	0.0-0.6	synthetic workload
Mean condition elements per rule	2-8	synthetic workload
Maximum condition elements per rule	4-12	synthetic workload
Mean action elements per rule	2-8	synthetic workload
Maximum action elements per rule	4-12	synthetic workload
Maximum outstanding firings	1-5	both
Maximum buffer length	40-200	both
Upper limit on invalidation probability	0.08-0.20	synthetic workload
Lower limit on invalidation probability	0.06-0.10	synthetic workload
Upper limit on validation probability	0.70-0.90	synthetic workload
Lower limit on validation probability	0.50-0.80	synthetic workload
Upper limit for initial instantiations	5-30	synthetic workload
Lower limit for initial instantiations	1-10	synthetic workload
Cycles by RULE to schedule firing	10	both
Cycles by RULE to process firing token	5	both
Cycles by WME to process firing token	2	both
Cycles by WME to process SCHED	2	both

Cycles by WME to process CANCEL	8	both
Cycles by WME to process CONFIRM	10	both
Cycles to pass WME tokens to Rete	10	both
Cycles by WME to pass firing token	2	both
Cycles by SIU for outgoing SCHED	5	both
Cycles by SIU for outgoing CANCEL	5	both
Cycles by SIU for outgoing CONFIRM	5	both
Cycles by SIU for incoming message	2	both
Cycles by SIU for (UN)THROTTLE	2	both
Cycles by Rete to process WME token	20	synthetic workload
Cycles for 1 iteration of MRA resolve-PE	10	both
Buffer length limit to throttle	10-20	both
Buffer length limit to unthrottle	5-10	both
Number of batches of observations	10000-50000	both
Interval at which observations are made	1-5	both

Figure A.1
Simulation Parameters

References

[ACHA89] Acharya A. and Tambe M., *Production Systems on Message passing Computers: Simulation Results and Analysis*, Proc. 1989 International Conference on Parallel Processing, 1989

[BERN81] Bernstein P. A. and Goodman N., *Concurrency Control in Distributed Database Systems*, Computing Surveys 13, 2, June 1981

[BROW85] Brownston L., Farrel R., Kant E. and Martin N., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, MA, 1985

[DAVI80] Davis R., *Meta-Rules: Reasoning about Control*, Artificial Intelligence 15, 1980

[DESU94] deSupinski Bronis R., *Simulating Cache Coherence with Atomicity and Sequential Consistency*, Masters' Project, University of Virginia, 1994

[DIXI87] Dixit V., *Transformation Techniques for Parallel Processing of Production Systems*, Ph.D. Thesis, University of Southern California, Oct. 1987

[FORG79] Forgy C. L., *On the Efficient Implementation of Production Systems, Ph.D. Thesis*, Carnegie-Mellon University, 1979

[FORG81] Forgy C. L., *OPS5 User's Manual*, CS-81-135, Carnegie-Mellon University, 1981

[FORG82] Forgy C. L., *Rete: A Fast Algorithm for the Many Pattern/ Many Object Match Problem*, Artificial Intelligence 19, 1982

[GARE79] Garey M. R. and Johnson D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979

[GUPT86] Gupta A., *Parallelism in Production Systems*, Ph.D. Thesis, Carnegie-Mellon University, March 1986

[GUPT88] Gupta A., Forgy C. L., Kalp D. and Tambe M. S., *Parallel OPS5 on the Encore Multimax*, Proc. 1988 International Conference on Parallel

Processing, Aug. 1988

[GUPT89] Gupta A., Forgy C. L. and Newell A., *High-Speed Implementations of Rule-Based Systems*, ACM Transactions on Computer Systems, Vol. 7, No. 2, May 1989

[HIGH89] Highland F. D. and Iwaskiw C. T., *Knowledge Base Compilation*, Proc. 11th International Joint Conference on Artificial Intelligence, IJCAI-89, 1989

[ISHI85] Ishida T. and Stolfo S. J., *Towards the Parallel Execution of Rules in Production System Programs*, CUCS-154-85, Department of Computer Science, Columbia University, 1985

[ISHI90] Ishida T., *Methods and Effectiveness of Parallel Rule Firing*, Proc. 6th IEEE Conference on Artificial Intelligence Applications, Washington DC, 1990

[ISHI94] Ishida T., *Parallel, Distributed and Multiagent Production Sytems*, Springer-Verlag, 1994

[KALP88] Kalp D., Tambe M., Gupta A., Forgy C., Newell A., Acharya A., Milnes B. and Swedlow K., *Parallel OPS5 User's Manual, The Production System Machine Project*, Department of Computer Science, Carnegie Mellon University, Nov. 1988

[KUOC91] Kuo C. M., Miranker D. P. and Browne J. C., *On the Performance of the CREL System*, Tech. Rep., Department of Computer Sciences, University of Texas at Austin, Feb 1991

[KUOS90] Kuo S., Moldovan D. and Cha S., *Control in Production Systems with Multiple Rule Firings*, Proc. 1990 International Conference on Parallel Processing Vol. 2, 1990

[KUOS91] Kuo S., *A Parallel Asynchronous Message-Driven Production System*, Ph.D. Thesis, University of Southern California, Sept. 1991

[LAMP78] Lamport L., *Time, Clocks and the Ordering of Events in a Distributed*

System, Comm. ACM 21, 7, July 1978

[LAMP79] Lamport L., *How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs*, IEEE Trans. on Computers 28, 1979

[MIRA87] Miranker D. P., *TREAT: A New and Efficient Algorithm for AI Production Systems*, Ph.D. Thesis, Columbia University, 1987

[MIRA90] Miranker D. P., Kuo C. and Browne J. C., *Parallelizing Compilation of Rule-Based Programs*, Proc. 1990 International Conference on Parallel Processing, Vol. 2, 1990

[MOLD89] Moldovan D., *RUBIC: A Multiprocessor for Rule-Based Systems*, IEEE Trans. Systems, Man Cybernet, July 1989

[PAPA86] Papadimitriou C., *Database Concurrency Control*, Computer Science Press, 1986

[REYN89] Reynolds P. F. Jr., Williams C. and Wagner R. R., *Parallel Operations*, UVa Computer Science Technical Report 89-16, Dec. 1989

[REYN92] Reynolds P. F. Jr., Williams C. and Wagner R. R., *Empirical Analysis of Isotach Networks*, UVa Computer Science Technical Report 92-19, June 1992

[ROMA89] Roman G. and Cunningham H. A., *A Shared Dataspace Model of Concurrency-Language and Programming Implications*, Proc. 9th International Conference on Distributed Computing Systems, Los Alamitos, Ca 1989, IEEE Computer Society Press, Silver Spring, MD, 1989

[SCHM88] Schmolze J., *An Asynchronous Parallel Production System with Distributed Facts and Rules*, Proc. AAAI-88 Workshop on Parallel Algorithms for Machine Intelligence and Pattern Recognition, St. Paul, MN, Aug. 1988

[SCHM90] Schmolze J. and Goel S., *A Parallel Asynchronous Distributed Production System*, Proc. 8th National Conference on Artificial Intelligence, AAAI-90,

1990

[SCHM92] Schmolze J. G. and Nieman D. E., *Comparison of Three Algorithms for Ensuring Serializable Executions in Parallel Production Systems*, Tenth National Conference on Artificial Intelligence, San Jose, CA, 1992

[SHAW85] Shaw D. E., *NON-VON's Applicability to Three AI Task Areas*, International Joint Conference on Artificial Intelligence, Aug. 1985

[STOL82] Stolfo S. J. and Shaw D. E., *DADO: A Tree-Structured Machine Architecture for Production Systems*, National Conference on Artificial Intelligence, 1982

[STOL91] Stolfo S., Dewan H. and Wolfson O., *The PARULEL Parallel Rule Language*, Proc. 1991 International Conference on Parallel Processing, Vol. 2, 1991

[WALT87] Waltz D. L., *Applications of the Connection Machine*, Computer 20,1, Jan. 1987

[WILL91] Williams C. and Reynolds P. F. Jr., *Combining Atomic Actions in a Recombining Network*, UVa Computer Science Technical Report 91-33, Nov. 1991

[WILL93] Williams C. C., *Concurrency Control in Asynchronous Computations*, Ph.D. Thesis, University of Virginia, 1993