DESIGN AND CONTROL OF PARALLEL RULE-FIRING PRODUCTION SYSTEMS

A Dissertation Presented

by

DANIEL E. NEIMAN

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1992

Department of Computer and Information Science

© Copyright by Daniel E. Neiman 1992

All Rights Reserved

DESIGN AND CONTROL OF PARALLEL RULE-FIRING PRODUCTION SYSTEMS

A Dissertation Presented

by

Daniel E. Neiman

Approved as to style and content by:

Victor R. Lesser, Chair

Daniel D. Corkill, Member

Janice Cuny, Member

James G. Schmolze, Member

Richard Giglio, Member

Arnold L. Rosenberg (Acting Chair), Department Head
Department of Computer Science

The process of acquiring a Ph.D. is a lengthy one, and sadly, the years are often measured by the passing of those we care for. I would like to dedicate this dissertation to the memory of my father, Joseph, who I think would have been very proud to see me complete my doctorate.

I would also like to dedicate it to my first cat, Lily, who I hope is still chasing birds and mice in some feline Elysian Fields.

ACKNOWLEDGMENTS

Foremost, I would like to acknowledge the continuous support and encouragement I received from my advisor, Victor Lesser. At a particularly dark time in my graduate student career, Vic let me work on a topic of my own choosing and has provided me with continuous support and guidance during my research. Jim Schmolze of Tufts University graciously agreed to be on my committee and introduced me to many of the intellectual concerns underlying correctness in parallel rule-firing production systems. Through his use of UMPOPS in his own research, Jim also inspired me to improve the performance and features of the language and we have had many stimulating conversations, usually in the frenzied weeks before a AAAI deadline. Dan Corkill and Jan Cuny also served on my committee and improved the quality of the research markedly through their comments and criticisms. I would also like to thank my outside member, Richard Giglio, for agreeing to serve on my committee.

This thesis would not exist were it not for the contributions of software and ideas from others in the field. Kelly Murray started it all by coming into my office one day and asking if I wanted to write a parallel OPS5 for his parallel Common Lisp. Without TopCl, this research would not have been possible. Thanks to Charles Lanny Forgy who implemented OPS5 and then very kindly released it into the public domain, thereby providing an invaluable tool to an entire generation of researchers and experimenters in the field of production systems. The language UMPOPS that is described in this dissertation is built entirely on top of this public domain OPS5. Toru Ishida gave permission to use and modify his Toru-Waltz benchmark. I am

also indebted to Salvatore Stolfo, the implementors of Alexsys, and the trustees of Columbia University for their permission to use Alexsys in my experiments.

Many of my friends contributed either directly or indirectly to the production of this thesis. Penni Sibun and John Martin selflessly proofread the rough drafts of this dissertation and any assaults made upon the English language in the final draft are despite their very best efforts. Keith Decker acted as LaTex and Postscript guru; because of Keith's indefatigable research into the latest innovations of desktop publishing, the creation of this document was much less painful than it could have been.

Because there is more to life than just thesis preparation (although it often didn't seem that way), I would like to thank those people whose presence made graduate school an enjoyable experience. The members of the DIS lab, past and present, Bob Whitehair, Keith Decker, Al (Bart) Garvey, Marty Humphrey, Dave Hildum, Zarko Cvetanovic, and Ed Durfee, proved to be an unusually cohesive and sociable group. The Westbrooks, Dave, Terry, Brian and Josh, served as the nucleus of many social gatherings, camping trips, and marathon croquet matches. Scott Anderson whiled away many a summer afternoon thrashing me on the tennis courts. Penni Sibun taught me to play bridge, talked me into being owned by my first cat, and has always been willing to listen to me complain about life in general. Claire Cardie and David Skalak have invited me along on several skating, hiking, and bicycling expeditions and have more than once rescued me during periods of automotive distress. John Martin and Maureen Tracy have made me their guest many times, allowing me to escape the stress of thesis preparation for a while.

My family, too, has been unstinting in their support during my years of grad school and has always welcomed me when I suddenly appeared after long periods of being incommunicado. My brother Bill, in particular, was always up for a ski trip or

a leisurely 12 or 16 mile hike along the Appalachian Trail, occasionally in the dark. Finally, I would like to acknowledge the contributions of my cats, Lily, Socrates, and Kittyhawke, who not only inspired many of the examples in this document, but also put up with my extended absences and late nights in the lab and repaid me with affection, entertainment, and the occasional live bird or rodent in the living room.

It is impossible to summarize the experiences and relationships of six years of graduate school in a few sentences, and I apologize to all of those whose names I omitted. To all those who have lent their friendship and support during the sometimes trying years of graduate school, I offer my sincere thanks and appreciation.

This research was conducted on a Sequent multiprocessor, which was purchased through NSF-CER contract DCR-8500332, and on a TI Explorer II, purchased through a grant from the Office of Naval Research under University Research Initiative grant number N00014-86-K-0764. This work was also partially funded by NSF contract CDA-8922572 and DARPA contract N00014-89-J-1877. Despite their kind support, these agencies require it to be known that the content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

ABSTRACT

DESIGN AND CONTROL OF
PARALLEL RULE-FIRING
PRODUCTION SYSTEMS

SEPTEMBER 1992

DANIEL E. NEIMAN

B.S., UNIVERSITY OF CONNECTICUT

M.S., UNIVERSITY OF CONNECTICUT

Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Victor R. Lesser

This dissertation studies the issues raised by the parallel execution of rules in a pattern-matching production system. There are two main areas of concern: maintaining correctness during the course of simultaneous rule executions, and controlling the execution of productions without introducing serial bottlenecks. It is demonstrated that guaranteeing program correctness using a serializability criterion introduces an unacceptable overhead and reduces the potential parallel speedup to a single order of magnitude. Instead of attempting to automatically extract coexecutable sets of parallel rules, the approach taken in this research is to define a minimal set of language constructs which allow correct parallel programs to be designed. The view that the rule-based computation has an algorithmic structure allows us to attach a semantic interpretation to rule firing. By examining the role of each rule in the overall computation, we can understand and begin to find a solution to the problems of controlling rule firing and ensuring correctness while maximizing effective use of parallel processing resources.

When rules are executed in parallel, the conventional control mechanisms applied to rule-based systems act to limit parallel activity. Two novel rule-firing policies are described: an asynchronous rule-firing policy that causes rules to be executed as soon as they become enabled, and a task-based scheduler that allows multiple indpendent tasks to run asynchronously with respect to each other while allowing rules to execute either synchronously or asynchronously within the context of each task. Because the asynchronous execution of rules reduces the opportunities for performing conflict resolution, methods for performing heuristic discrimination at various points in the rule execution cycle are discussed.

The experimental results of this research are presented in the context of UMass Parallel OPS5, a rule-based language that incorporates parallelism at the rule, action, and match levels, and provides language constructs for supporting the design of parallel rule-based programs including a locking scheme for working memory elements and operators for specifying local synchronization of rules and actions. Results are presented for a number of programs illustrating common AI paradigms including search, inference, and constraint satisfaction problems.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT viii

LIST OF TABLES xv

LIST OF FIGURES xvi

1. INTRODUCTION 1

   1.1 Distinguishing Production Systems from Related Architectures and Languages 11

      1.1.1 Data-directed Programming and Imperative Languages 11
      1.1.2 Blackboard systems 12
      1.1.3 Database systems 13
      1.1.4 Logic programming 14

   1.2 Contributions 14

      1.2.1 Speedup Due to Rule Parallelism 15
      1.2.2 Correctness 15
      1.2.3 Control 17
      1.2.4 UMass Parallel OPS5 – An Experimental Testbed for Parallel Rule Firing 19

   1.3 Organization of the Dissertation 20

2. RELATED WORK 24

   2.1 Overview: Rule-based Systems 24
   2.2 The OPS5 Language 24

      2.2.1 Definitions 26

         2.2.1.1 Levels of Parallelism 30

      2.2.2 Control of OPS5 Programs 30
      2.2.3 The Rete Net 32

   2.3 Research in Parallel Production Systems 34

2.3.1 Compilation of the Rete Net . . . 35
2.3.2 Parallelism in OPS5 . . . 36
2.3.3 Production Parallelism . . . 38
2.3.4 Node and Intra-node Parallelism . . . 39
2.3.4.1 Extremely Fine Grained Parallelism within the Rete Net . . . 41
2.3.5 Action Parallelism . . . 41
2.3.6 Application Parallelism . . . 42
2.4 Parallel Execution of Rules . . . 45
2.4.1 Achieving Serializable Behavior in a Parallel Program . . . 45
2.5 Parallel Rule-firing Production Systems . . . 47
2.5.1 The CREL System . . . 47
2.5.2 PARULEL . . . 49
2.5.3 Control of Rule Sequencing . . . 51
2.5.3.1 Parallel Rule Firing with Fuzzy Logic . . . 53
2.5.4 A Note on Rule versus Instance Parallelism . . . 53
2.5.5 Architectures for Production Systems . . . 55
2.5.5.1 DADO . . . 55
2.5.5.2 Implementation of OPS5 on Non-Von . . . 57
2.5.5.3 CUPID and DRete . . . 58
2.5.5.4 Message Passing Architectures . . . 58
2.5.5.5 Current Trends in Implementation Architectures . . . 59
2.6 Conclusion: Related Work . . . 60
3. CORRECTNESS AND DESIGN . . . 61
3.1 Correctness and Serializability . . . 62
3.2 A Locking Scheme for Ensuring Partial Correctness of Working Memory 66
3.2.1 Region Locks and the Make-Unique Construct . . . 69
3.2.2 Principal Advantages of a Working Memory Locking Scheme . 70
3.2.3 Limitations of Working Memory Locks . . . 72
3.2.4 A Mechanism for Detecting Interactions Due to Negative Condition Elements . . . 73
3.2.4.1 Experimental Verification of Overhead Analysis . . . 80
3.2.5 Conclusions: Guaranteeing Serializability . . . 84
3.3 Designing for Correct Parallel Execution . . . 85
3.3.1 Spurious Rule Interactions . . . 86
3.3.2 Semantics of Rule Firing . . . 87

3.3.2.1 Contention for Resources 89
3.3.2.2 Control Elements 91
3.3.2.3 Search States 92
3.3.2.4 Merging Solutions 93

3.3.3 Representation of Unique Objects (Data Parallelism) 93
3.3.4 Inference 94
3.3.5 Domain Facts 96
3.3.6 Summary: Semantics of Rule-based Systems 97

3.4 Functionally Accurate Computations 97
3.5 Summary: Correctness in Parallel Rule-Firing Systems 99

4. CONTROL ISSUES IN PARALLEL RULE-FIRING SYSTEMS 101

4.1 Rule-Firing Policies 104
4.2 Asynchronous Rule-Firing 105

4.2.1 Experiments with Rule-Firing Policies 106

4.2.1.1 Experiment 1: Explicit Synchronization 107
4.2.1.2 Experiment 2: Synchronization via Conflict Set . . . 108
4.2.1.3 Experiment 3: Asynchronous Production Execution . 110
4.2.1.4 An Experiment with Unbalanced Rule Execution Times 111

4.2.2 Summary of Experiments with Asynchronous Rule-Firing . . . 114
4.2.3 Monotonicity in the Eligibility Set 114

4.3 Control Tasks 116

4.3.1 Defining Tasks and Task Quiescence 117
4.3.2 The Task Implementation 119
4.3.3 Summary: Control Tasks 120

4.4 Sequential Control 122

4.4.1 Set Functions 124

4.4.1.1 Set-oriented Rules and Asynchronous Firing Policies 125
4.4.1.2 Acquiring Locks for Set-Rules 127

4.4.2 A Model of Program Phases 127
4.4.3 Mixed-Mode Parallelism and Mode-changing 128
4.4.4 Optimistic Concurrency 131

4.5 Heuristic control 131

4.5.1 Dynamic Control of Rule-Firing Policies 134
4.5.2 Interactions between Consistency Maintenance and Heuristic Control 136

4.6 Conclusion: Control of Parallel Production Systems 138

5. UMass Parallel OPS5 140
5.1 The Rule-Firing Architecture 140
5.1.1 Rule Demons 143
5.2 Modifications to LHS Syntax in UMPOPS 147
5.2.1 LHS Meta-level Notation 147
5.2.2 Annotating Mode-changing Productions 147
5.2.3 Other Uses of the Meta Notation 148
5.3 New Righthand Side Functions in UMPOPS 150
5.3.1 Invoking Action and Match-level Parallelism in the RHS . . . 150
5.3.1.1 Action Parallelism 151
5.3.1.2 Match-Level Parallelism 152
5.3.2 Make-unique 152
5.3.3 Control Task Syntax 154
5.3.4 Set Functions and Synchronization Groups 155
5.3.4.1 Syntax of the Set Notation 156
5.3.4.2 Synchronization Groups 157
5.3.5 Map-vector 158
5.4 Multiple Worlds 160
5.5 Implementation of Parallel Matching in UMPOPS 163
5.5.1 The Rete Net 163
5.5.1.1 Rete Net Overview 164
5.6 Implementing Match-level Parallelism 168
5.7 Synchronization of 2-input Nodes 172
5.7.1 Synchronization and Sharing of Memory Nodes 178
5.8 Race Conditions 179
5.8.1 Avoiding Critical Regions 179
5.9 Implementing Action-level Parallelism 181
5.10 Summary: UMPOPS 183
6. Experiments 185
6.1 Analyzing the Toru-Waltz Benchmark for Rule Parallelism 186
6.1.1 Mode Changes in Toru-Waltz 190
6.2 The Travelling Salesperson Benchmark 191
6.2.1 Heuristic Control in TSP 194
6.2.2 Asynchronous Rule-Firing in TSP 196
6.2.3 Merging Solutions 197

6.2.4 Queue Latencies in TSP 200
6.3 Alexsys: Parallelization of a "Real-World" Rule-based System 202
6.3.1 The Alexsys program 202
6.3.2 Parallelizing Alexsys 205
6.3.3 Modifications to Alexsys 207
6.3.4 Data Management in Alexsys 208
6.3.5 Experimental Results with Alexsys 211
6.3.6 Conclusions: Alexsys 213
6.4 Performance Analysis of Toru-Waltz and TSP 214
6.4.1 Performance Measurements: Toru-Waltz 215
6.4.2 Performance Measurements: TSP 220
6.4.3 Measurements of Contention in the Rete Net 220
6.5 Summary 227
7. CONCLUSION 229
7.1 Motivation 229
7.2 Contributions 230
7.2.1 Control and Rule-Firing Policies 231
7.2.1.1 The Asynchronous Rule-Firing Policy 232
7.2.1.2 Heuristic Control in Asynchronous Rule-Firing Systems 233
7.2.1.3 Task-based Rule-Firing Policies 234
7.2.2 Explicit Control of Programming Sequencing 235
7.3 Correctness 236
7.4 UMPOPS: A System for Benchmarking Parallel Rule-based Programs 239
7.5 Future Work 241
APPENDICES
A. THE TORU-WALTZ BENCHMARK 244
B. THE TRAVELLING SALESPERSON PROBLEM 252
BIBLIOGRAPHY 259

LIST OF TABLES

Table Page

3.1 Frequency of rule interactions compared to total number of rule executions. 67

6.1 Performance of the nondeterministic UMPOPS scheduler using a single rule-demon compared with the performance of a serial OPS5 scheduler using conflict resolution for three benchmark programs. . . 216

LIST OF FIGURES

Figure Page

1.1 The "ideal" architecture of the parallel rule-firing system; rules fire as soon as they become eligible. 6

1.2 The architecture of a parallel rule-firing system with rule scheduling, locking and control constructs. 6

1.3 The recognize-select-act loop of conventional production system architectures. 8

2.1 A "complete" cognitive model of *Felis Domesticus* 27

2.2 An example of conflict resolution. The "best" rule is chosen on the basis of recency and specificity. 31

2.3 The Rete net for a simple OPS5 program 35

2.4 Match-level parallelism for a single working memory change in the cat example. 40

2.5 Action parallelism combined with node parallelism greatly increases the number of concurrent node activations. 43

2.6 Rule parallelism allows all the instantiations in the conflict set to be executed concurrently. 44

3.1 If the concurrent execution of rules can produce a result which is produced by any sequential execution, the program is said to be *serializable*. 63

3.2 When mutually disabling rules are allowed to fire concurrently, the result may be a working memory state which could not be produced by any sequential rule firing. 64

3.3 Execution of clashing rules in OPS5 can result in the assertion of redundant working memory elements. 65

3.4 The overhead for acquiring locks in the Toru-Waltz benchmark measured in terms of percentage of total rule execution time. 71

3.5 An algorithm for asynchronously detecting rule interactions involving negated condition elements. 79

3.6 The essential architecture for a parallel rule-firing system with a serial mechanism for lock acquisition and rule-interaction detection. 81

3.7 The parallel speedups of two runs of the Toru-Waltz benchmark are shown. It can be seen that the performance of the version incorporating the lock-detection mechanism is approximately 25% slower than the version without. 82

3.8 The processor utilization graph for Toru-Waltz without rule-interaction detection. During the enumeration phase, all 20 processors are employed in executing rules. 83

3.9 The processor utilization for Toru-Waltz with rule-interaction detection enabled. During the enumeration phase, only an average of 10 rules execute at a given time and the remaining processors are idle. . 84

3.10 Rules instantiations demonstrating the possibility of two interacting chains of inference. 95

4.1 Processor utilization for a synchronous rule-firing policy with bottleneck mode-changing rules. 111

4.2 Processor utilization for a synchronous rule-firing policy with mode-changing rules eliminated. The delays due to conflict resolution are exaggerated for illustrative purposes. 112

4.3 Processor utilization for a fully asynchronous rule-firing policy. 112

4.4 Processor utilization for the circuit benchmark with rules of unequal run time, using a synchronous rule-firing policy. 113

4.5 Processor utilization for the circuit benchmark with rules of unequal run time, using an asynchronous rule-firing policy. 113

4.6 The rule-firing architecture required to implement multiple asynchronous control tasks. 121

4.7 If data independence can be insured, then multiple scheduling processes can be assigned to tasks, avoiding potential serial bottlenecks in the scheduling phase. 121

4.8 The location of the gating element affects the amount of partial matching which can take place in the match process. 129

5.1 The architecture of the parallel rule-firing system. 145

5.2 By partitioning the memories of the Rete net, a multiple world implementation suitable for parallel search can be transparently achieved. To minimize copying, a "base" space or partition can be defined that contains knowledge guaranteed to remain stable over the course of the search. 162

5.3 A token arrives at an AND node. 166

5.4 A token arrives at the lefthand input of a NOT node. 169

5.5 A token arrives at the righthand input of a NOT node. 170

5.6 When matching tokens arrive at an AND node simultaneously, synchronization errors can occur. 174

5.7 The synchronization process for an AND node. 176

5.8 Race conditions due to intra-node parallelism. 180

6.1 The time required by the initialization phase of the Toru-Waltz program can be reduced by the use of rule and action-level parallelism.188

6.2 The time required to execute the mode-changing production in Toru-Waltz can be reduced by the use of match-level parallelism. 192

6.3 The **propagate** rule from TSP. 195

6.4 The **init-solution** rule which initializes a data-merging episode in TSP. 198

6.5 The **new-and-improved** rule which implements a merge operation for TSP. 199

6.6 The parallel speedup achieved for Alexsys. 213

6.7 The processor utilization graph for parallel Alexsys with a maximum of 18 concurrent allocation tasks. 214

6.8 The solution qualities produced by the parallel Alexsys allocation process graphed against maximum concurrent allocation tasks. 215

6.9 The parallel speedup graph for Toru-Waltz with 20 processors. The left axis shows run-time in seconds. The right axis shows the speedup factor achieved. 218

6.10 The processor utilization graph for Toru-Waltz with 19 "demon" processors. In the initialization phase, a large number of action demons become active while date is added to working memory. In the `enumerate` phase, 19 rule demons become active. During the mode change from the `enumerate` phase to the `reduce` phase, 18 match demons are active. One rule demon must be active during this time to perform synchronization. Finally, rule parallelism becomes dominant during the `reduce` phase and tapers off as less work becomes available. 219

6.11 The parallel speedup for Toru-Waltz in terms of rule firings per second. With periods of reduced parallel firing due to initialization and mode-changing eliminated, rule firing rates approach 500 rules per second. 221

6.12 The processor utilization for the TSP benchmark with 20 processors. 222

6.13 The parallel speedup for the TSP benchmark with 20 processors. The graph depicts the decrease in execution time as the number of processors increases, the ratio of parallel run-time to serial run-time, and the number of rules fired per second. 223

6.14 The average time required to execute certain rules in Toru-Waltz, plotted against number of processors. 224

6.15 The average time required to execute certain rules in TSP, plotted against number of processors. 225

CHAPTER 1

INTRODUCTION

This dissertation investigates the implications of parallelism on the design and control of rule-firing systems. Rule-based (or production) systems have been used extensively to build expert systems and to experiment with cognitive models. The advantage of such systems is their ability to encapsulate pattern-driven knowledge in discrete packages; their major disadvantage has been the high overhead and consequent slow speed of rule matching and execution. Even before multiprocessors became widely available commercially, the potential of parallel processing for increasing the speed of production systems was being studied [Acharya and Tambe, 1989, Blelloch, 1986, Forgy, 1979, Forgy, 1980, Gupta, 1984, Hillyer and Shaw, 1988, Ishida and Stolfo, 1985, Krall and McGehearty, 1986, McCracken, 1981, Moldovan, 1986, Morgan, 1988, Oflazer, 1984, Stolfo and Miranker, 1984, Uhr, 1979, Tenorio and Moldovan, 1985]. Initially, this research focused on increasing the speed of matching rules against productions, largely because of Forgy's assertion that 90% of the work of productions takes place in the match phase [Forgy, 1979]. Initial forecasts of the speedup possible due to fine-grained parallelism at the match level were quite optimistic – in the range of three orders of magnitude. But these expectations were not to be realized – Gupta examined the characteristics of existing expert systems and concluded that the maximum likely speedup was a single order of magnitude, regardless of the number of processors available [Gupta, 1987]. Essentially, the reasons for this limitation are the relatively small number of rules affected by each working memory change, which limits the amount of work which can be performed during the match phase, and the relatively high ratio of overhead and scheduling to the actual work being performed at this fine level of granularity. That is, the amount of work being performed by each match process is fairly small and the overhead of spawning and scheduling processes to perform that work becomes significant.

While these results serve to place some realistic limits on the expectations for parallelism, they do not represent the final word on potential speedup. If the speedup for a single working memory change or single rule firing is limited, it is certainly possible to perform many working memory changes in parallel (*action* parallelism) or to execute many rules concurrently (*rule* parallelism). Either of these approaches

will increase the matching to be performed, and therefore the number of processors that can be effectively employed.

Considerable research is being performed on parallel rule-firing production systems, that is, systems in which many rules are allowed to fire at once [Ishida and Stolfo, 1985, Ishida, 1990, Kuo *et al.*, 1991, Kuo and Moldovan, 1991, Moldovan, 1989, Pasik and Stolfo, 1987, Schmolze, 1991, Stolfo *et al.*, 1991b, Wolfson and Ozeri, 1990]. There are several models of parallel rule firing, ranging from distributed implementations in which non-intersecting sets of rules and/or working memory are assigned to each processor to shared-memory implementations in which all processors have access to all of production and working memory; this research was performed using the shared-memory model. The level of parallelism which will be discussed in most detail is *instance* parallelism, the least restrictive form of rule parallelism, in which any rules within a set, including multiple instantiations of the same rule, are allowed to execute concurrently.

Measurements of the efficiency of parallel rule firing depend on many factors, including the number of processors gainfully employed, the number of useful activities pursued versus redundant or unnecessary actions, the reduction in computation time per number of processors and the amount of time that processors spend idle. One of the primary measurements used in the experimental section of this research is parallel speedup; the ratio of parallel execution to an equivalent serial implementation. Because the language used to generate the results is optimized with respect to previous Lisp-based OPS5s, all speedup measurements are recorded in relation to a configuration of the system which devotes only a single processor to executing rules. The use of this configuration eliminates the overhead of the conventional OPS5 rule scheduling routines and provides a more accurate measurement of parallel speedup. The effectiveness of a parallel implementation can also be measured by the average number of processors which are kept active during the course of problem solving. With this metric in mind, the ideal architecture for a parallel rule-firing system is one in which rules fire as soon as they become eligible, thereby ensuring the maximum usage of processing resources (see Figure 1.1). In practice, for reasons which will be discussed shortly, it is rarely feasible to execute rules totally asynchronously for the entire course of a computation. Thus, we end up with variations of the architecture shown in Figure 1.2: eligible rules are placed in an *eligibility set*,[1] undergo processing to insure that they do not conflict with other eligible rules, are scheduled, and then placed on an execution queue. In the model displayed here, a single processor is responsible for the conflict resolution/scheduling phase. Although more complex

[1]The common term for this data structure is "conflict set." This usage is historical and implies that all the rules which are currently able to fire are in some way conflicting or competing for a single processing resource and will therefore require *conflict resolution* to select the appropriate action. In this dissertation, I will use the more accurate term "eligibility set" to describe the set of rule instances eligible to fire and reserve "conflict set" for those situations in which the eligible rules actually undergo conflict resolution.

models with multiple scheduling processors can be devised, the evaluation of system performance will remain essentially valid.

When modeling the behavior of this architecture in terms of maximum rule throughput, we can assume that rules are generated as fast as they can be executed, that there exists an infinite number of servers or *rule demons*, and that rules fire as soon as they become eligible. This is equivalent to modeling the rule-firing architecture as an $M/M/\infty$ queue [Kleinrock, 1975] which assumes multiple queues and an infinite number of servers. If we assume an infinite (or very large) number of servers to execute rules, the limiting factor in the system will be the rate at which rules can be placed in the execution queues. We can estimate the influence of the serial portion of the computation in terms of the average time it takes to execute a production. If we call the time of rule execution T_{RE} and the serial scheduling/control period T_S, then the number of active processors will be limited to T_{RE}/T_S. For example, if the scheduling and control activities take as much as 10% of the average rule execution time, then by the time the tenth rule is scheduled, the first will have completed execution and the average degree of parallelism will be limited to a factor of 10.

Clearly, the higher the overhead of scheduling, control, or any other serial activity, the lower the rate of parallel rule firing. The implications of this analysis are far-reaching and motivate much of the work on correctness and control described in this thesis. Rules are designed to represent a fine-grained unit of recognition and reaction. Because the life span of a rule-firing is so short, prolonged deliberation over control decisions is not possible and any delays in executions will be more significant than for coarser-grained systems. Virtually all algorithms for managing rule-based systems contain serial processing phases and must be revised to support parallel activity by reducing or eliminating these phases. In this dissertation, success at parallelizing rule-based systems will be measured in terms of the reduction of serial overhead caused by control or correctness algorithms and the number of processors which could be kept active (assuming an unlimited number of processors). Limitations on parallelism due to hardware constraints, such as bus contention on shared memory machines, will not be explicitly addressed due to lack of capabilities for measuring such influences directly.

The exploitation of parallelism due to concurrent execution of rules does not come without a cost. Unlike match parallelism which can be implemented transparently without changing the semantics or syntax of the language, existing rule-based languages can not easily support rule (or action) parallelism. Results of attempts to automatically extract parallelism from existing rule-based systems have been disappointing [Ishida and Stolfo, 1985]. Most rule-based programs rely on a sequential control scheme which quite effectively limits the number of rules eligible to fire at any given time. Those rules which are eligible to fire in parallel can potentially interact, causing inconsistencies and errors to appear within working

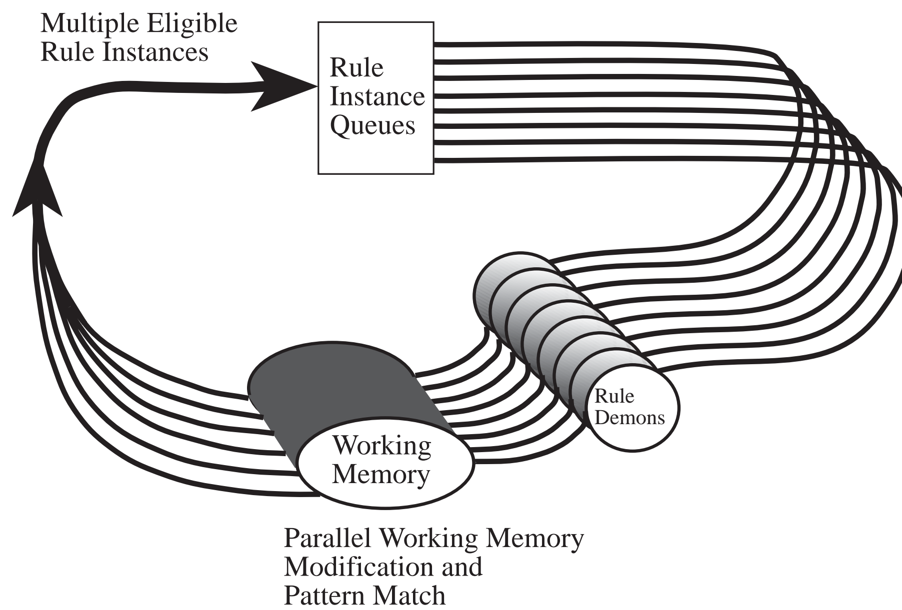


Figure 1.1: The "ideal" architecture of the parallel rule-firing system; rules fire as soon as they become eligible.

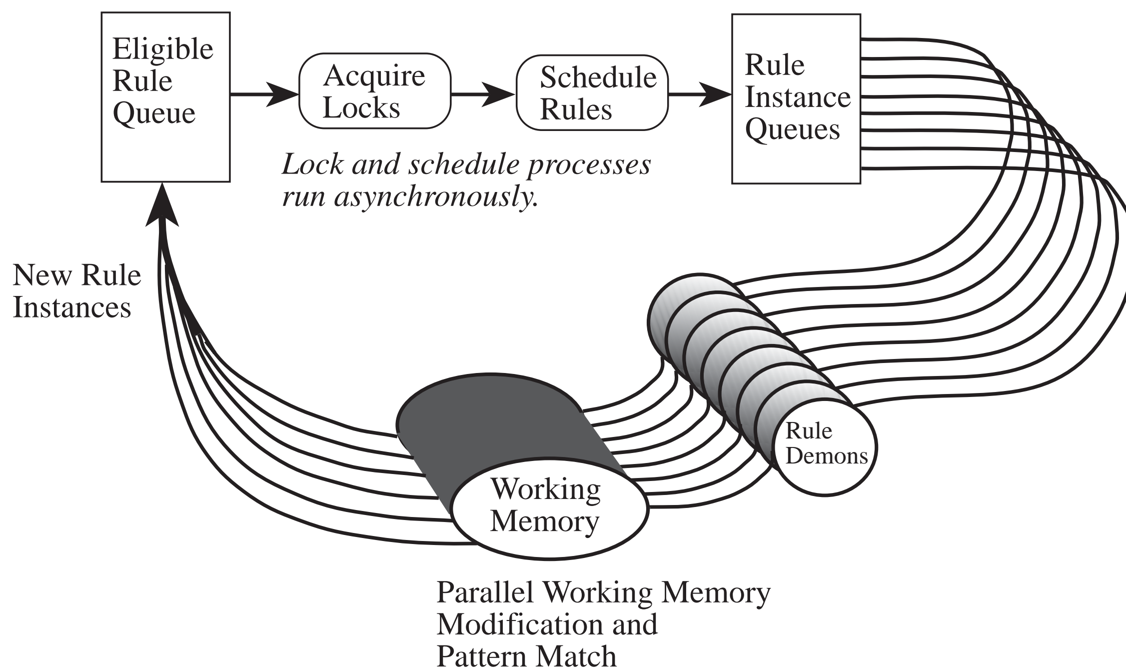


Figure 1.2: The architecture of a parallel rule-firing system with rule scheduling, locking and control constructs.

memory; mechanisms are therefore required to examine eligible rules to insure that they can safely execute concurrently.

There are several possible approaches to the problem of insuring that rules do not interact in a pathological fashion. The most thoroughly researched has been the implementation of mechanisms to ensure that the parallel execution of rules is *serializable*, that is, that there is *some* serial execution of rules which could produce the same result. The principal advantage of these mechanisms is that, like match-level parallelism, they can be applied to existing systems and the result will be provably correct (or at least serializable). The definition of correctness as serializability has been extended by several researchers [Kuo and Moldovan, 1991, Srivastava and Wang, 1991] to incorporate the concept of conflict resolution; that is, the parallel execution produces a result which would be produced by a serial execution incorporating a specific conflict resolution algorithm. (In general, this work assumes that the primary function of conflict resolution is to control the sequencing of rules.) As will be discussed in this dissertation, although all these rule interaction detection algorithms guarantee correctness, they are, as will be demonstrated, expensive relative to the cost of executing rules, and may restrict the level of parallelism achieved by the system.

An alternative approach is to *design* rule-based systems so as to produce high levels of parallelism and correct execution. One of the theses that will be explored in this dissertation is the contention that a combination of design techniques, programming idioms and supporting language mechanisms will make it possible to create highly parallel rule-firing systems by eliminating the need for a serializing rule-interaction-detection phase. The drawback to the design approach is that it places considerable demands on the designer of the system; design of parallel rule-firing systems is not well understood, and the behavior and performance of rules during the course of parallel execution frequently defies intuition. Research on the formal verification of parallel rule-firing programs currently underway [Gamble, 1990], however, verification is not addressed in this thesis. Instead, the rule-firing mechanism is instrumented so that the performance and degree of parallelism exploited can be measured and so that the number of interacting rules is reported. If insufficient parallelism is achieved, or many rules are competing for resources, then a redesign is probably in order.

Parallel rule-firing requires a re-examination of the most basic premises of rule-firing languages; in particular, changes must be made to the basic rule-firing algorithms. The conventional rule-firing architecture is based on a simple cycle: match (all applicable rules); select (one or more rules to execute); and fire (all selected rules) (see Figure 1.3).

The process of selecting rules to fire, the conflict resolution phase, is problematic during parallel rule firing. First, conflict resolution assumes that if rules conflict, then only a single one should be executed. If it is possible to execute rules concurrently, such as when performing a parallel search, the rationale underlying

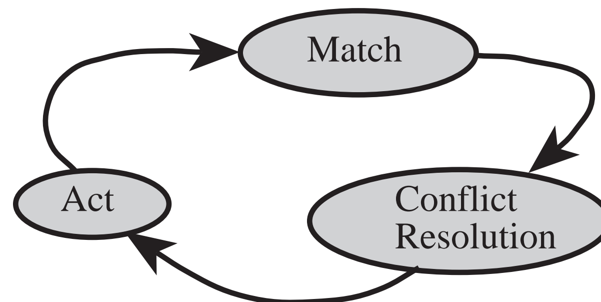


Figure 1.3: The recognize-select-act loop of conventional production system architectures.

conflict resolution should be re-evaluated. Because the process of selecting the best rule (or rules) requires that *all* eligible rules be examined, the system must achieve *quiescence* before conflict resolution can take place, where quiescence is defined as the termination of all matching activities and rule executions. But, in a parallel system, multiple concurrent multi-rule tasks may be active simultaneously and control activities for these tasks may be performed independently. Under such circumstances, achieving a global quiescence may cause eligible rules to languish within the system's conflict set for considerable lengths of time. Conflict resolution is *synchronizing* because it requires that all activities cease before it can commence. Because the actual control activities carried out during conflict resolution are typically performed by a single processor, we observe that conflict resolution is also *serializing*; during control activities, processing resources not involved in conflict resolution remain inactive. The synchronizing and serializing characteristics of the conflict resolution process are also shared by any other activity which requires discriminating a set of all eligible rules. In particular, most schemes for guaranteeing serializability require a synchronizing run-time rule interaction detection algorithm.

The original purpose of conflict resolution was to focus the attention of what is essentially a data-directed mechanism [McDermott and Forgy, 1978]. In order to reduce control costs, superficial and easily extracted characteristics of rule instances such as specificity and recency were used to heuristically select rule instances to fire. Perhaps because of the difficulty in extracting more relevant aspects of the rule instances to use in performing control decisions, rule scheduling in production systems remains relatively unsophisticated and conflict resolution is used primarily as a method for ensuring that a desired rule sequencing is achieved. Since the only way to reduce its synchronizing and serializing bottlenecks is to remove or localize conflict resolution, it is desirable to investigate alternative methods of heuristically controlling and sequencing rule firings. In addition, when productions are executed in a multiprocessing environment, a new control criterion becomes apparent; it is desirable to schedule rules in such a way that processor utilization is maximized, assuming that increased processor utilization will generate a result more rapidly. (In uniprocessor systems, this is not an issue; as long as there are rules in the

conflict set, full processor utilization can be achieved.) Thus, there are three control requirements for parallel rule-firing production systems: rules (or sets of rules) must fire in a correct sequence; rules must be selected to fire in such a way that the quality of the solution is maximized and the cost of achieving the solution is minimized; and maximum processor utilization should be obtained whenever feasible.

In summary, the research areas associated with parallel rule-firing production systems can be divided into three subproblems:

- Devising design techniques, language mechanisms, and programming idioms which will allow the creation of rule-based systems which can exploit high degrees of parallelism.

- Ensuring that these parallel rule-based systems execute correctly.

- Controlling the execution of the parallel rule-based systems to ensure the correct sequencing of rule firings, high quality/low cost solutions, and high processor utilization.

This thesis will address each of these issues in the context of UMPOPS (UMass Parallel OPS5), an experimental parallel rule-firing language based on OPS5, augmented with a lock-based scheme for ensuring consistency and a scheduler which provides multiple rule-firing policies and heuristic control mechanisms [Neiman, 1992b].

The remainder of this introductory chapter discusses the characteristics of rule-based systems that differentiate them from other parallelizable systems, outlines the contributions of my research, and provides an overview of this dissertation.

1.1 Distinguishing Production Systems from Related Architectures and Languages

Although the study of parallelism in production systems can be justified simply by the need to have these systems execute as rapidly as possible, it is worth discussing the aspects of production systems that differentiate them from similar paradigms that have been studied with respect to parallelism. The three most distinguishing characteristics of rules are their data-driven nature, their "semantic" content, and their granularity. The data-driven control flow of rule-based systems distinguishes them from conventional imperative programming languages whose control flow is predetermined at compile-time. The granularity of rule-based computation positions it between blackboard systems (which are comparatively coarse-grained) and logic programs (which tend to be parallelized at a fine-grained level). The semantic nature of rule firings distinguishes rules' activities from database queries which are not necessarily goal-directed.

1.1.1 Data-directed Programming and Imperative Languages

Parallelizing rule-based systems differs from parallelizing more conventional imperative languages in part because of the data-driven nature of the production system architectures; it can not always be determined in advance when two rules will become eligible to fire in parallel. Unlike the assertion of variables in an imperative language, the assertion (or deletion) of a working memory element has a significant duration during which inferences drawn from the contents of working memory may not remain valid. In a working memory which is in continual flux, as is the case during parallel rule-firing, it is a challenge to ensure that working memory remains consistent and the computation correct during the course of rule firing. This is particularly true because rules are activated by modifications to working memory; inconsistencies or transient errors in memory may be reflected in incorrect rule-firings.

1.1.2 Blackboard systems

When discussing parallelism, it is primarily the degree of granularity that distinguishes a production system from a blackboard system (BBS). Both paradigms are data-driven from a central data structure (either working memory or blackboard) and implementors of both types of systems must be concerned with problems of consistency, control, and resource allocation [Fennell and Lesser, 1977, Corkill, 1989, Nii *et al.*, 1989, Decker *et al.*, 1991]. The basic unit of computation in a BBS is the *knowledge source* (KS). Knowledge sources typically operate on a time scale 10–1000 times longer than that of rule executions. Because a KS will typically affect a considerable amount of data and only a single KS may modify a data element at a given time, the number of KS's which can execute concurrently may be limited unless the application contains a great deal of data parallelism. KS's are initially stimulated by preconditions and their eligibility is fully verified at run time. Because this verification requires substantial checking of the blackboard knowledge base, control, consistency, and scheduling deliberations will occupy a relatively less significant fraction of the run time in a blackboard system, making it more cost-effective to perform sophisticated reasoning activities during these phases of the computation.

1.1.3 Database systems

In many ways, rule-based systems resemble relational databases (rules resemble queries in terms of their expressiveness and working memory classes resemble relations) and efforts have been made to create rule-triggered databases and rule-based systems which can operate on large databases or which incorporate database-like operators[Delcambre and Etheredge, 1988, Gordin and Pasik, 1991, Miranker, 1990a, Sellis *et al.*, 1987, Stonebraker *et al.*, 1986, Widom and Finkelstein, 1990]. Many of the problems of maintaining consistency in a database during parallel queries and

assertions are mirrored in parallel rule-firing activities. The principal difference is that in a database, queries and assertions can be expected to be completely random (directed from the environment); and conflicts must be resolved by forbidding one or the other operation. In a rule-based system, modifications to working memory can frequently be assigned a semantics which allows conflicts to be resolved or avoided. Databases are expected to remain relatively fixed; in general, most modifications to the database are made before queries take place. Triggering rules in a database context is difficult because mechanisms must be devised to identify situations in which modifications to the database might activate rules; one common approach is to use preconditions similar to those use in blackboard systems, followed by full queries of the database. In contrast, rule-based systems are designed to be reactive, so that the changes to memory immediately trigger rule firings; there is a much tighter coupling between rule executions and modifications to memory.

1.1.4 Logic programming

Like rule-based languages, logic-based programming languages, notably Prolog, have been promoted for the representation of expert knowledge [Clocksin and Mellish, 1981, Feigenbaum and McCorduck, 1983]. There has been a considerable amount of research into the parallelization of logic programming languages [Conery, 1987, Delcher and Kasif, 1989, Lin and Kumar, 1991, Maruyama *et al.*, 1985]. One would suppose that there would be considerable synergy between research into the parallelization of logic programs and rule-based programs but this has not been the case. Much of the research into parallelizing logic programs has concentrated on the inference process, in particular, mapping backtracking into OR trees and independent clauses into AND trees. These processes are similar to those employed in rule-based programming; for example, mapping conflicting rule executions into parallel search is essentially the same as implementing backtracking as a traversal of an OR tree, while executing multiple clauses in an AND tree is similar to parallel rule-firing. In both paradigms, the fundamental control mechanism is modified due to the capability for parallel execution. However, the control and consistency issues which arise in parallel rule-firing systems and which are addressed in this thesis do not appear to have been examined in the context of parallel logic programs. This is apparently because the granularity of logical inferences is smaller than that of rules, thus allowing little opportunity for deliberation between logical inferences.

1.2 Contributions

This section provides an overview of the major contributions of this thesis. The main contribution is the assertion that rule parallelism offers the potential for speedup of at least an order of magnitude over that predicted by Gupta for match-level parallelism. Secondary contributions consist of descriptions of the

modifications to the control and design of rule-based systems which are required to achieve this level of parallelism and analyses of the influence of control and rule-interaction detection activities on the performance of rule-firing systems. The final contribution is the implementation of a parallel production system which allows predictions about the performance of parallel rule-firing programs to be empirically verified.

1.2.1 Speedup Due to Rule Parallelism

The primary reason for parallelizing a rule-based system (or indeed, any system) is to increase its performance. One of the contributions of this research has been to demonstrate near-linear speedups for some applications on a twenty-one processor shared-memory multiprocessor, thus establishing a lower limit for potential speedup due to rule-level parallelism. Analyses of the overhead required by locking and scheduling mechanisms, supported by measurements of contention for resources within the Rete net pattern matcher, control overheads, and processor utilizations indicate that another order of magnitude speedup is possible. With the advent of the new generation of large shared memory multiprocessors and virtual shared-memory multiprocessors which can support parallel activity at this level, this is a significant result.

1.2.2 Correctness

This thesis contributes to the study of correctness of parallel rule-firing systems an analysis of the overhead of guaranteeing serializability and a discussion of alternative methods for producing correct results. It is demonstrated both through analysis and experiment that, at the granularity of typical OPS5 rule executions, current algorithms for ensuring serializability will incur a time cost of approximately 10% of a rule's execution time. Even ignoring synchronization costs, this limits the available speedup due to rule parallelism to a factor of ten.

This research proposes and implements an alternative scheme which uses conventional database locks to ensure correctness for positively matched elements and design techniques and language mechanisms which allow specific program idioms to be implemented and proved correct. Although this approach does not provide the same guarantees of correctness that are attached to run-time rule detection algorithms, the locking mechanism consumes approximately 1% of a rule's run-time, increasing the potential for parallelism by an order of magnitude.

Previous approaches to ensuring correctness in parallel rule-firing systems have been based on purely syntactic features of rules. The design approach proposed here is novel in that it emphasizes the semantic role of each rule in the computation: if it is understood why rules interact, then these interactions can be avoided or demonstrated to be harmless. An example is the concept of *search*: by interpreting

competing rules as alternative operators in a state-space search, it is possible to structure the computation so as to avoid the possibility of rules simultaneously modifying or asserting the same data items. Each rule is allowed to execute in an independent partition (either real or virtual) of working memory; this reduces the interactions between the competing search paths to the single point where a solution must be selected; an interaction easily managed using locks. A language mechanism for declaring and managing multiple worlds is implemented in order to allow parallel search to be expressed easily (see Section 5.4).

1.2.3 Control

The main contributions of this thesis to the control subproblem are the development of synchronous, asynchronous, and task-based rule-firing policies, and the development of methods for implementing heuristic control without resorting to a globally synchronizing conflict resolution phase. I describe a scheduler architecture which allows rules to be assigned locks and to be executed with varying degrees of priority and I discuss how this architecture can be modified to support more sophisticated control requirements.

An asynchronous rule-firing policy allows rule instances to execute as soon as they become enabled. Such a policy is empirically demonstrated to provide a 1.5–2.0 times increase in performance over a synchronous rule-firing policy. However, firing rules asynchronously creates a number of problems.

The first problem is that of heuristic control and rule sequencing. The only method of controlling the order in which rules fire in OPS5 is through the conflict resolution mechanism. Without a synchronous conflict resolution phase, each eligible rule can no longer be compared with all other eligible rules in order to select the best to fire. Thus, control must take place incrementally as rules execute. To replace the rule sequencing mechanisms, I introduce new righthand-side mapping and iteration operators and a set-oriented production semantics which allow single rules to take the place of many sequential rule firings. To replace heuristic control mechanisms provided by conflict resolution, I introduce several control points at which rule instances may be rated, scheduled, or pruned. I discuss the implications on solution quality of both incremental and sequential control.

Not all applications are amenable to asynchronous rule execution. Because situations occasionally arise in which synchronous conflict resolution must be performed on some subset of eligible rules, a rule-firing architecture has been developed in which rule instantiations can be grouped into *task* groups. In a task group, all tasks execute asynchronously with respect to all other tasks; within a task, the rule-firing policy can be defined to be serial, parallel but synchronous with conflict resolution, or wholly asynchronous. User-defined conflict resolution routines may be attached to each task, thus allowing a convenient method of expressing sophisticated control concepts at a local level. Because tasks execute asynchronously with respect to each

other and processors are assigned opportunistically to eligible rules independent of task affiliation, local synchronization of tasks does not necessarily reduce processor utilization. It is important to emphasize that the partitioning of tasks is a control mechanism and does not necessarily guarantee that rules within one task will not affect other tasks. (The partitioning of data for tasks is discussed in Section 5.4.)

The task group construct is similar to the *contexts* of Kuo and Moldovan [Kuo *et al.*, 1991], but presents a solution to the problem of identifying local quiescence in a multiple-task environment as well as addressing the problem of managing multiple independent flows of control in a parallel rule-firing system. The emphasis on asynchronous execution of rules and tasks introduces a new and previously unaddressed problem to rule-based programming: determining when *quiescence* has occurred relative to a set of eligible rule instantiations. Quiescence is defined as the state in which no rules belonging to a task are currently executing, and no future working memory changes will affect the task by creating or removing instantiations from its conflict set. While a completely satisfying solution to the quiescence problem probably does not exist, the approach taken in UMPOPS is to select certain key elements asserted in the context of a task as indicative of local quiescence: when the key elements reach quiescence, the task is assumed to be quiescent.

1.2.4 UMass Parallel OPS5 – An Experimental Testbed for Parallel Rule Firing

UMass Parallel OPS5 (UMPOPS) was developed for purposes of experimenting with parallelism in rule-based systems. UMPOPS supports rule-level, action-level, and match-level parallelism and allows "mixed-mode" parallelism to be selectively employed by the programmer. The language is heavily instrumented to provide information about processor utilization, contention for resources, and the time required for the various phases of rule processing. This information is not only useful in evaluating the performance of the system, but can also be used to "tune" applications to achieve greater processor utilization. UMPOPS is implemented in Lisp and is therefore relatively easy to modify; this flexibility has proven valuable in the development of such features as task-based control policies, locking mechanisms, and multiple worlds. The dialect Lisp used to implement UMPOPS is Top Level Common Lisp,[2] a version of Common Lisp that incorporates constructs for invoking parallel activities and maintaining critical regions on a shared memory multiprocessor. The experiments described later in this dissertation were carried out on either a 16 or 21 processor Sequent Symmetry multicomputer.

[2]TopCl and Top Level Common Lisp are trademarks of Top Level, Inc.

1.3 Organization of the Dissertation

The research described in this thesis covers a wide range of interrelated topics, including the implementation of a parallel rule-firing language, the design of parallel applications using that language, the control issues that arise in parallel rule-based systems, and the problems of maintaining consistency during the course of concurrent rule firing. The problems of correctness and design are closely related and will be discussed in Chapter 3 while the issues underlying control of parallel rule firing systems will be covered in Chapter 4. The contents of these chapters justify the design decisions underlying UMass Parallel OPS5 (UMPOPS), which is introduced in Chapter 5. The final chapters of this thesis will describe the design and implementation of three parallel rule-firing programs of different natures; the performance of these programs will then be analyzed in terms of their potential and actual parallelism. Finally, the experiments performed using UMPOPS will be discussed in terms of their implications for parallelism within production systems. A synopsis of the chapter organization follows.

Related Work: Chapter 2 gives a brief overview of rule-based programming and the OPS5 language and describes the previous research on parallelizing rule-based systems. Other current work in this domain is compared with the approach discussed in this dissertation.

Correctness: Chapter 3 describes the problems of producing correct results and maintaining a consistent working memory during the course of parallel rule-firing. A locking mechanism is described for working memory elements that prevents pathological behaviors due to parallel modification or references to existing working memory elements. An algorithm for guaranteeing serial behavior in systems containing productions with negated condition elements is described. The performance of this algorithm is analyzed and experiments are described that empirically verify the analysis.

The high cost of this algorithm motivates the discussion in the latter part of the chapter which is concerned with the design of correct parallel programs which require only inexpensive locking mechanisms. The design philosophy is based on a taxonomy which lists the possible roles that rules may play in a computation and proposes schemes for resolving conflicts between rules based on the semantics of rule interactions.

Control Issues: Chapter 4 describes the rule-firing policies developed for parallel rule execution and argues that an asynchronous rule-firing policy should be preferred whenever possible. The problem of eliminating bottlenecks due to serial rule-firing is discussed and language modifications, including a set-oriented syntax and righthand-side iterative operators, are described which allow multiple sequential rule-firings to be compressed into a single execution whose overhead can be reduced through the focused used of action or match-level parallelism. An architecture is described that allows heuristic control to take place incrementally during the course

of asynchronous rule-firing. Techniques for maximizing processor utilization are discussed throughout the chapter in the context of sequential and heuristic control mechanisms.

UMass Parallel OPS5 Language Overview: Chapter 5 discusses UMPOPS, the experimental parallel rule-firing language developed over the course of this research. The implementation of the parallel rule-firing system is discussed, including the modifications to the Rete net to support parallelism, the modifications to the scheduler required to support multiple rule-firing policies and incremental heuristic control. The architecture of the rule-execution demons and a description of their processing of tasks at the action, match, and rule levels is described. Various suggestions for increasing the flexibility of the rule demons are included. Some of the difficult problems presented by parallel rule-firing that were discussed in previous chapters are revisited in terms of the specific language constructs that they made necessary to incorporate into the parallel rule-firing system. In particular, mechanisms for ensuring local synchronization of activities during parallel working memory modifications, and for ensuring quiescence during all levels of parallel activity are discussed. An experimental version of UMPOPS that supports search through multiple worlds using a partitioned Rete net is described, and some of the tradeoffs involved in this implementation are analyzed.

Experiments with UMPOPS: Chapter 6 discusses the design and construction of two benchmark programs, Travelling Salesperson and Toru-Waltz, as examples of programs using asynchronous rule-firing policies and heuristic control. The performance of these programs under various rule-firing policies and levels of parallelism are analyzed An extended design example is then presented; the parallelization of Alexsys, a "real-world" production system developed at Columbia University. The design principles developed throughout this dissertation are applied to Alexsys and it is demonstrated that an essentially serial control structure can be parallelized by a rule-firing policy that implements multiple tasks, each running asynchronously with respect to each other, but applying synchronous conflict resolution to rules within a task. The compromises in solution quality required by the multiple execution of tasks in Alexsys are discussed.

Conclusion: Chapter 7 summarizes the results of the experimental work associated with this dissertation, examines the potential for future work, and discusses the contributions of this research and its implications on parallel rule-based systems and other AI architectures.

Appendix A: The source code to the Toru-Waltz benchmark program.

Appendix B: The source code to the Travelling Salesperson Problem.

CHAPTER 2

Related Work

This chapter gives an overview of production systems and the OPS5 language and discusses related research in the construction of parallel rule-firing production systems.

2.1 Overview: Rule-based Systems

This section briefly describes the history of production systems and their use in the construction of knowledge-based systems. A brief synopsis of OPS5 is provided for readers unfamiliar with the language.

2.2 The OPS5 Language

The programming language OPS5 was written by Charles Forgy at Carnegie-Mellon University as one in a series of production system languages [Forgy, 1981].[1] It achieved a great deal of popularity largely because of its use in the R1 project [McDermott, 1980], its general availability as a public domain program, and its efficiency due to the use of the Rete net pattern matcher (described in Section 2.2.3). The following section provides a brief overview of the major concepts associated with the OPS5 language. A complete tutorial is beyond the scope of this dissertation, however, a complete syntactic description of OPS5 can be found in the OPS5 user's manual [Forgy, 1979] and there are several texts on programming OPS5 available, e.g. [Brownston *et al.*, 1985, Wogrin and Cooper, 1988, Sherman and Martin, 1990].

An OPS5 program consists of *rules* matching against a *working memory.* Working memory consists of a set of facts. Each fact is represented as a linear set of attribute-value pairs associated with a class, i.e. `(class ^att`$_1$ `val ^att`$_2$ `val ^att`$_3$ `val...)`. Working memory elements are created using the `make` command, deleted using the `remove` command and modified using the `modify command`. The

[1] OPS reportedly stands for Official Production System.

modify command simply does a remove which deletes the existing element followed by a `make` which recreates the working memory element with the modifications incorporated. At creation time, each working memory element is given a unique timetag which identifies that element; two otherwise identical working memory elements created at different times are assigned distinct timetags. Thus, working memory in OPS5 is represented as a *multiset* in which multiple objects of the same value may be present. A rule consists of a lefthand side (LHS) pattern and a righthand side (RHS) set of actions. (The terms lefthand and righthand side are due to the fact that productions have historically been written as LHS –> RHS). The lefthand side consists of a series of patterns which are matched against working memory. A rule is considered eligible to fire when there exists one or more sets of working memory elements such that there is one working memory element in the set for every positive pattern in the LHS, and there is no working memory element in working memory that matches any negated pattern. LHS patterns consist of conjunctions; the only way to program disjunctions (`IF A or B THEN ...`) is to code them as multiple productions.

The righthand side of a production consists of actions. These actions can consist of changes to working memory, I/O operations, or arbitrary function calls. It is assumed in this thesis that rules being parallelized contain only working memory operators in their righthand sides; I/O operations are intrinsically serial and will not be considered.

An example of a small OPS5 rule set is shown in Figure 2.1.

2.2.1 Definitions

The basic data structures and concepts involved in rule-based programming are summarized below.

Working Memory: A production system consists of a set of productions examining a set of facts which describe the current state of the system. In OPS5, this set of facts is called *working memory.* Each fact is represented by a single *working memory element* which consists of a *class* followed by a list of attributes and values. For example, a typical working memory element might have the form

```
(cat ^name Socrates ^color orange ^size large ^weight heavy)
```

Each working memory element is assigned a *timetag* which describes the order in which the working memory elements were created, and serves to uniquely identify each element. Working memory is represented as a multiset in OPS5; working memory elements which contain identical values will be stored in separate areas of memory and will be assigned unique timetags. Despite their name, timetags do not usually record the actual creation time of a working memory element, but the order in which they are created (which is not necessarily meaningful in a parallel system). The primary purpose of timetags is to uniquely identify each working memory

```
(literalize cat name state action)

(literalize see cat obj)

(literalize attack attacker victim)

;If cat is hungry and cat sees food, cat will eat food.

(p hungry_cat
  (cat ^name <cat> ^state hungry)
  (see ^cat <cat> ^obj food)
 -->
  (modify 1 ^action eat))

;If cat is hungry and cat sees critter, cat will try to eat critter.

(p hunting_cat
   (cat ^name <cat> ^state hungry)
   (see ^cat <cat>  ^obj  << pigeon duck fish >>  <victim>  )
  -(attack ^attacker <cat>)
-->
   (modify 1 ^action pounce)
   (make attack ^attacker <cat> ^victim <victim>) )

;If cat has nothing better to do, it will purr.

(p happy_cat
   (cat   ^name <cat>  ^action <> purr  )
  -(cat   ^name <cat>  ^state << aggressive hungry >> )
 -->
   (modify 1 ^action purr))

;Cats are territorial creatures.

(p aggressive_cat
  (cat ^name <cat> ^state <> aggressive)
  (see ^cat <cat> ^obj cat)
 -->
  (modify 1 ^state aggressive ^action hiss))

;Cats have no respect for expensive furniture and houseplants.

(p playful_cat
  (cat ^name <cat> ^state playful)
  (see ^cat <cat> ^obj  << chair string plant hallucination >> <victim> )
 -(attack ^attacker <cat>)
 -->
  (make attack ^attacker <cat> ^victim <victim>))
```

Figure 2.1: A “complete” cognitive model of *Felis Domesticus*

element. Because creation time is occasionally of interest to the experimenter, the parallel rule-firing system described in this dissertation has been modified to record the actual time at which elements are created.

Productions: A production consists of a lefthand side (LHS) which contains a list of patterns to be matched against working memory and a righthand side (RHS) which contains a list of instructions to be executed in the event that the production is fired.

The Lefthand Side: The lefthand side contains a list of *condition elements.* Each condition element consists of a pattern which can match one or more elements in working memory. There must be at least one corresponding working memory element for every condition element in order for the rule to be instantiated (that is, for it to be entered into the conflict set). Condition elements may be negated, in which case the rule only matches if there is *no* working memory element which satisfies the negated condition element. Condition elements may contain variables; a rule may only fire if there is a set of working memory elements which can generate a consistent set of variable bindings.

The Righthand Side: The righthand side of a production contains the operations to be performed if the rule is fired. This can contain any combination of changes to working memory, input/output statements, or function calls. The execution time of a production is equal to the amount of time required to execute all the statements in the righthand side. In general, studies of parallelism in production systems attempt to reduce this execution time by increasing the speed of the working memory changes.

The Matching Process: *Matching* is the process by which a new or modified working memory element is compared against the lefthand side of all the productions in the system in order to see if any of them are enabled by the latest change to working memory. This matching process is considered to be the most time-consuming aspect of executing a production system and considerable research has been done to determine if match time can be significantly reduced by performing the match process in parallel [Gupta, 1987].

In OPS5, the matching process takes place when working memory elements are added to, or deleted from, memory. This means that the match process actually takes place at the same time as the righthand execution phase. The implication of this is that the match and execution phase are not actually separate as the conventional description of the production system execution cycle indicates. When operating in parallel, it is important to remember that working memory may still be changing while the match process is taking place.

Conflict Set: The conflict set is the list of all the rules which are eligible to fire. A conflict set entry contains the name of the production, copies of the working memory elements which caused the production to match, a binding list which contains the values of variables bound in the lefthand side of the production, and rating information which may or may not be used when performing conflict resolution.

2.2.1.1 Levels of Parallelism

UMPOPS currently supports rule-level, action-level, and match-level parallelism. These levels of parallelism are described below.

Rule Parallelism: Rule parallelism (variously called production parallelism or application parallelism in the literature) allows multiple rules to be executed concurrently.

Match Parallelism: When match parallelism is invoked, the matching process which determines which productions are enabled by a working memory change is carried out in parallel. This reduces the amount of time required for a single working memory change to take place.

Action Parallelism: If a production contains multiple actions in its righthand side, it is possible that they may be able to be executed concurrently, thus reducing the execution time of the production by a factor proportional to the number of actions (assuming all actions take approximately the same amount of time to execute).

2.2.2 Control of OPS5 Programs

Rule-based systems are data-driven; the rules which are eligible to fire depend entirely on the state of working memory. Because (in a serial system) only one rule can execute at a time, if more than one rule is eligible to fire, the production system must perform *conflict resolution.* During conflict resolution, all eligible rules are examined and the one which is perceived to be most useful according to the conflict resolution algorithm is fired (see Figure 2.2).

As productions fire, they change working memory which in turn changes the contents of the conflict set. Therefore, conflict resolution must be performed after each production execution. Because of the lack of imperative control mechanisms, programmers of production systems frequently exploit the conflict resolution mechanism in order to obtain a specific sequence of rule executions.

Conflict resolution algorithms are typically optimized to be fast and heuristic, using only syntactic information which can be quickly accessed [McDermott and Forgy, 1978]. The conflict resolution algorithms in OPS5, `MEA` and `LEX`, are typical in this respect; they select rule instantiations based primarily on the creation time of working memory elements and the number of condition elements in the lefthand side of a production. These are known as the *recency* and *specificity* conditions; it is assumed that the most recently added elements are most relevant to the computation and that rules with more conditions are more specific and thus more likely to be appropriate to a given situation than a more general rule. The use of more sophisticated meta-rules or scheduling algorithms [Davis, 1980, Hayes-Roth, 1985] is difficult due to the inability of OPS5 to express meta-level patterns. The issue of conflict resolution and control in parallel production systems is described in Chapter 4 of this dissertation.

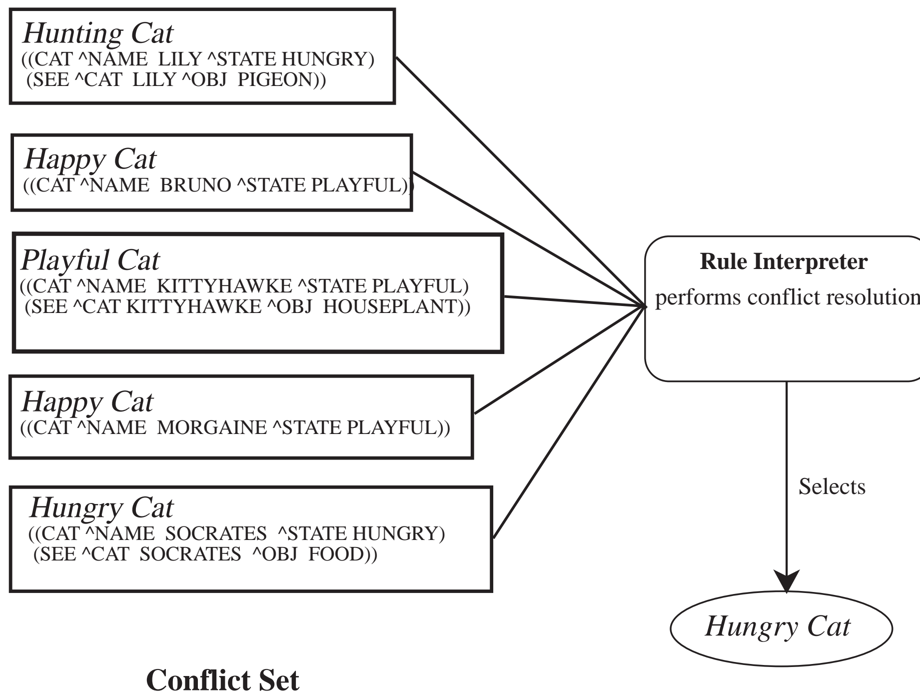


Figure 2.2: An example of conflict resolution. The "best" rule is chosen on the basis of recency and specificity.

2.2.3 The Rete Net

In production systems, most of the processing time is spent determining which rules are eligible to fire. In OPS5, this process consists of matching the lefthand sides of productions against working memory. When a set of working memory elements is found such that there is a working memory element for every non-negated condition element in the lefthand side and there exist no elements which match negated condition elements, the rule is eligible to fire. As a principal bottleneck in rule firing, this matching process should be as fast as possible. The Rete net is an efficient implementation of a pattern matcher based on the following observations:

- Working memory changes only incrementally from cycle to cycle.
- Many productions in a rule base are frequently structurally similar and may share one or more terms.

The first observation implies that it should be possible to store partial matches and only match against those working memory elements which change, rather than implementing the naive approach of comparing each production against all of working memory after each set of working memory changes. This naive approach would take $O(pw^n)$ comparisons, where p is the number of productions, w is the number of working memory elements, and n is the maximum number of elements in

a lefthand side. Sharing of tests between productions reduces the total number of comparisons that must take place.

The matching process works by passing tokens consisting of one or more working memory elements through the net, performing tests on them at each node. The "top" of the Rete net is composed of *alpha* nodes which consist of simple tests on the class of the working memory element and specific fields. This part of the network possesses no memory and resembles a conventional discrimination net; tokens are passed to succeeding nodes in the network only if the tests at the current node succeed. Alpha tests are not very time-consuming and parallelizing their execution does not lead to large improvements in performance.

Beta tests are responsible for unifying variable values between fields of a condition element (intra-element tests) or between two condition elements (inter-element tests). Each of the beta nodes has two inputs and two memories, one associated with each input. As a token arrives at a beta node, it is stored in memory and tested against the *opposite* memory to see if one or more consistent bindings can be achieved. If so, a new token is constructed from the incoming token and the stored token. This new token is then propagated through the beta node's out list (a list of successor nodes). The memories associated with the beta nodes store partial matches, making it unnecessary to repeat the entire computationally expensive unification process after each working memory modification. The cost of executing a beta node is proportional to the size of the memory against which the incoming token is tested. The two main beta node types are the **AND** and **NOT** nodes. Beta nodes present numerous opportunities for parallelism; for example, multiple beta nodes can be executed in parallel, or, if the architecture supports sufficiently fine-grained processing, an incoming token can be compared to each corresponding token in memory simultaneously.

At the bottom of the Rete net is a series of *production* nodes; when a token arrives at one of these nodes, the production corresponding to the node is placed in the conflict set, instantiated with variable bindings from the incoming token. The production node has no memory, thus only one production firing ever results from a given combination of working memory elements. Figure 2.3 shows the Rete net for the simple OPS5 example of Figure 2.1. The implementation of the Rete net is quite complex and a complete discussion is beyond the scope of this document. For a more thorough reference see [Forgy, 1982].

2.3 Research in Parallel Production Systems

Considerable research has been done on increasing the speed of production systems and of OPS5 in particular. The research divides into a number of categories:

- Increasing the efficiency of pattern matching through compilation.

- Specialized architectures customized for rule matching and execution.

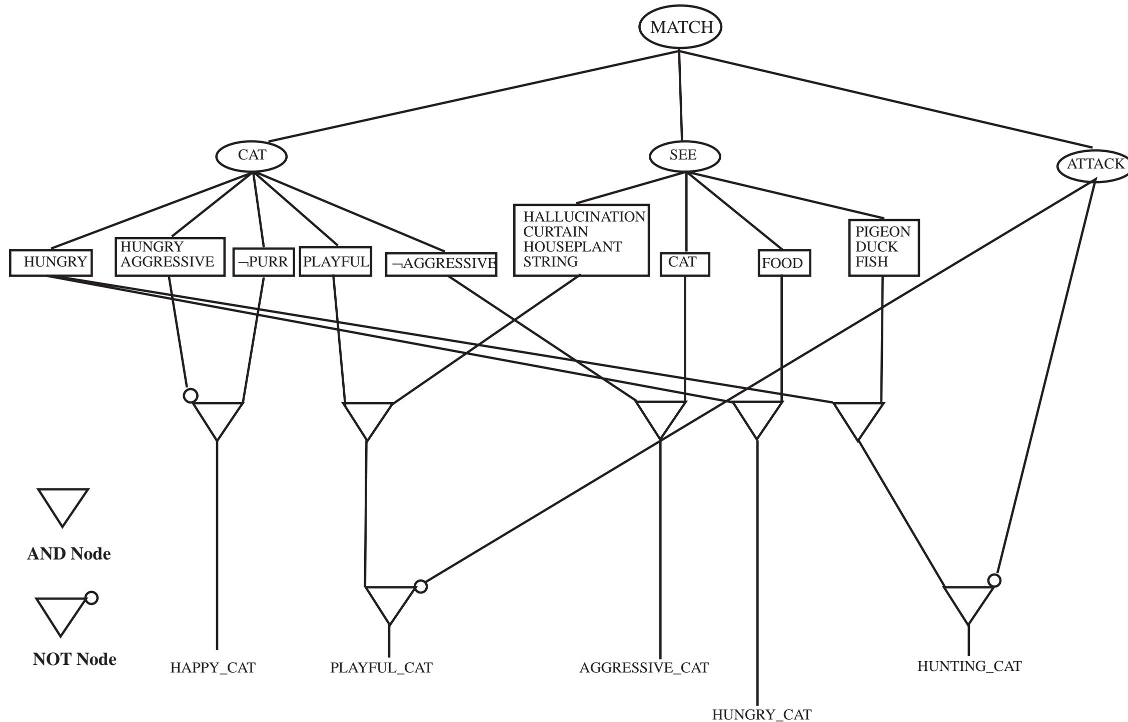


Figure 2.3: The Rete net for a simple OPS5 program

- Parallelism at various levels of granularity.

These categories are not necessarily distinct; in particular, many of the hardware architectures proposed for production systems have incorporated parallelism in their design.

2.3.1 Compilation of the Rete Net

The Rete net algorithm, as originally coded, was interpreted. The tests for a particular node were evaluated at run time and applied to incoming tokens. In his paper on the Rete net[Forgy, 1982], Forgy described an approach towards compiling the pattern matching network directly into assembly language which he has subsequently used in implementing the programming language OPS83 [Forgy, 1984]. Compilation of rules allows the production system to run substantially faster than the interpreted version. The OPS5.c compiler, a compiled version of the TREAT algorithm was reported to provide speedups of 50-200 times [Miranker, 1990b] over Lisp-based interpreted OPS5 code. CParaOPS5, a Rete-based system developed at CMU, provides roughly the same order of magnitude speedup [Gupta *et al.*, 1988]. Both of these languages are written in the C programming language for efficiency purposes.

There are disadvantages to the compiled approach to pattern matching; typically, new productions can not be incrementally added to the system, debugging of rules is more difficult (because less information about the state of the network can be accessed), and the compilation itself is time-consuming which can be disadvantageous in a development environment in which the rule set is continually undergoing modification. While compilation can reduce the time required to match productions against working memory substantially, there still exists an upper bound on the rule execution speed which can be achieved using a uniprocessor simply because the righthand side of a rule can contain an arbitrary number of actions, including relatively slow I/O operations and calls to arbitrary Lisp code.

2.3.2 Parallelism in OPS5

One of the principal studies on parallelism in OPS5 has been done by Anoop Gupta at CMU [Gupta, 1984]. In a very thorough analysis, he demonstrated that the average speedup in production systems due to parallelism would be much less than expected (on the order of 10's rather than 1000's). There is little question that this analysis holds for *existing* applications of production systems, however, it can be argued that the analysis is based principally on these existing systems, and that architectures (or applications) can be developed which provide opportunities for much greater degrees of parallelism. The unfortunate implication of this conclusion is that rather than gaining a speedup from a simple change in machines and implementation language; the programmer of the production system must explicitly think in terms of exploiting parallelism.

In his work, Gupta identifies a number of types of parallelism which can occur within the execution of a production system and analyzes their effect on the performance of the system. These types of parallelism are:

- Application parallelism
- Production parallelism
- Action parallelism
- Node parallelism
- Intra-node parallelism

The above levels of parallelism describe a hierarchy in which each level essentially implies all the levels above it. Each level of parallelism adds a certain degree of speedup to an application. Thus, the system with the highest performance would be one which employs parallelism at all levels. The following discussion briefly describes each type of parallelism and indicates the assumptions that Gupta makes when estimating the effect of that type of parallelism on the performance of an OPS5 system.

2.3.3 Production Parallelism

Gupta defines *production parallelism* as the assigning of a processor to each production and the matching of all productions affected by a working memory change at the same time. It does not employ parallelism at lower levels of the implementation. According to Gupta's analysis, the effect of production parallelism on performance is very small, in fact, approximately a factor of two. The reasons are as follows: first, only a small number of productions (26, on the average, in Gupta's test set) are affected by any one working memory change. This would seem to indicate an average speedup of 26; however, there are further factors limiting parallelism in the canonical rule-based architecture. Once the productions are matched, conflict resolution must still be performed, therefore the matching process cannot terminate until the last production has been entered into the conflict set. Because the expressions which determine whether rules can fire can be arbitrarily complex, and because matching is performed by propagating tokens through a potentially unbalanced tree-like data structure, the time required for productions to match is typically not uniform[2]. Some productions enter the conflict set considerably later than other instantiations enabled by the same changes to working memory. This causes a loss of parallelism of a factor of five. Additional losses are incurred because of loss of sharing in the Rete net and the overhead due to parallelism.

Several of these assumptions can be questioned. In situations which possess *action parallelism* (see below), the number of working memory elements and thus the number of productions affected might be much larger. In an asynchronous system, it may not be necessary to examine the conflict set before executing productions, in fact, the conflict set may turn out to be unnecessary (or undesirable) in a parallel system. Without the conflict set bottleneck, an additional factor of 5.1 (according to Gupta) could be obtained.

2.3.4 Node and Intra-node Parallelism

In *node* and *intra-node* parallelism, the execution of each node in the Rete net is assigned to a separate processor. If a node in the network has multiple descendants, all of the subsequent nodes can be evaluated concurrently (see Figure 2.4). If a particular working memory change can affect many productions then the branching factor of the associated nodes will be high and so will the speedup provided by node parallelism.

Node parallelism assumes that no other process affects the data of a particular node while it is executing. Intra-node parallelism allows the same node to be executed by several processors simultaneously; this gain in parallelism avoids long delays during which access to a node must be restricted, but incurs a cost in terms

[2]The match time can be made to be more uniform by identifying "hot spot" rules and creating multiple constrained copies of these rules [Pasik and Stolfo, 1987].

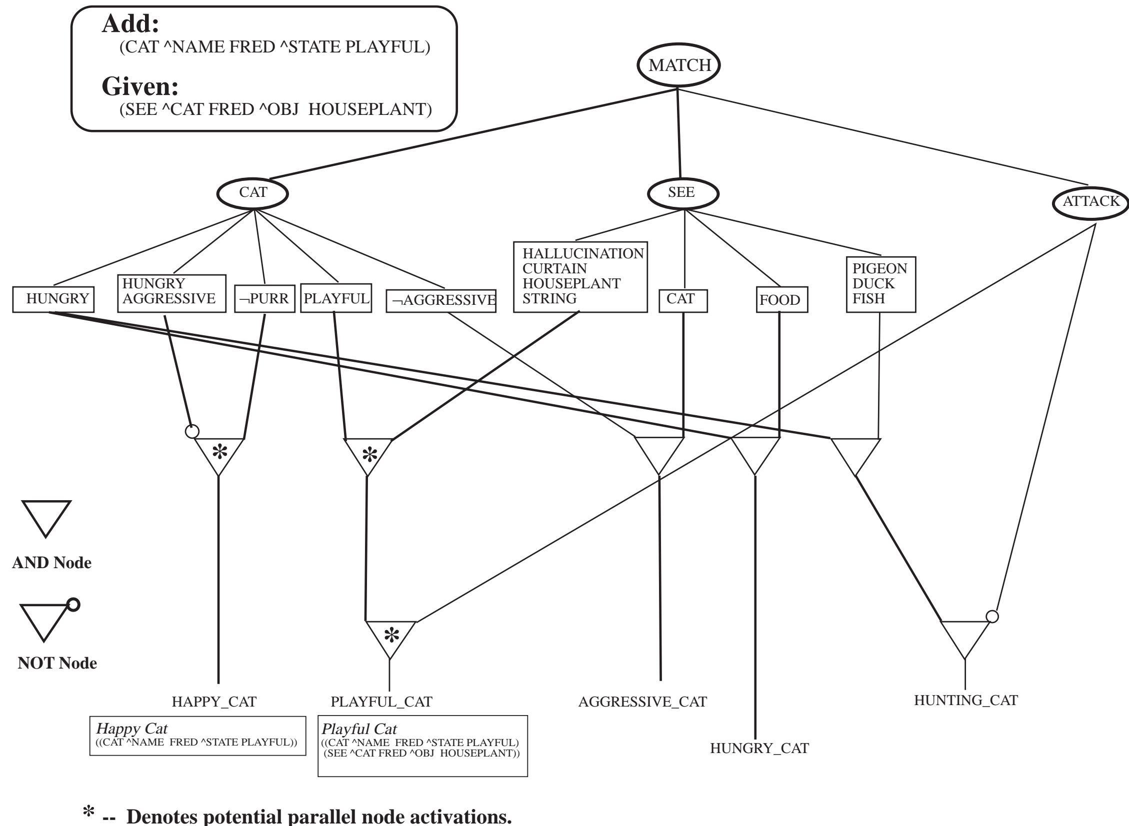


Figure 2.4: Match-level parallelism for a single working memory change in the cat example.

of contention for the resources of the node and problems in keeping the memory of the node consistent during parallel memory accesses. In his analysis of node and intra-node parallelism, Gupta assumes the fairly simple pattern matching operations available in OPS5. In OPS5, the number of operations computed at a given node is only an order of magnitude greater than the overhead required to schedule and execute parallel processors. Very little computation occurs within a particular node, perhaps fifty to one hundred instructions, on average. In order to be beneficial, parallelism must be achieved with very little overhead.

The speedup gained by node parallelism alone is therefore only marginal and acquired only by optimizing the scheduling mechanism both in hardware and software. If the complexity of computations performed at each node increases, the advantage to be gained from node parallelism also increases. It might be possible to increase this complexity either by enhancing our pattern matching language to allow more sophisticated expressions or by greatly increasing the size of the memories stored at each node, a natural consequence of applying production systems to applications requiring very large databases.

2.3.4.1 *Extremely Fine Grained Parallelism within the Rete Net*

When a token enters a two input AND node, it is compared against all elements in the opposite memory. This operation can certainly be executed in parallel, however the tests are so simple that parallelism can only be beneficial at the finest levels of parallelism. This approach is taken by Kelly and Seviora using DRete on the CUPID architecture [Kelly and Seviora, 1989]. While their experiments indicate a very high degree of speed up in the matching process, it is not clear that a corresponding increase in speed would be achieved in a full implementation.

2.3.5 *Action Parallelism*

Action parallelism is the changing of multiple working memory elements simultaneously. In OPS5, action parallelism is equivalent to executing all the elements of the righthand side of a production in parallel, or executing the righthand sides of multiple productions in parallel. By allowing action level parallelism, the number of productions affected per matching cycle increases, as well as the number of node activations (see Figure 2.5). With the use of action parallelism, the potential for increasing speedup within the match process is proportional to the number of rules executing concurrently and the average number of righthand actions in each rule. The level of programming complexity encountered by both the OPS5 implementor and the OPS5 programmer in systems which allow action parallelism is considerable, At the implementation level, there are all the problems of intra-node parallelism as well as possible consistency errors due to race conditions in the network and multiple simultaneous writes to memory nodes. At the programming level, there are the problems of non-serializable behavior, that is, behavior which could not have been achieved had all the working memory changes taken place in a serial order. To avoid implementation level errors, the system must provide mechanisms for locking memory nodes and synchronizing conflicting actions, as a result the contention for resources within the net can potentially degrade system performance drastically as the number of parallel actions increases.

2.3.6 *Application Parallelism*

Gupta notes that the Soar architecture appears to be capable of a type of parallelism which he calls *application parallelism* in which all rules in the conflict set are allowed to execute concurrently (see Figure 2.6). (Application parallelism is equivalent to what I have been calling rule parallelism.) Gupta speculated that the Soar architecture might provide large degrees of production parallelism and high degrees of parallel node activation because all instantiations which enter the conflict set are executed without conflict resolution. This property of Soar has not yet been extensively studied.

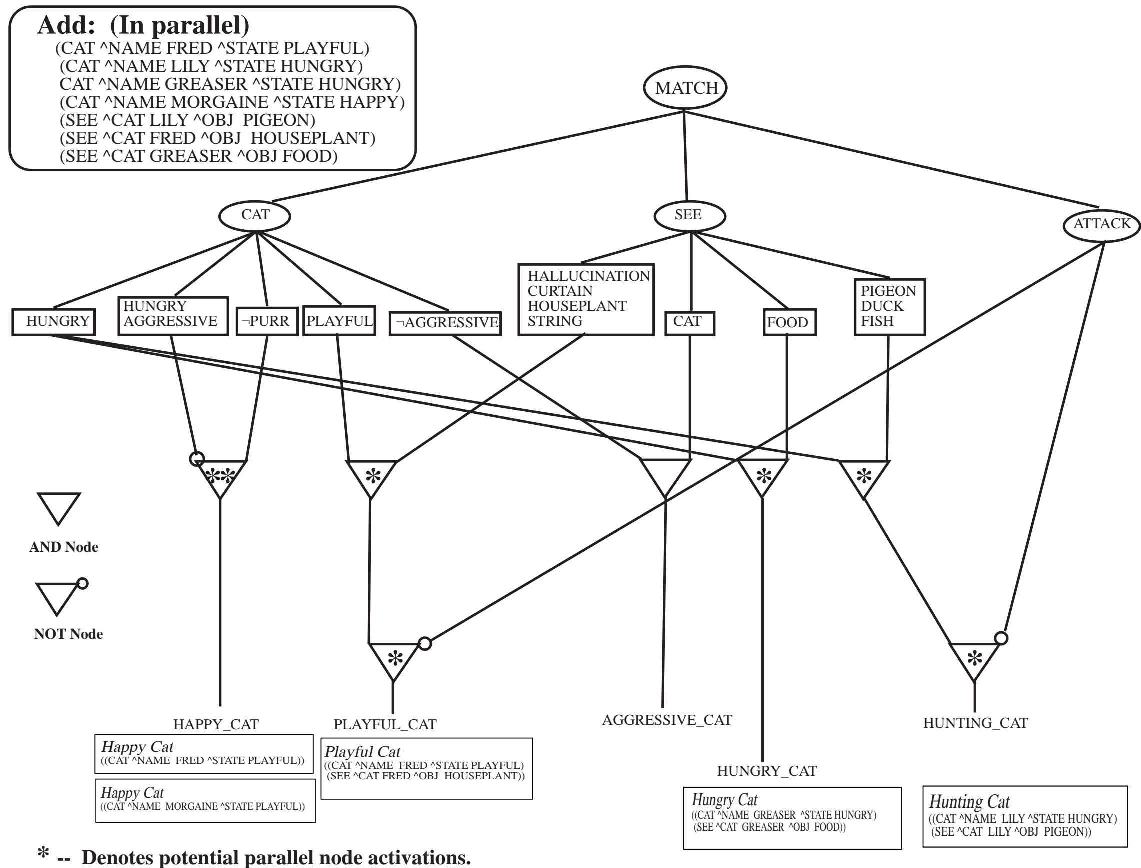


Figure 2.5: Action parallelism combined with node parallelism greatly increases the number of concurrent node activations.

In OPS5, the righthand side of productions contain mostly modifications to working memory, therefore application parallelism is similar in nature and effect to action parallelism except that the concurrently executing productions need not be referring to, or modifying the same working memory elements. Therefore, problems of contention for resources in the Rete net are reduced and the potential speedup in the matching process is significantly increased. The speedup offered by application parallelism is directly proportional to the number of productions which can perform useful tasks while executing concurrently. This number is dependent only on the application; a system which is executing a large number of loosely coupled tasks may be able to maintain a very high level of rule activations.

2.4 Parallel Execution of Rules

Ishida and Stolfo were among the first to study parallel rule-firing in depth [Ishida and Stolfo, 1985]. They described two major problems in executing all

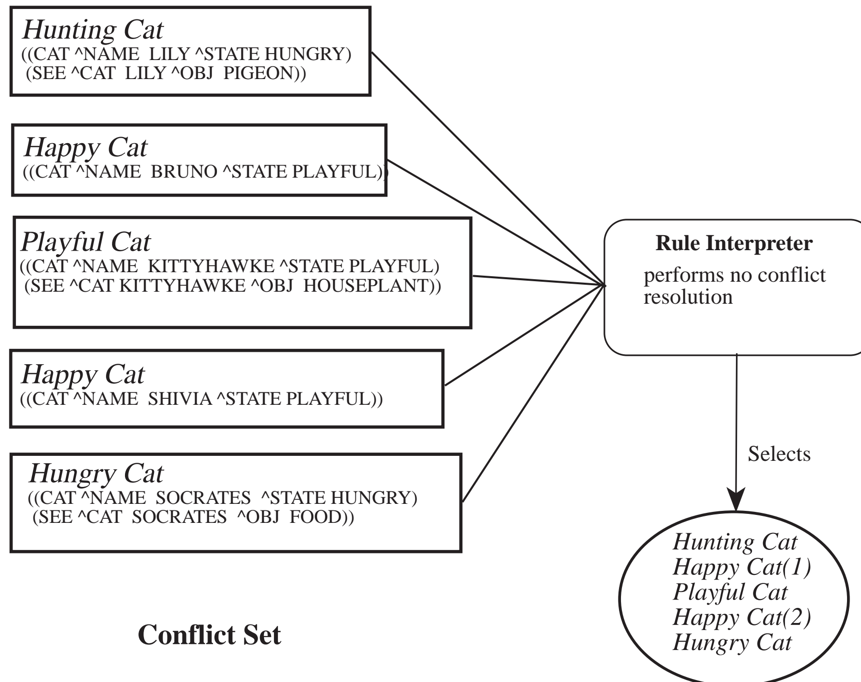


Figure 2.6: Rule parallelism allows all the instantiations in the conflict set to be executed concurrently.

rules in parallel: synchronizing concurrent production firings to avoid interference between productions, and decomposing problems to achieve maximum parallelism. They propose algorithms for detecting interference. Tellingly, these algorithms produced disappointing results on most benchmarks until the benchmarks were rewritten in a less serial form. On one (rewritten) problem, they report an expected speedup of 7.5 on a 32 processor system, not including possible speedups due to node and intra-node level parallelism. The algorithms derived by Ishida and Stolfo for detecting interactions between productions are static, and are performed on the rule base before execution. In work which builds upon that of Ishida and Stolfo, Schmolze has developed three algorithms which contain both static and runtime components and more precisely determine when rules can co-execute without violating serialization constraints, thus producing larger subsets of productions which can be co-executed [Schmolze, 1989].

2.4.1 Achieving Serializable Behavior in a Parallel Program

When productions run in parallel, the possibility exists that they will interfere with each other. Schmolze identifies two types of rule interactions, disabling and clashes. A production *disables* another production when it causes a change

in working memory which removes the second production from the conflict set.[3] Productions *clash* when they cause conflicting changes in working memory. For example, when production A adds a working memory element, and production B deletes it, the final state of working memory depends on the order in which the productions fire.

If productions can interfere with each other, then the results of running them in parallel will not necessarily be the same as running them serially, and the answers achieved by such a system may not be deterministic. In order to guarantee serializable results, Schmolze develops algorithms which analyze rules for potential disabling/clashing behavior and uses this information to *synchronize* the conflict sets. A conflict set is said to be synchronized if it possesses no instantiations which can either clash with or disable each other. Schmolze reports on three algorithms of varying precision for identifying rule sets which can be synchronized. Each algorithm has a static phase which examines the rule set, and a runtime *Select* phase which processes the conflict set and produces a subset of co-executable productions. The trade-off between the algorithms is between speed and precision. Static analysis is imprecise, because values have not yet been determined for many of the variables used in the matching process making it impossible to determine all the possible non-serializing relationships. A static analysis, therefore, must err on the side of safety and prevent rules from co-executing which only *potentially* interact. An algorithm which examines actual instantiations within the conflict set at runtime can be more precise, but dynamic detection of serialization violations increases the cost of conflict resolution and thus reduces the speedup obtained from parallel execution of the productions.

While the dynamic analysis of the conflict set does allow potentially co-executable productions to be precisely identified, it also limits potential parallelism in that it requires that an instantiation be compared with all other instantiations in the conflict set. This implies that the system must achieve quiescence, that is, that there be no matching taking place during the Select process. The quest for the maximum synchronization sets of executable productions prohibits asynchronous rule execution.

One of the major motivations behind the research described in this thesis has been to reduce the overhead required by runtime interaction detection algorithms. This subject is discussed further in Chapter 3.

---

[3] Strictly speaking, it is an *instantiation* of a production rather than the production itself which is placed in the conflict set, but for brevity, I'll use "production" or "rule" to mean "instantiation of a production/rule" when it won't cause confusion.

2.5 Parallel Rule-firing Production Systems

A number of researchers have been developing parallel rule firing production systems; this section will discuss these projects and compare them with the architecture and assumptions behind UMPOPS.

2.5.1 The CREL System

CREL (Concurrent Rule Execution Language) [Kuo *et al.*, 1991, Miranker *et al.*, 1989] is a parallel rule-firing system based on the OPS5.c language. OPS5.c is semantically identical to OPS5 except that it is written in C and employs the TREAT pattern matching algorithm. CREL adopts a non-deterministic rule firing paradigm in which rules are executed in parallel without a conflict resolution phase. Correctness (where serializability is used as the correctness criterion) is enforced by a combination of intensive static compilation and program transformations and by dynamic run-time checking. The static compilation partitions the rule set into a number of data-independent rule *clusters*. Each cluster executes independently and asynchronously with respect to all other clusters. Clusters communicate via message passing. Rule parallelism is allowed within an individual cluster, however, because rule interactions within a cluster cannot be eliminated through static analysis, dynamic runtime checking must be performed to insure that rules will not interact. Optimizing transformations are used to increase the discriminating power of the static compilation; however these transformations are largely syntactic and do not yield a great deal of parallelism. For example, the "control variable smart" transformation partitions rules into clusters based on the "secret messaging" or mode-changing condition elements contained in the rules. Because these secret message elements are largely used to sequence steps in a computation, the clusters, though independent, may never actually execute in parallel. The primary advantage obtained through clustering in CREL is reduction of the overhead of dynamic run-time checking during multiple rule-firing within a cluster; disappointingly, clusters rarely execute concurrently.

The *copy and constrain* transformation [Pasik and Stolfo, 1987] used in CREL is more useful in extracting parallelism; this transformation assumes that the values which can be assumed by particular variables in the LHS are enumerable. Multiple copies of rules are created with the variables replaced by constants from the set of possibilities; because it can be determined at compile time that these rules will not interact, they can be assigned to separate clusters. The copy and constrain transformation also has implications in match parallelism (which is also supported by CREL) as the multiplication of rules reduces the potential for the so-called "hot spot" rules which consume excessive amounts of match time. The problem with the copy and constrain transformation is that it literally creates new copies of rules, discriminated by the substitution of constants for specific variables. Thus, this transformation will increase the size of production memory and create a potentially

large number of semantically identical rules. Copy and constrain is not employed in the UMPOPS system; a combination of match-parallelism and memory hashing reduces the overhead of the hot-spot rules while rules whose instances are known in advance to match independent and discrete variables can be specified as executable without locking overhead.

The principal conceptual contributions of CREL are the elimination of conflict resolution in favor of a non-deterministic rule-firing policy and the notion of asynchronously executing independent rule-clusters. The major deficiency in the CREL research is the reliance on primarily syntactic features of individual rules rather than a high level treatment of rule semantics when determining rule clusters. All that can be said about rules coexisting in a single cluster is that they will not disable or clash with rules in any other cluster.

2.5.2 PARULEL

The PARULEL (PArallel RULE Language) system under development at Columbia University [Stolfo *et al.*, 1991b, Stolfo *et al.*, 1991a] is the result of a research program to design an inherently parallel language without the implicit sequentialities imposed by conventional production system architectures. The designers of PARULEL reject conventional conflict resolution control mechanisms in favor of a *meta-rule* oriented approach to control. In general, it is assumed that all rules which enter the conflict set are eligible to fire in parallel. However, before execution, the state of the conflict set is itself entered into a "meta-working memory" where it is examined by a set of meta-rules. These meta-rules are designed by the programmer to distinguish those rules which might interact or which should not execute concurrently. These meta-rules *redact* or eliminate offending rules from the conflict set. The advantage of the meta-rules is that the definition of "conflict" is left up to the programmer and thus the detection of rule interactions can be based on more than simple syntactic features of rules (given that a suitable vocabulary for expressing conflicts is provided). In order for meta-rules to be able to reference the state of the conflict set, PARULEL (which is otherwise syntactically similar to OPS5) is augmented with a syntax for building redaction meta-rules; these rules consist of tests for the existence of instances in the conflict set and predicates describing possible relationships between these instances.

Simulations of the performance of the PARULEL system on various benchmarks indicate that the meta-redaction model provides considerable parallelism, however this parallelism is measured only in reduction of production cycles and not actual run time. A number of questions remain concerning the actual performance of the PARULEL system. The first is a question of synchronization. The control flow of the PARULEL system is described as:

1. Match all rules to construct the conflict set.

2. Redact conflict rule instances.

3. Fire all rules remaining in the conflict set.

Although control meta-rules are allowed to fire asynchronously as rules enter the conflict set, it appears that a system-wide quiescence must be obtained before it can be guaranteed that no rule interactions will occur. Thus, PARULEL may exhibit the same synchronization overhead seen in previous architectures. While the use of meta-rules is an elegant and parsimonious solution to the problem of rule interactions (parsimonious because it uses the same basic rule interpreter to perform control as well as to execute rules), the amount of computation which must be performed to ensure that no rules interact is still $O(N^2)$ for N instances in the conflict set. Thus, it is not clear that the overhead of redaction will be any less than that encountered in more conventional run-time interaction detection algorithms such as [Schmolze, 1991].

The principal contribution of PARULEL, then, is its abandonment of the conventional sequential paradigm imposed on rule-based system architectures. It should be noted that the PARULEL system is just one component in a larger project, PARADISER, which provides a complete environment for parallel execution including mechanisms for incrementally performing nonmonotonic changes to the database during execution [Wolfson *et al.*, 1990] and mapping rule executions to processors to ensure that load balancing is achieved.

2.5.3 Control of Rule Sequencing

Most investigations of parallel rule-firing have concentrated primarily on serializability as a criterion for correctness. In their research with the RUBIC (Rule-Based Inference Computer) system [Moldovan, 1989], Kuo and Moldovan develop additional criteria for correctness – not only *compatibility* but also *convergence* [Kuo and Moldovan, 1991]. The parallel rule firing system is required to not only produce the same result as *some* sequential firing of rules, but it is required to produce the same result as a controlled sequencing of rules. They define the notion of *context*: rules are allowed to fire in parallel in a context if they are both compatible (produce serializable results) and are guaranteed to reach a correct solution. In problems which are non-convergent (not guaranteed to reach a correct solution), sequential conflict resolution is applied within that context. In the Single-Context-Multiple-Rule (SCMR) model, only a single context is active at any given time, but multiple rules may fire within a context. Dynamic rule interaction checking (using a parallelism matrix computed at compile time) is used to insure that rules are compatible. In the Multiple-Context-Multiple-Rule model, multiple contexts are allowed to be active simultaneously. Contexts execute asynchronously with respect to each other.

The RUBIC system operates in a distributed multicomputer environment (the Intel iPSC/2 hypercube). Thus, results and algorithms are not directly comparable with those produced for shared memory multiprocessors. However the MCMR model

proposed by Kuo and Moldovan is quite similar to the multiple asynchronous task architecture with "control-variable-smart" transformations proposed by Miranker. The principal difference is the explicit discrimination between convergent contexts which do not need conflict resolution and sequential contexts which require serial conflict resolution in order to reach a correct solution.

2.5.3.1 Parallel Rule Firing with Fuzzy Logic

An alternative (and rather elegant) approach to resolving conflicts between executing productions has been taken by Siler, et al. in the programming language FLOPS (Fuzzy Logic Production System) [Siler *et al.*, 1987]. In the FLOPS system, all eligible productions are executed concurrently. There is no conflict resolution and no backtracking. Instead, a *memory conflict* algorithm is employed which resolves contradictions in memory using "weakly monotonic" fuzzy logic. Each rule generates both values for attributes and confidence levels. If a rule produces an attribute value with a confidence level greater than or equal to the existing value, then the previous attribute value is replaced with the most recent value. Naturally, this approach depends on the ability to generate meaningful and accurate confidence values. In order to ensure program correctness, parallel rule firing with fuzzy logic still requires that the state of working memory be independent of the order in which rules are executed.

2.5.4 A Note on Rule versus Instance Parallelism

A number of research efforts, particularly, but not necessarily, those involving implementations on distributed multicomputer systems, e.g. [Oflazer, 1984, Tenorio and Moldovan, 1985, Kuo and Moldovan, 1991, Ishida *et al.*, 1990, Xu and Hwang, 1991] allow only a single instantiation of each rule to execute at a given time. Each rule is explicitly assigned to a given processor, and only that processor will be able to execute that rule; I will call this "strict" rule parallelism, because only distinct rules are allowed to fire concurrently. Such an architecture creates what is known as the *partitioning problem*; rules must be assigned to processors in such a way that rules which are capable of executing concurrently are not assigned to the same processors, otherwise concurrency will be reduced. In general, rule parallelism of this type will require generation of many almost identical rules in order to create acceptable levels of concurrency.

From a conceptual point of view, strict rule parallelism (as opposed to instance parallelism in which many instantiations of the same rule are allowed to fire concurrently) seems contrary to the spirit of production systems. A rule theoretically represents a unit of knowledge which can be applied when a subset of working memory achieves a given configuration, and it seems unreasonable to limit the application of expert knowledge to one set of data at a time. Instead, it seems

far more reasonable to be allowed to apply the same "quantum" of knowledge to data objects at the same time.

Although this is speculation, I believe that strict implementations of rule parallelism are an artifact of two historical factors, the limited memory available on early distributed multiprocessors which made it infeasible to distribute the entire knowledge base to all agents, and the initial discussion of rule parallelism by Ishida and Stolfo which assumed that only a single instance of each rule would be allowed to execute. Neither of these factors is a convincing reason for continuing the tradition of strict rule parallelism; as discussed previously, recent work on guaranteeing serializability has demonstrated that correctness can be maintained despite parallel firing of instances, while technological advances have made it feasible to substantially increase the size of the memory allocated to distributed processors. (Strict rule parallelism has never made any sense on shared memory architectures as the entire rule set is accessible to all processors and rules are assigned to processors on a purely opportunistic basis.)

The benchmarks that have been developed for testing parallel rule-firing systems uniformly display very high levels of instance parallelism and low levels of strict rule parallelism. Thus, if memory is limited, for whatever reasons, it makes more sense to distribute the entire rule set (or relevant subset of rules) to all processors and partition according to distribution of working memory elements.

2.5.5 Architectures for Production Systems

The previous section described a number of algorithms for incorporating parallel processing into production systems. Pragmatically, the algorithm chosen depends almost as much on the available hardware as it does on the inherent parallelism within the problem area. A number of machine architectures have been proposed for the rapid execution of production systems; they range from uniprocessors to machines with thousands of processors which support extremely fine-grained parallelism. There is by no means universal agreement on the correct degree of granularity for these architectures; the questions of how many processors, and how powerful, are closely tied to the degree and location of maximum potential parallelism within the production system. This section will discuss a number of proposed architectures for executing production systems.

2.5.5.1 DADO

The DADO machine [Stolfo and Miranker, 1984] was an attempt to develop a parallel tree-structured architecture which would efficiently execute expert system programs. The prototype DADO2 machine had 1023 8-bit processors *each* producing approximately 0.5 MIPS. The tree architecture minimizes communication costs; each node is responsible for transmitting to the nodes immediately below it, and propagating results from lower nodes upwards through the tree. Each node was

implemented using a microprocessor with a small (16K) amount of memory. The DADO architecture can operate in either a semi-SIMD mode (in which the single instructions are function calls rather than machine language calls) or MIMD in which nodes execute autonomously. The DADO architecture can apparently support most levels of parallelism present in rule-based systems by assigning tasks to different levels of the hierarchy. To implement production parallelism, each production is assigned to a processing element (PE) at a fixed level of the tree. Processing elements below the PE assigned to production matching are assigned to specific working memory elements. In the ideal case, the DADO architecture should produce matches independent of the number of productions or working memory elements. Because it is unlikely that there will be enough processors to map to each production and working memory element, multiple assignments can be made, causing some decrease in performance. The algorithms available for DADO can be tailored to the nature of the production system program being executed. For example, some programs may not have a significant amount of production parallelism but may contain rules with extremely large lefthand sides; in such cases, more processors might be allocated to matching working memory [Stolfo and Miranker, 1984].

There has been a certain amount of controversy regarding the DADO architecture, particularly whether the power of the large number of DADO processors could be utilized given the properties of OPS5 programs as analyzed by Gupta [Gupta, 1984, Stolfo, 1984]. The results of the DADO2 project have been reported in [Stolfo, 1987]. Work is now proceeding on DADO4, an architecture which comprises 15 high-speed 16-bit RISC processors each running at approximately 12.5 MIPS.

2.5.5.2 Implementation of OPS5 on Non-Von

The Non-Von architecture, a massively parallel multiple-SIMD machine developed at Columbia University, has also been considered as a vehicle for executing OPS5 [Hillyer and Shaw, 1988]. The key to Non-Von's performance is the heterogenous nature of its architecture. Working memory elements are assigned to small processing elements (SPEs) and operations which refer to attributes of the working memory elements are performed associatively. Operations at a greater level of granualarity are carried out in the large processing elements (LPEs). The architecture contains a large number of SPEs (on the same order as the average number of working memory elements in the standard production system), and a much smaller number of LPEs (approximately 32). Benchmarks based on simulations of the Rete algorithm using data gathered from existing expert systems promise upwards of 850 production executions per second as compared to 1 to 5 on a Lisp-based interpreter running on hardware of equivalent cost; at the time of this research, this would have been a VAX 11/780. Whether this performance would actually be achieved by a working prototype is not known, as the project has been discontinued.

2.5.5.3 *CUPID and DRete*

Another approach to fine-grained parallelism has been taken by Kelly and Seviora with the distributed Rete (DRete) algorithm designed for the Cupid architecture [Kelly and Seviora, 1989]. This architecture consists of a matching processor networked to a host. The host performs conflict resolution; the matching processor performs the matching actions. The CUPID architecture consists of a large number of small processors. The underlying approach is that of very fine granularity. Each beta node in the Rete net has to perform a number of comparisons proportional to the number of tokens in that node. The DRete algorithm splits each node so that a copy exists for each token stored in that node's memory. This allows each comparison to be performed on each node in parallel, thus allowing each beta node to proceed in essentially unit time. There is, however, an overhead associated with generating new copies of nodes for new tokens as they are propagated through the net. The effectiveness of the DRete algorithm increases as the number of tokens stored in each node increases.

2.5.5.4 *Message Passing Architectures*

Multiprocessors with distributed memory are not ideally suited for executing production systems because the communication costs required to transmit updates to working memory largely eclipse the advantages gained by parallel processing at the node levels. These architectures are most suited for large grained parallelism in which each processor contains its own working memory and productions and works on separate tasks. Communication costs are decreasing, however, and distributed memory architectures are becoming more effective. These architectures are particularly attractive because they provide more processors at less cost than the more expensive shared memory machines. Research on executing production systems on message passing computers is described in [Tambe *et al.*, 1989, Acharya and Tambe, 1989, Schmolze and Goel, 1990, Acharya *et al.*, 1991].

2.5.5.5 *Current Trends in Implementation Architectures*

Although research into special-purpose architectures for production systems is continuing, the availability of powerful general purpose parallel processors has prompted a trend to move towards commercially available systems. An advantage of this approach is that it allows production systems to be written using currently available state-of-the-art technology without the requirement for massive hardware development projects. Architectures such as the Connection Machine which employ very large numbers of processors provide opportunities for performing research on applications of fine-grained parallelism to production system matching [Hillyer and Shaw, 1988, Morgan, 1988]. Because of the SIMD nature of such processors, the mapping between the architecture and the match process is not straightforward,

particularly if the lefthand side employs variable binding and unification. Perlin solves the problem of variable binding by enumerating all possible values which could be achieved by a variable during the matching process and assigning processors accordingly [Perlin, 1989].

The work by Gupta predicted fairly low levels of concurrent node activation and a relatively high overhead associated with scheduling fine-grained parallelism. The conclusion reached was that the preferred architecture for executing a parallel production system (based on studies of existing systems) is a shared memory system containing no more than 64 high-speed processors augmented with a hardware scheduler for allocating processors to node activations. The research presented in this dissertation provides additional evidence that shared-memory architectures are an appropriate choice for the implementation of parallel rule-firing production systems. With the advent of a new generation of virtual shared-memory machines, the limitations imposed by the limited bandwidth of the shared memory bus can be sidestepped; although such architectures may require new compilation and processor allocation strategies to attain full efficiency.

2.6 Conclusion: Related Work

This chapter has provided an overview of the current state of the research in parallelizing rule-based systems. Match-level parallelism has been the most intensively studied application of parallelism to date, and a number of special purpose architectures have been proposed which can potentially support several thousand working memory changes per second. But the research by Gupta has demonstrated that match parallelism alone will not yield significant performance improvements. Current research has therefore turned to examining the potential for increasing concurrency in production systems by executing rules in parallel. A number of these research programs including CREL, PARULEL, and RUBIC were discussed. The following chapters will discuss my own contributions to this area and my approaches to controlling rule executions and maintaining correctness during the course of parallel rule firing.

CHAPTER 3

CORRECTNESS AND DESIGN

One of the principal problems underlying the parallel execution of rules is maintaining the correctness of results and the consistency of working memory during the course of concurrent and possibly interacting rule firings. The most frequently used criterion for correctness is serializability, that is, whether a result produced by a parallel execution could have been produced by any serial execution. Kuo and Moldovan [Kuo and Moldovan, 1991] and Srivastava [Srivastava and Wang, 1991] add a more rigorous criterion: the result produced must be equivalent to that produced by a serial rule firing algorithm incorporating a specific conflict resolution policy.

This chapter gives a brief overview of approaches to maintaining correctness and consistency during the course of parallel rule firing and describes the approach I take in UMass Parallel OPS5: a read/write locking scheme for working memory elements. Because the approach suggested does not guarantee serializable programs, this thesis makes the somewhat weaker claim that correct programs can be *designed* using the working memory locking scheme. The design philosophy suggested is to interpret potential rule conflicts in terms of the semantics of that interaction. That is, if one assumes that a rule is fulfilling a role in a high-level computation, then there must be a reason for any changes that it makes to working memory. Conflicts between rules can be resolved by identifying and preferring the rules whose actions are most appropriate in the context of the high-level computation. In most programs, potential rule interactions will take place at well-defined points in the computation and mechanisms to resolve conflicts can be constructed that do not impact on unrelated rules.

The second half of this chapter is devoted to discussing this design philosophy in terms of common usages of rules and the functions of the working memory elements that they create. The role of rules in several high level computational models such as search and inference are discussed, and methods are suggested for avoiding or minimizing the cost of rule interactions.

3.1 Correctness and Serializability

Rules that are executed in parallel may interfere with each other, leading to a working memory state that could not have been produced by any serial execution

(A 1) (B 1) (D 1)

P1

P3

(A 1) (B 1)
(C 1) (D 1)
No P3 match
if P1 is executed first.

P1 & P3

(A 1) (B 1) (E 1)

P1

(A 1) (B 1)
(C 1) (D 1)
(E 1)

If P1 and P3 execute concurrently,
the result must be the same as the
one possible serial execution order.

P1: +(A <x>), -(C <y>) -> +(C <x>), +(D <x>).
P3: +(B <x>), -(C <y>) -> -(D <x>), +(E <x>).

Figure 3.1: If the concurrent execution of rules can produce a result which is produced by any sequential execution, the program is said to be *serializable*.

of those same rules. Such parallel executions are said to be *non-serializable* (see Figure 3.1).[1] *Disabling*, one such type of interference, occurs when two or more rules mutually disable each other, where a rule disables another if and only if executing it causes the other to no longer match. Given a set of rules to execute in parallel, disabling will cause a non-serializable result iff there is a cycle of disabling relations among the rules (see Figure 3.2). *Clashing* behavior occurs when two rules perform competing modifications to the same working memory element, i.e. one rule potentially deletes a working memory element that is added by another (see Figure 3.3). Because OPS5 does not enforce a unique representation for working memory elements (working memory is a multiset, not a set) clashing behavior *per se* cannot cause non-serializable effects. Instead, clashing rules may create redundant instances of working memory elements leading to spurious rule firings and subsequent explosive growth of working memory size.

[1]The diagrams dealing with serializability in this chapter are adapted from those in Schmolze's paper on guaranteeing serializability[Schmolze, 1991]. The diagrams have been modified to correspond somewhat more closely with the syntax of OPS5.

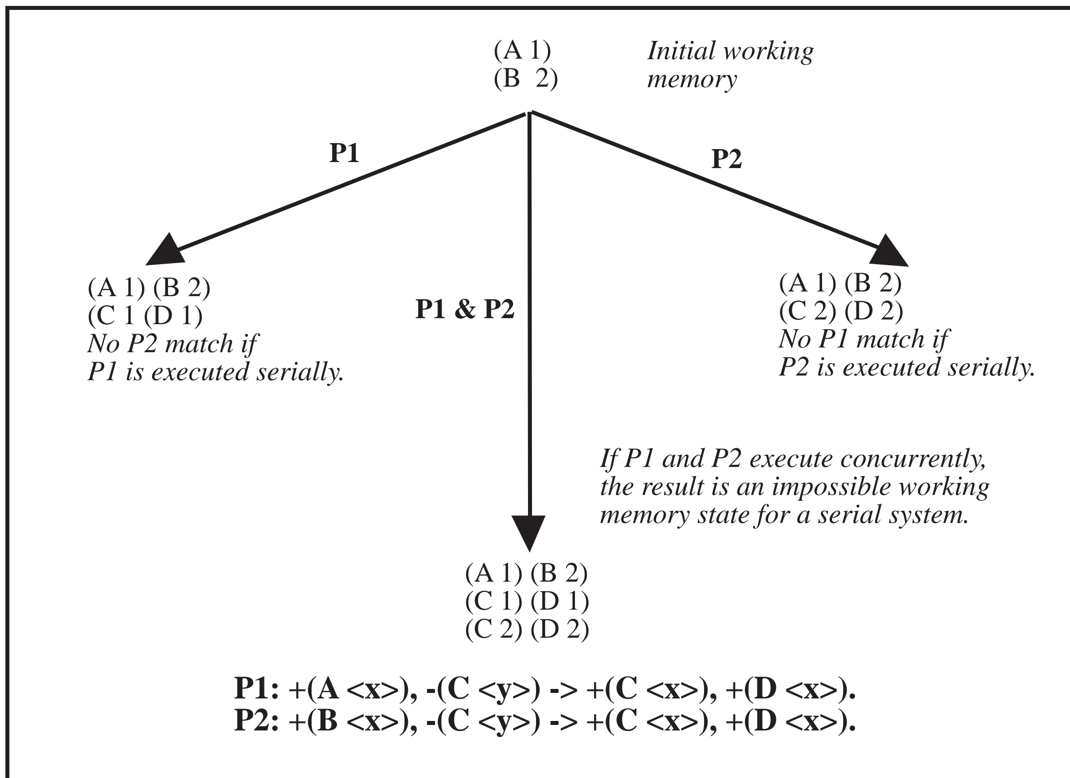


Figure 3.2: When mutually disabling rules are allowed to fire concurrently, the result may be a working memory state which could not be produced by any sequential rule firing.

A number of techniques have been developed for detecting rule interactions [Ishida and Stolfo, 1985, Tenorio and Moldovan, 1985, Miranker *et al.*, 1989, Ishida, 1990, Schmolze, 1991]. These algorithms usually consist of a static analysis phase which is performed at compile time and a runtime component which dynamically examines all eligible rules and selects a co-executable set. These algorithms can differ in the precision with which they identify potential rule interactions. For example, Ishida and Stolfo's original algorithm developed a complete table of rule compatibilities at compile-time and the dynamic phase consisted simply of a table lookup. A purely static analysis may prove unnecessarily restrictive in preventing rules from firing concurrently because it does not have access to the variable bindings which are not instantiated until rule execution time. The static analysis is therefore limited to identifying potential interactions and generating tests to be performed at runtime to determine whether the interactions actually occur. Thus, the Ishida and Stolfo algorithm unnecessarily prohibited the parallel execution of many independent rules, including the parallel execution of any instantiations of the same rule.

In comparison, Schmolze's algorithms for detecting rule interactions are very precise; the compile-time analysis not only identifies rules which can not execute concurrently, but identifies potential interactions and develops tests to be performed

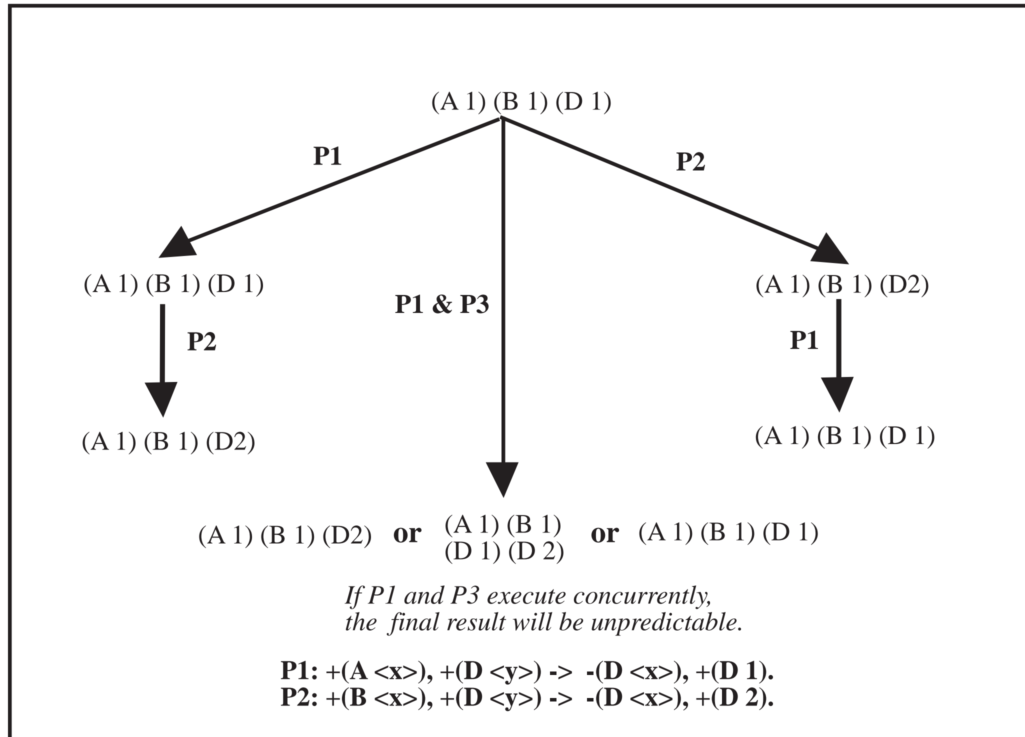


Figure 3.3: Execution of clashing rules in OPS5 can result in the assertion of redundant working memory elements.

at run time to determine whether instantiations truly interact. These runtime tests are able to precisely identify rule interactions due to clashing or disabling relationships between rules, but incur a potentially high (time) cost, both because of the synchronization cost of ensuring that all eligible rules have been identified, and because the actual tests required to precisely identify the rule interactions may be expensive relative to the cost of rule execution. Thus, there is a trade-off between the precision of interaction detection algorithms and the overhead of the tests; high precision leads to increased concurrency, but at a cost. These runtime rule analysis algorithms are typically (but not always, see [Schmolze and Neiman, 1992]) performed synchronously over the entire set of eligible rule instantiations, the output being a set of rule instantiations partitioned so that rules within each partition are guaranteed not to be mutually clashing or disabling. Although the production of maximally parallel sets would be prohibitively expensive, an attempt is usually made to insure that the set of co-executable productions is as large as possible. Because the runtime detection of rule interactions requires a best-case $O(N^2)$ pairwise comparison of rule instantiations (where N is the number of eligible rules), the cost of detecting pathological rule interactions rises sharply as the number of eligible rules increases.

Table 3.1: Frequency of rule interactions compared to total number of rule executions.

Benchmark	Rules Executed	Rule Interactions
Travelling Salesperson	510	1
Toru-Waltz	370	20
Circuit	NA	0
Parallel Alexsys	1930	1

3.2 A Locking Scheme for Ensuring Partial Correctness of Working Memory

In the programs which have been developed for UMPOPS (see Chapter 6), rule interactions have been observed to occur only rarely. Typical statistics for interaction frequency is given in Table 3.1. Rather than accept the synchronization delays associated with a full analysis of rule interactions, it was chosen to enforce only a subset of the correctness criteria using a scheme of read/write locks on working memory elements. By use of a locking scheme similar to that used in database management systems, it is possible to prohibit rule instantiations from modifying working memory elements which are being referenced by other instantiations and to prohibit rule instantiations from referencing working memory elements which are currently being modified. If such interactions are prohibited, cyclical disabling relationships cannot exist, and serializable executions will result. Because lock acquisition is inexpensive, the overhead of maintaining correctness during rule-firing is minimized. However, a locking scheme cannot prevent rule interactions due to interactions due to negated condition element clauses in rules; this represents a compromise between performance and the need for careful program design; this trade-off is discussed in following sections.

The implementation of the locking mechanism is as follows. Each working memory element is assigned a structure which contains a read counter and a write flag and a lock to ensure that modifications of these structures occurs in a critical region. As each rule instantiation enters the conflict set, each working memory element that appears on the lefthand side and that is modified in the righthand side is placed on that instantiation's *write* list. Each working memory element that is referenced in the LHS but not modified is placed on the instantiation's *read* list.

Before the rule instantiation is executed, each list is examined. If any of the working memory elements on the read list have their write flag set, then another rule currently executing is about to modify that working memory element. Because the rule instantiation will eventually be disabled by the removal of the element being modified, it is removed from the eligibility set but not executed. Similarly,

if any of the working memory elements on the write list have their write flag set, the instantiation is also removed from the conflict set, otherwise, if other rules are referencing the working memory element, the rule instantiation is not executed, but is placed back on the eligibility set. After rule execution all read locks in working elements are decremented. Write locks are never reset because their associated working memory elements will have ceased to exist after execution of the rule instance. All read and write privileges associated with working memory elements must be obtained before rule execution, thus the righthand side of a production may be considered atomic; either all of the actions will be executed, or none will be. A single scheduling process is used to assign all locks so deadlock will never occur during the assignment of locks to a rule instantiation.

Although the creation of a working memory element is certainly a write operation, it is treated as a special case. When a rule creates a working memory element, it actually acquires a read lock (increments the read counter) for that element. The reason for the special treatment is that when asynchronous rule firing is enabled, a rule might be stimulated by the addition of a working memory element and become eligible to fire even while the working memory element is still being added. If a write lock were obtained on the working memory element, any rules stimulated by that element would be unable to fire (and, if fact, would be discarded from the eligibility queue). If a read lock were not obtained on the working memory element, then the rule stimulated by the element could theoretically delete it, causing competing match operations and a possible race condition within the Rete net.

3.2.1 Region Locks and the Make-Unique Construct

The concept of locking elements to prevent interactions due to concurrent modifications is widely used in database systems and a similar scheme to the one just described has been implemented in a DBMS-based production system [Sellis *et al.*, 1987]. Their implementation uses *region locks* to prevent interactions due to negative conditions. A region lock typically prohibits access to a class of working memory elements, possibly restricted by value and, depending on the precision with which the region can be identified, may prove unduly pessimistic in restricting access to working memory [Fennell and Lesser, 1977, Corkill, 1989].

UMPOPS provides a mechanism similar in nature to region locking which allows a single working memory element to be locked, even before that element has been created. This mechanism, called `make-unique`, allows the programmer to define a working memory element and certain key fields to be unique, that is, only one attempt to create an element with the class and key values will succeed and all other attempts will be prohibited. The `make-unique` operator requires less overhead than general region-locking because only the rule instantiations attempting to create the new element check for uniqueness. Once the element has been created, the standard

lock mechanism will prevent any possible clashing behaviors. The **make-unique** operator requires that the programmer predict in advance that multiple instantiations may attempt to create the same element; the trade-off is that checking for interactions can be performed very precisely without imposing an overhead on unrelated rule firings. As will be seen in Section 6.2.3, the **make-unique** mechanism is crucial for implementing common programming idioms such as merging the results of a parallel search.

3.2.2 Principal Advantages of a Working Memory Locking Scheme

The working memory locking scheme presents a number of advantages and disadvantages as opposed to the serializability guarantees provided by Schmolze's or Ishida's algorithms. The advantages are listed below.

Low Overhead: The overhead of the UMPOPS locking scheme is limited to the generation of the read and write lists and the actual acquisition of the locks, both of which incur minimal costs. Because locks are acquired independently of other executing instantiations, the lock acquisition time is $O(N)$ where N is the number of elements referenced by the instantiation. This implies that the cost of lock acquisition for a rule instantiation does not increase as the size of the eligibility set or degree of parallelism increases. In contrast, the overhead associated with the scheme proposed by Schmolze is at best $O(N^2)$ where N is the number of instantiations in the eligibility set [Schmolze, 1991].[2] As will be seen in the following section, guaranteeing full serializability using an expanded locking scheme may impose a serial delay of as much as 10% of rule execution time, thus limiting the maximum obtainable parallelism to a factor of 10. Measurements of the working memory locking scheme described above indicate that the overhead is no more than 1-2% of rule execution time, allowing even the single scheduler implementation of the lock manager in UMPOPS to provide a potential speedup of 50 to 100-fold. The overhead for acquiring locks for various rules for a typical program is shown in Figure 3.4.

Rules can execute asynchronously: When acquiring working memory locks, it is not necessary to compare each eligible rule against all others, so synchronization is not required and rule-firing may take place asynchronously. Interaction detection does not necessarily imply synchronization; Schmolze has recently developed an asynchronous version of his interaction detection algorithm in which eligible rules are checked against only currently executing rules. This approach to runtime interaction detection proved considerably faster than the synchronous version but shares with all other asynchronous firing policies the disadvantage of prohibiting the development of (nearly) maximally parallel rule sets.

[2] To limit this overhead, Schmolze limits the number of rules scheduled for execution to be a small multiple of the number of available processors when executing rules synchronously.

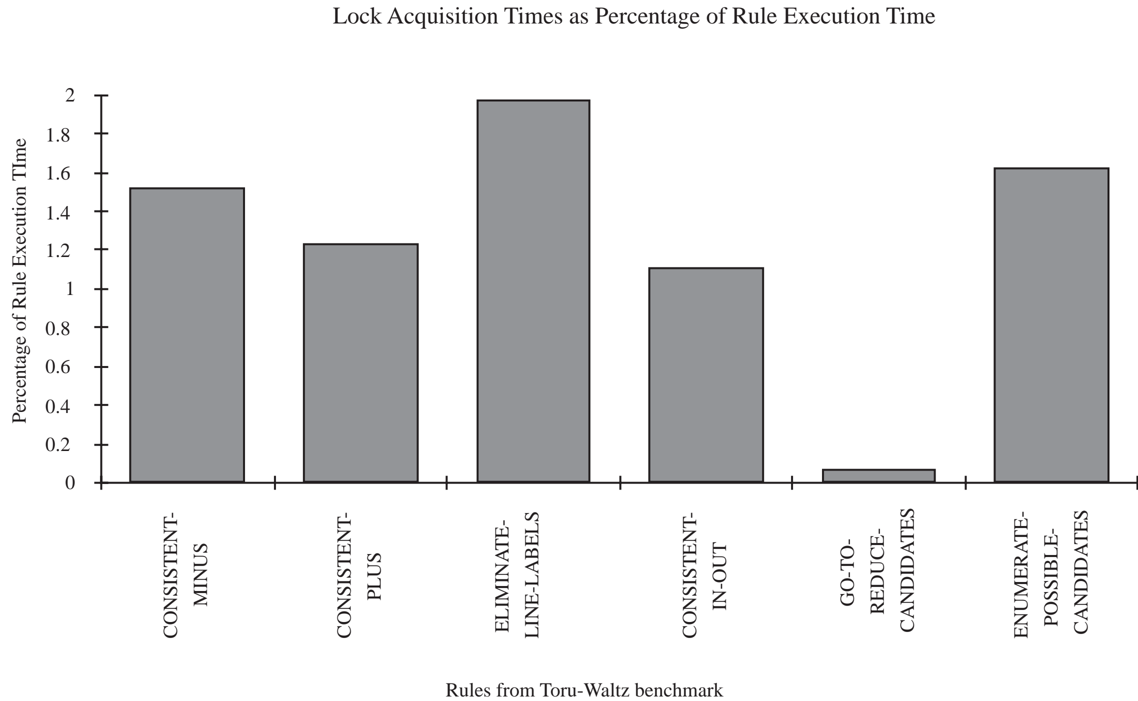


Figure 3.4: The overhead for acquiring locks in the Toru-Waltz benchmark measured in terms of percentage of total rule execution time.

No compile-time analysis is required: Because all working memory elements read or written by a rule instantiation are automatically determined at run-time, no compile-time analysis is required. This is only truly significant in systems in which the RHS syntax is complex or continually changing or in developmental systems in which the rule-base is in constant flux, requiring frequent compilation.

3.2.3 Limitations of Working Memory Locks

The primary disadvantage of the working memory locking scheme is that it does not *guarantee* serializable programs; instead it allows serializable programs to be designed and constructed. The burden falls upon the programmer to ensure the correctness of the program. Interactions involving rules that contain negated condition elements cannot be detected or prevented because it is not generally possible to acquire a lock on an element or set of elements which does not exist (however, it is precisely this capability that is provided for the single element case by the `make-unique` construct). A novel algorithm for guaranteeing complete serializability and an analysis of the accompanying cost is given in the following section.

When executing rules asynchronously, the locking scheme accepts and schedules rules in the order in which they arrive; thus opportunities for heuristic control are

not available and heuristically preferable rules may be locked out by less suitable rules which happen to be instantiated first. As reported elsewhere [Neiman, 1991], this is compensated for by the increased throughput provided by the asynchronous rule execution policy. A method for improving this situation is suggested in the following chapter. A final disadvantage of the locking scheme is that the current algorithm requires a central scheduler and lock acquisition takes place serially, thus keeping lock overhead to a minimum is critical. Lock acquisition could be performed in parallel by using a version of compile-time analysis to ensure that potentially interacting rule instantiations are handled by the same lock manager. The experience with UMPOPS indicates, however, that lock acquisition does not represent a significant enough bottleneck to justify the increased complexity of multiple lock managers.

3.2.4 A Mechanism for Detecting Interactions Due to Negative Condition Elements

The locking scheme described in the previous section can only ensure serializable behaviors between rules that do not contain negated condition elements. Rules that contain negated condition elements in their lefthand side can be disabled by the addition of working memory elements or enabled by the removal of working memory elements. Because it is not possible to acquire a lock on a working memory element that does not exist, a simple locking scheme is not sufficient to ensure that there are no conflicts between such rules. To illustrate this problem, consider the following example which illustrates a common *initialization* idiom. If a working memory element of the class `marsupial` does not previously exist, then it is created. Because of the negated condition element, it is possible for multiple instantiations of an initialization rule such as the one shown below to be enabled concurrently.

Rule:

```
P-Init-marsupial
   (Is-A ^type marsupial ^value <m>)
  -(marsupial)
 -->
   (marsupial ^type <m>)
```

Working Memory:

```
(Is-A ^type marsupial ^value wombat)
(Is-A ^type marsupial ^value koala)
```

In a serial system, either **(marsupial ^type wombat)** or **(marsupial ^type koala)** would be asserted; however if these instantiations were executed concurrently, *both* **(marsupial ^type wombat)** and **(marsupial ^type koala)** would be asserted – this is a non-serializable result. The following section outlines and presents a short cost analysis for a scheme for asynchronously insuring consistency in programs containing rules with negated condition elements.

Previous schemes for detecting interactions between rule *instantiations* due to negated elements have relied on run-time interaction detection using specific tests developed during a compile-time analysis [Schmolze, 1991, Kuo *et al.*, 1991]. These tests are applied to potentially executable instantiations on a pair-wise basis with all other eligible instantiations. (In Schmolze's asynchronous algorithm [Schmolze and Neiman, 1992], the eligible instantiation is compared against all *currently executing* rules in order to confirm that none of their working memory modifications will disable it.) When examined, the comparison tests which are applied to eligible rules look very similar to those tests performed within the pattern matcher, and for a good reason. The pattern-matcher must also detect disabled rules and eliminate them from the conflict set. The observation of the equivalence of tests performed by the interaction detection algorithms and the pattern matcher inspired the following scheme. Instead of producing the tests during a precompilation phase, they are instead produced automatically by the traversal of tokens through the pattern matcher. This use of the tests from the pattern matcher reflects the intuition that the work being done by run-time interaction detection is precisely that work that would be done by the pattern matcher in order to retract disabled instantiations from the conflict set in a serial implementation.[3] This observation justifies the following experiment. Because the same tests are used to perform matching and interaction detection, it is possible to assert that any optimization that increases the speed of run-time interference checking could also be applied to the pattern matching phases, thus the relative overheads of the two processes should remain the same despite any modifications to either the algorithms or the implementation language.[4]

In the Rete net [Forgy, 1981], when a working memory element is positively matched, a token representing that element is concatenated to a set of tokens being propagated through the network. We can similarly create a *pseudo-token* corresponding to a successful match of a negated element. This token represents a

[3]The similarity between locking and pattern-matching in rule-based systems was exploited in the opposite direction by Stonebraker, Sellis, and Hanson in a system that used database locking techniques to detect rules that should be triggered following database updates [Stonebraker *et al.*, 1986].

[4]There are cases in which one rule will always disable another; in such cases, interaction detection algorithms need perform no instantiation-specific tests, and the overhead will be very low. Thus the equivalence between tests holds only for cases where disabling relations between rules cannot be determined at compile-time.

pattern of the working memory elements that would disable this instantiation. This pattern is simply the set of tests encountered by the working memory element as it proceeds through the matching process; specifically, the inter-element *alpha* tests preceding the **NOT** node, concatenated to the tests performed by the **NOT** node and unified with the positively matched tokens in the rule instantiation.

In the initialization idiom shown above, the pseudo-token generated for `P-Init-marsupial` would be simply `((class = marsupial)` – any rule instantiation created an element of the class `marsupial` would conflict. Consider, however, the more complex rule instantiation shown below, stimulated by the addition of the working memory element `(A wombat)`. The resulting pseudo-token would have the form `((class = B) (field(1)=wombat) (field(2)=koala))`. Any currently executing rule which creates an element matching this pattern would disable this instantiation of `P-interaction-example`. By applying the above test to all executing rules, it can be determined whether the instantiation below would be disabled or whether it can be safely executed.

Rule:

```
P-interaction-example
    (A <x>)
   -(B <x> koala)
   -->
    (B <x> koala)
```

WM:

```
    (A wombat)
```

When a rule instantiation is created, we now have two sets of tokens: positive tokens that describe working memory elements enabling a token, and negative pattern tokens that describe working memory elements whose creation would disable the instantiation. We can acquire read/write locks for the positive tokens using the technique described in the previous section. The negative tokens are used in the following way.

We first require that, for each instantiation, a list of the working memory elements that it will assert be created before the interaction detection algorithm is run (and therefore, before rule execution). To provide maximum parallelism, this list is created when the instantiation is created by a rule demon. This modification to the rule execution algorithm does not represent an increase in overhead because the formation of working memory elements must eventually take place, and it does not matter whether it occurs before the rule is inserted into the conflict set or immediately before the rule is executed, assuming that most rules entering the

conflict set eventually fire.[5] Before each rule instantiation is executed, it posts all the elements it is about to positively assert on to an **ADD** list. This list may be hashed according to the class of the element being referenced, but should not be highly structured in order to minimize add and delete times. The steps of the interaction detection algorithm then proceed as follows.

1. As each candidate instantiation arrives, ensure that locks can be acquired for each positively referenced working memory element as described in the previous section (however the locks are not actually acquired until after the following step is completed).

2. Once it is certain that all the necessary working memory locks can be acquired, compare the instantiation's negated pattern elements against the list of all working memory elements on the **ADD** list. If any of these elements match against a negated pattern element, then the rule instance will be disabled by some currently executing rule and should not fire.

3. If the rule instantiation is not disabled by a currently executing rule, acquire the locks on the positively referenced elements, post the working memory elements that it is about to create onto the **ADD** list and schedule it for execution.

4. After the rule has completed execution, remove from the **ADD** list the elements it has just placed in working memory.

This algorithm is depicted pictorially in Figure 3.5.

Overhead analysis: The scheme described above for ensuring that currently executing rules do not disable a rule with negative condition elements ensures that the execution of the production system will be fully serializable. However, because the lock mechanism must run serially in order to avoid deadlocks between rules simultaneously attempting to acquire locks, we have to determine whether or not checking the negated tokens against the **ADD** list is inexpensive compared to the time necessary to execute a rule. If not, the tests on each instantiation will form a serial bottleneck and the performance improvements due to parallel rule execution will be lost. In order to form an estimate of the overhead associated with the above algorithm, we note that the processing performed in matching each working memory element being asserted against each negated pseudo-token pattern is essentially equivalent to the time of a beta node activation within the Rete net (for the check against the **ADD** list) and two memory node activations (one for each addition or deletion to the **ADD** list). This approximation is reasonable because the tests contained within the negated pseudo-tokens are derived from the **NOT** nodes that

[5] However, if one rule adds an element which enables many succeeding rules, then that rule's run time will be increased. Thus, mode-changing rules have been observed to be more expensive in this version of UMPOPS

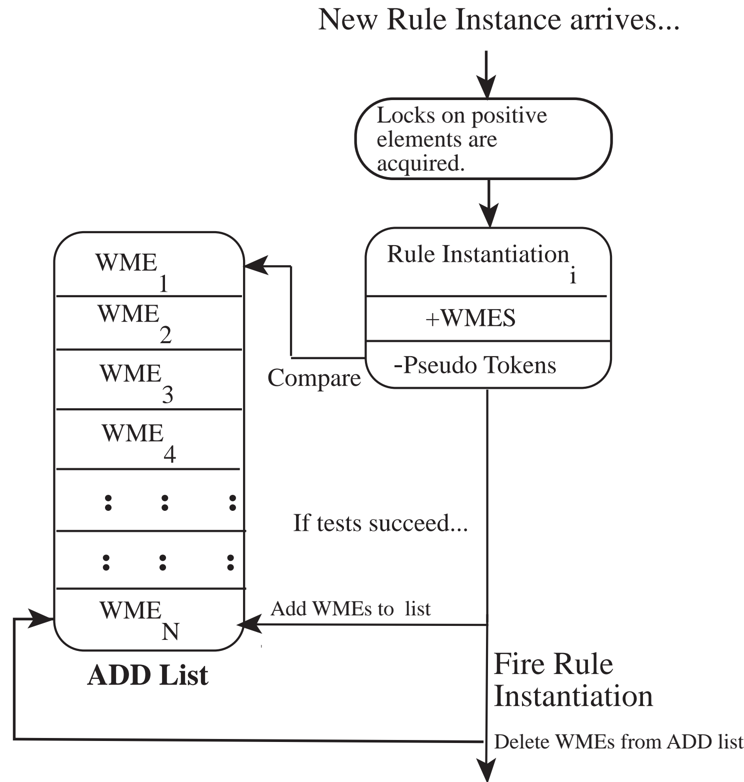


Figure 3.5: An algorithm for asynchronously detecting rule interactions involving negated condition elements.

generated them. The beta nodes are the most time-consuming component of the pattern matching process and the number of beta nodes executed can be used to create an estimate of relative costs. Using the statistics gathered by Gupta [Gupta, 1987], we note that the average rule instantiation activates approximately 40 beta node and memory operations, with the actual figures, of course, depending on the size and complexity of the lefthand side conditions. Assuming some unavoidable implementation overhead in the above algorithm, we see that the runtime detection of interactions due to negated tokens may incur costs of as much as 8 to 10% of the cost of actually executing the rule for *each negated condition in the rule.* Because the detection of interference must be carried out within a critical region of the scheduler, an overhead of one negated rule per rule, on average, would limit the potential parallelism within the system to a factor of 10 to 12, exclusive of other scheduling costs. Because the size of the **ADD** list increases proportionally to the number of rules executing or scheduled to execute, the overhead of the negated token test increases as the degree of parallelism increases, further diminishing the concurrency of the system.

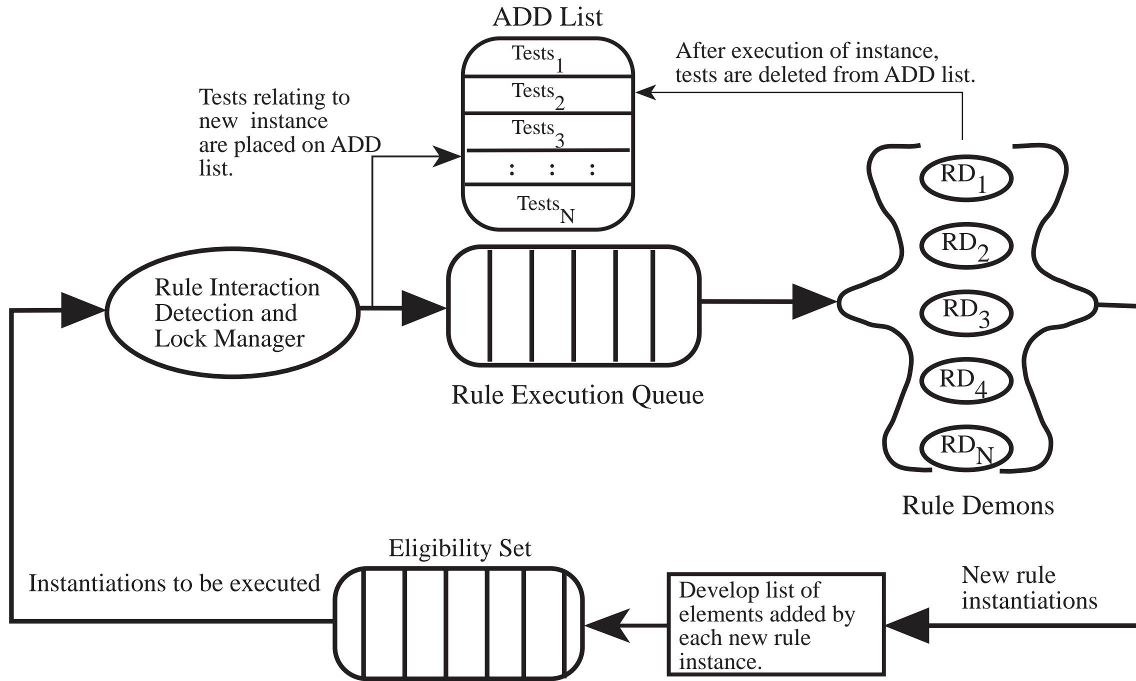


Figure 3.6: The essential architecture for a parallel rule-firing system with a serial mechanism for lock acquisition and rule-interaction detection.

3.2.4.1 *Experimental Verification of Overhead Analysis*

In order to verify this overhead analysis, an experimental version of UMPOPS was developed that incorporated the locking mechanism described in this section. The architecture of this system is illustrated in Figure 3.6. The program used to test the performance of the locking mechanism was a version of Toru-Waltz, an implementation of the Waltz line-labelling algorithm implemented in OPS5 by Toru Ishida (see Chapter 6 for a full description of this program). Toru-Waltz is a good benchmark for testing mechanisms for acquiring locks and identifying rule interactions because it contains rules that are fairly small and execute rapidly. Because the parallelism in a rule-firing architecture of the type depicted in Figure 3.6 is directly related to the ratio of lock acquisition time to rule execution time,[6] the performance of rules with rapid execution times will emphasize serial bottlenecks.

Two versions of Toru-Waltz were tested, one version that required locking only positive condition elements, and one version that required checking for disabling conditions on negative condition elements. Using 21 processors on a Sequent Symmetry, the first version executes in 1.6 seconds; the second in 2.1 seconds (see Figure 3.7). Using only working-memory locks, the overhead for lock generation

---

[6] For the sake of brevity, I will refer to both lock acquisition and detection of interactions due to negatively referenced condition elements as lock acquisition.

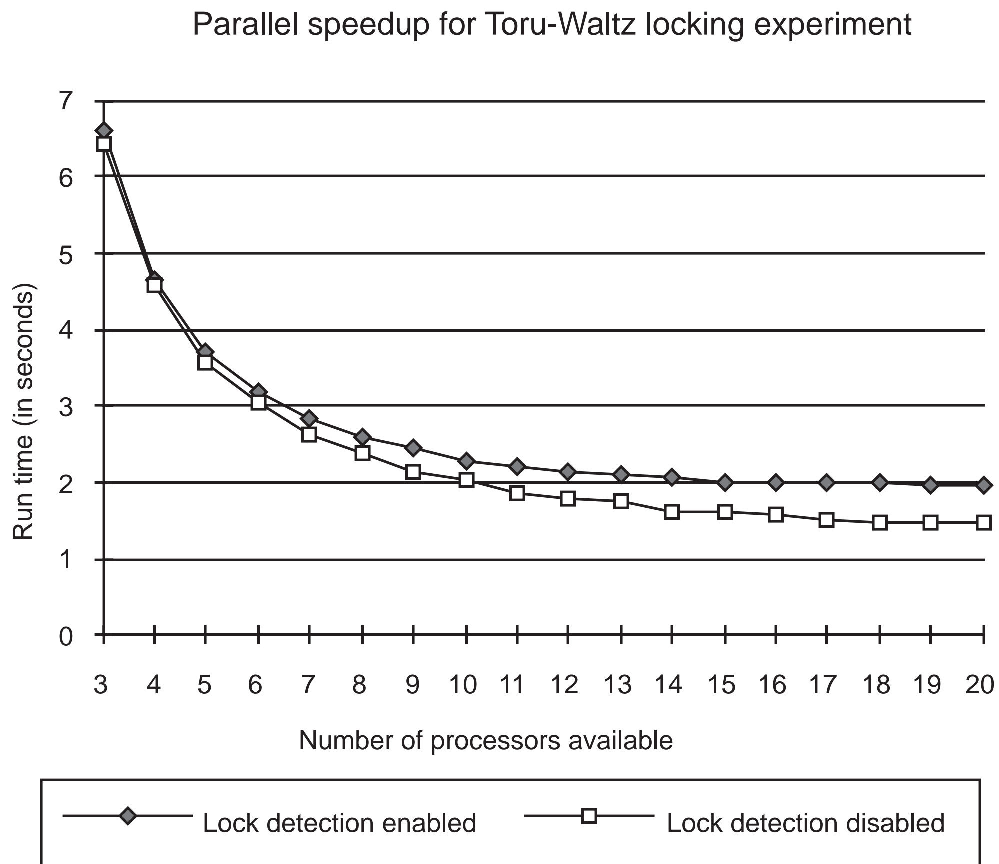


Figure 3.7: The parallel speedups of two runs of the Toru-Waltz benchmark are shown. It can be seen that the performance of the version incorporating the lock-detection mechanism is approximately 25% slower than the version without.

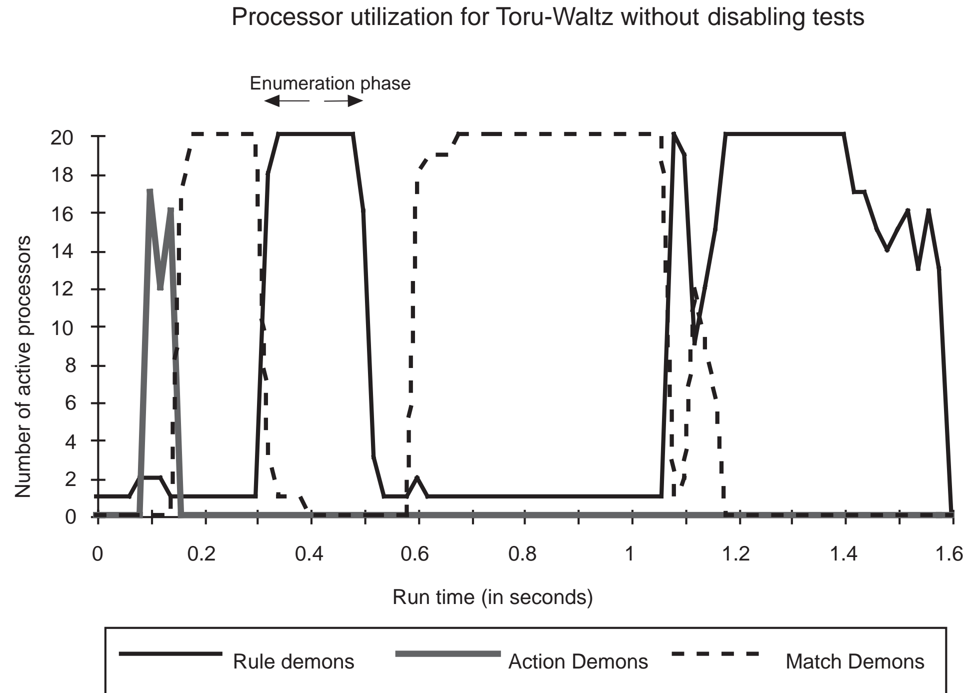


Figure 3.8: The processor utilization graph for Toru-Waltz without rule-interaction detection. During the enumeration phase, all 20 processors are employed in executing rules.

for each rule ranges from 1 to 3%. When the mechanism which detected disabling conditions due to negated condition elements is used, lock overheads are observed which range from 5 to 15% of rule execution times. The effects of these overheads can be seen by comparing the processor utilizations obtained by rule parallelism in Figure 3.8 and Figure 3.9. The Toru-Waltz program consists of two main phases, an *enumeration* phase in which working memory elements representing possible line labels are created, followed by a second phase in which inconsistent labels are deleted. During the enumeration phase, many new working memory elements are created and corresponding tests are placed in the ADD list, requiring tests to be performed during lock acquisition. During this phase, the ratio of lock acquisition to rule execution times is typically 1 to 10, and as expected, the average parallelism obtained during this phase averages no more than a factor of 10. During the second phase of Toru-Waltz, there are no additions made to working memory. Because the ADD list is empty and no tests are performed, the lock acquisition overhead is reduced to approximately 5%, and there is little appreciable reduction in available parallelism as measured by processor utilization.

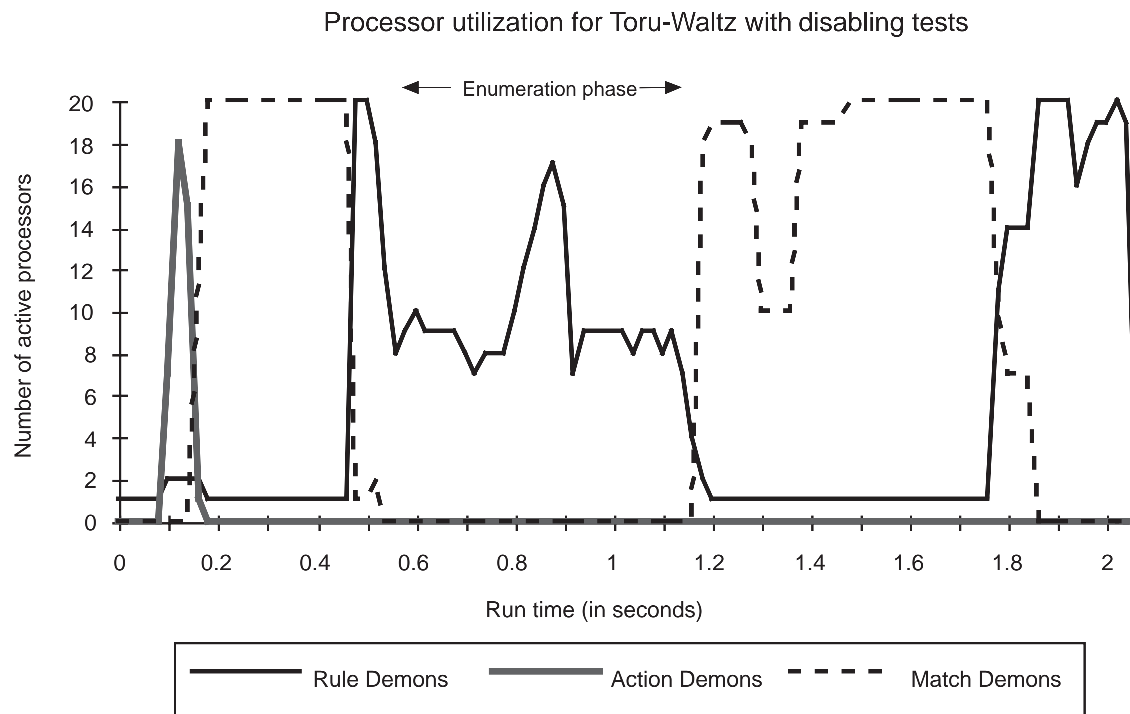


Figure 3.9: The processor utilization for Toru-Waltz with rule-interaction detection enabled. During the enumeration phase, only an average of 10 rules execute at a given time and the remaining processors are idle.

3.2.5 Conclusions: Guaranteeing Serializability

One can conclude from this experiment that one pays a fairly high price for ensuring serializability. While the mechanisms for ensuring consistency can certainly be optimized, because of the symmetry of testing described previously, any truly dramatic decrease in testing time would also be reflected in the matching phase. It appears, therefore, that an overhead of approximately 10% is unavoidable for ensuring serializability in a rule-firing architecture such as the one described in Figure 3.6. This will limit parallelism to a single order of magnitude if rule-interaction detection is performed.[7]

One could potentially improve throughput by doing a compilation-time analysis similar to Schmolze's on the rule set and using the resulting syntactic knowledge to partition the rules according to potential interactions such that multiple lock acquisition processes can be used. Another possibility for decreasing the time required to acquire locks is to apply micro-level parallelism to the checking process such that new candidate rules are compared against executing rules in parallel.

[7]On the bright side, applications that require fewer than 10 processors can employ run-time interference detection without appreciable overhead.

Because the overhead figures which I have obtained are proportional to the ratio of lock acquisition times to rule execution times, interaction detection mechanisms may prove to be more suitable for environments such as blackboard systems in which the units of execution are of a higher granularity. Although this discussion has focused on locking and detection of rule interactions, the analysis is appropriate for any overhead such as control scheduling or heuristic pruning which occurs during sequential processing.

3.3 Designing for Correct Parallel Execution

Since ensuring serializability using run-time interaction detection is expensive, the best approach to obtaining high levels of parallelism appears to be designing rule-based systems in which rule interactions do not arise or in which they can be handled safely by relatively inexpensive mechanisms such as working memory locks. Because of the semantic nature of rules in a production system, each righthand side action can be assumed to have a definite purpose. Thus, situations in which rules can potentially interact can be detected at design time and appropriate remedial actions taken. This approach places the burden of ensuring correctness squarely on the programmer.

The key to this approach is the question "Why are these rules interacting?" We can create a (partial) taxonomy of roles that working memory elements play in computations and devise mechanisms for each that will avoid rule interactions. (The taxonomy is partial because I am unaware of any method for completely enumerating the potential uses of a data item.)

3.3.1 Spurious Rule Interactions

My discussion of the semantic avoidance of rule interactions implicitly assumes that the rules in the eligibility set are in some sense non-identical. It is possible for semantically identical rule instantiations to occur within the eligibility set if the working memory elements that satisfy a set of condition elements can be combined in multiple ways and no ordering relationship is specified between these elements. For example, such spurious rule instantiations will occur when a "counting" idiom is used:

```
(p at-least-3-cats
     (cat ^name <x>)  ;find a cat and bind its name to <x>
     (cat ^name {<> <x> <y>}) ;another cat, <y>, not named <x>
     (cat ^name {<> <x> <> <y> <z>}) ;yet another cat, <z>,
                                      ;not named <x> or <y>
     -->
    ...)
```

Because each `cat` element can match any condition element, a combinatorial number of instantiations would appear in the conflict set. This situation can be avoided by imposing some kind of order on the condition elements, for example, by sorting the elements according to lexical order by name; or by adding a second-order predicate to the rule language (such as `(number-of-elements ^type cat ^number 3)`); or by adding features to the language which make such awkward idioms unnecessary. For the purpose of this discussion, I will assume that all spurious combinatorial instantiations of this nature are due to the lack of expressiveness in the OPS5 programming language and have been eliminated by clever programming or by the addition of new language constructs. For a discussion of modifications to the Rete pattern matcher which facilitate counting and sorting operations, see [Schor *et al.*, 1986].

3.3.2 Semantics of Rule Firing

In this thesis, I use the phrase "semantics of rule firing" frequently; this usage is based on the theory that each rule firing and accompanying working memory change is purposeful. The philosophy behind rule-based systems is that each rule captures some unique quantum of knowledge about the domain and each rule firing applies that unique knowledge in a manner appropriate to the current state of the world as mirrored in working memory. Although the knowledge captured in a rule is ideally domain knowledge, in practice, rules are used to represent control knowledge as well. Therefore, throughout the computation, working memory element are used for many purposes: to represent real world "facts"; to represent states of the computation; and to control the sequencing of rule firings. It is possible to establish a (by necessity, partial) list of the uses of working memory elements and, by implication, a list of the semantic function(s) of the rules creating those elements.

- Real World "Facts": The use of working memory elements to represent facts about a domain is common in AI programming; similar representations are universally used in logic programs, GPS-like problem solvers, and so on.

- States of the Computation: Most AI problems can be cast in terms of search. Any state in the search space can be represented by a unique set of working memory elements. State elements are distinguished from domain facts in that contradictory facts or elements are allowed to exist, as long as they exist in disjoint states.

- Meta-knowledge about the computation itself: Working memory elements are frequently used in a "meta" fashion to control the sequence of rule firing; they can also be used to represent transient control information. Typical uses of meta-level representations include descriptions of goals, subgoals, search spaces explored, and problems yet to be solved.

- Expendable Resources: Related both to search and to real world facts, working memory elements can be used to represent resources that must be apportioned in a unique fashion, that is, only one consumer is allowed access to each resource.

In like manner, we can assign a semantics to rule executions, characterizing rules as initialization rules, control rules, meta-control rules, or domain specific operators in a search space. If we understand how each working memory element is being used, then we can be prepared to understand the import of rule firings and possible rule interactions and design parallel rule-based programs accordingly. In many cases, understanding the causality of rule interactions allows expensive but general interaction detection algorithms to be replaced with inexpensive language constructs tailored for specific types of rule interactions.

3.3.2.1 Contention for Resources

A working memory element may represent a *resource*, that is, some kind of consumable quantity which can be used by only one rule. For example, in an airline reservation system, a rule might appear as follows:

```
;If there is a passenger who desires a seat
;and there is a seat which  satisfies the constraints
    on smoking and windows,
;and there is no passenger currently assigned that seat
;then assign that seat.
  (P reserve-seat
    {<passenger> (passenger ^seat nil ^window <w-flag>
                           ^smoking <s-flag> ^flight <flight>)
    (seat ^number <seat> ^window <w-flag>
                        ^smoking <s-flag> ^flight <flight>)
   -(passenger ^seat <seat> ^flight <flight>)
    -->
    (modify <passenger> ^seat <seat>))
```

Two possible errors can arise if multiple instantiations of this rule execute concurrently. The passenger could be assigned multiple seats (clashing), or the seat could be assigned to multiple passengers (disabling). The first error can be prevented trivially through the use of working memory locks. But the second error occurs because of the negatively referenced element, an interaction which is much more expensive to guard against. We can use our understanding of the semantics of *resources* to transform this rule into one which can use only positive locks by marking each resource as it is assigned.

```
;If there is a passenger who desires a seat
;and there is a seat which satisfies
;    the constraints on smoking and windows,
;and that seat has not yet been assigned,
;then assign that seat.
 (P reserve-seat
   {<p> (passenger ^seat nil ^window <w-flag>
                  ^smoking <s-flag> ^flight <flight>)}
   {<s> (seat ^number <seat> ^window <w-flag>
             ^smoking <s-flag> ^flight <flight> ^assigned nil}
 -->
      (modify <p> ^seat <seat>)
      (modify <s> ^assigned t))
```

In this version, both sources of error are avoided by working memory locks; because both elements are modified, write locks are obtained for both resources and no other rule instantiations are allowed to reference them while the **reserve-seat** rule executes. Now, what transformation was employed? We converted a universal quantifier, "for all passengers, none is assigned this seat," to an existential quantifier, "there exists a seat that is not assigned." Universal quantifiers require checking *all* relevant working memory elements while an existential quantifier only requires finding a single element. Because the universe of possible resources is usually finite and enumerable, the transformation to existential quantifiers can take place, reducing the set of possible rule interactions to those that can be inexpensively prevented by working memory locks.

3.3.2.2 Control Elements

A common (perhaps the most common) source of rule interactions observed in systems which attempt to automatically extract parallelism from existing OPS5 programs is the coexistence of domain and control rules in the conflict set. This is an artifact of the use of conflict resolution to impose a control flow upon a program. Stages of the computation are triggered by the addition of *mode* elements to working memory. These mode elements trigger the relevant domain rules. In order to change mode, a control rule is also defined; it matches only the modal working memory element. Because the control rule contains only one condition element in its lefthand side and domain rules typically have multiple condition elements, the *specificity* clause of the conflict resolution mechanism ensures that the control rule will only execute after all domain rules have been executed. Although not a transparent method of implementing control flow, this method is commonly employed in production systems such as OPS5 that do not provide any explicit methods of specifying program sequencing. In a parallel rule-firing system, one runs the risk of firing the mode-changing rule in parallel with the eligible domain rules, potentially terminating the phase of the computation prematurely.

The solution to this type of interaction, developed independently by a number of researchers [Ishida, 1991, Kuo and Moldovan, 1991, Neiman, 1992a, Schmolze, 1991], is to explicitly partition the rule set into domain and control rules. Control rules are annotated as such (e.g., the meta-notation, `mode-changer` in UMPOPS). The scheduler (or rule-firing engine) is responsible for ensuring that control rules fire if and only if all domain rules have been fired and all working memory changes have been completed. In this way, program sequencing is achieved without the risk of rule interaction and, incidentally, without the need for manipulating the conflict resolution protocol. Because imposing control flow on a computation is one of the primary uses of conflict resolution in the OPS family of languages, creating a dichotomy of control and domain rules greatly reduces the need for conflict resolution routines and their attendant overhead.

3.3.2.3 Search States

Working memory represents a state in a computation and rules alter that state. If rules in a conflict set are truly in conflict, then they represent alternative valid modifications to that state. As such, these rules will likely interact as they modify the current state of working memory. In such a case, we can resolve the interactions by casting the computation in terms of search. In a search process, it is frequently necessary to represent distinct alternative states of a computation. By partitioning the states in the search space (through a number of mechanisms which will be described shortly), it is possible to execute conflicting rules without interactions. The result of this is two (or more) discrete sets of working memory elements, each representing a valid solution (or path to a solution). If partitioning the state is not feasible (for reasons of space or time), then performing conflict resolution between competing rules will be necessary to eliminate the interaction.

Representing rules as operators in a state space search eliminates one problem – dynamically identifying when rules interact – and replaces it with another – identifying those rules in the conflict set that actually conflict. That is to say, once a search is initiated, not all rules eligible to fire will have been stimulated by changes to the same state in the search space; those rules operating in separate states will, by definition, not interact or conflict and therefore should be allowed to fire asynchronously in parallel. UMPOPS provides two mechanisms, the multiple worlds construct, and the task construct, for partitioning search and operators in the search space. These mechanisms are discussed in Sections 5.4 and 4.3, respectively.

3.3.2.4 Merging Solutions

Partitioning a search process merely delays the inevitable: eventually a solution will have to be selected; at this point divergent paths in the computation must join and interaction is unavoidable. This interaction is manageable, however, because it occurs at a well-defined point in the computation and specific language constructs

can be used to ensure that the merging or selection of solutions takes place in a correct manner. The solution-merging idiom is described in more detail in Section 6.2.3.

3.3.3 Representation of Unique Objects (Data Parallelism)

When a direct one-to-one correspondence can be made between "real-world" objects and working memory elements, then operations can usually be applied independently to each working memory object. If objects are interconnected (e.g., semantic nets, graphs, or circuit representations), then only interactions propagated through the interconnections represent potential sources of conflict. Because these interactions are localized and purposeful, the propagation mechanisms can usually be modified so that positive locking mechanisms are sufficient to maintain correctness. Data parallelism has proven to be one of the easiest and most profitable forms of concurrency to exploit.

3.3.4 Inference

Production systems are sometimes referred to as forward-chaining inference systems. Starting with an initial set of facts, represented by the initial state of working memory, productions representing logical inferences can fire, adding new facts to the database. These facts in turn stimulate additional rule firings.

Monotonic inference is trivial to implement in OPS5. The one difficulty is that OPS5 does not represent working memory as a set; multiple identical elements may be present in working memory. This multiset implementation of working memory is inappropriate for an inference system in which facts are expected to be unique. Presence of multiple versions of facts in the database can lead to redundant triggering of productions; in a non-monotonic system in which facts are allowed to be deleted, not all duplicate facts in the database may be retracted, leading to logical errors or contradictions. To avoid the multiset problem, logical inferences in OPS5 must explicitly check for the prior existence of the fact which they are about to assert. A negative condition element ensures that the fact about to be asserted has not yet been created. As we have seen, the presence of a negative condition element in a rule's LHS can lead to conflicts when executing in parallel. For example, consider the rule set and accompanying working memory state shown in Figure 3.10.

If both rules were to execute at the same time, two versions of the working memory element `(on ^obj1 counter ^obj2 Kittyhawke)` would be asserted. Suddenly, there would be two cats raising havoc in the kitchen!

The above example, though whimsical, demonstrates that multiple trains of inference might lead to the same conclusion. In a medical diagnosis domain, for example, there may be multiple ways of concluding that a patient is suffering from a particular ailment. Because there is no run-time checking for negated elements in

Rule Set:

```
;If a cat is hungry
;   and sees food
;   and the food is on the counter
;   and the cat is not on the counter
;then
;   the cat will be on the counter.

 (p bad-cat
    (cat ^name <cat> ^state hungry)
    (see ^cat <cat> ^obj food)
    (on ^obj1 counter ^obj2 food)
   -(on ^obj1 counter ^obj2 <cat>)
  -->
    (on ^obj1 counter ^obj2 <cat>))

;If there is a forbidden object
;   and the cat sees the object
;   and the cat is not already on the object
;then
;   the cat will certainly jump on the object

(p typical-cat
    (cat ^name <cat>)
    (forbidden-obj ^obj <taboo>)
    (see ^cat <cat> ^obj <taboo>)
   -(on ^obj1 <taboo> ^obj2 <cat>)
  -->
    (on ^obj1 <taboo> ^obj2 <cat>))
```

Working Memory:

```
 (cat ^name Kittyhawke ^state hungry)
 (see ^cat Kittyhawke ^obj counter)
 (see ^cat Kittyhawke ^obj food)
 (forbidden-obj ^obj counter)
 (on ^obj1 counter ^obj2 food)
```

Figure 3.10: Rules instantiations demonstrating the possibility of two interacting chains of inference.

UMPOPS, inference rules such as those above would create redundant elements if executed in parallel.

The problems caused by redundant elements can be eliminated by implementing OPS5 memory as a set. The OPS5 working memory creation routines can easily be modified to check for the prior existence of elements before they are asserted. Tokens that duplicate existing elements can simply be discarded without altering the results of the logical inference. Because the check for prior existence would take place during the working memory assertion, it would be performed by the rule demon executing the instantiation and would not place an additional serial burden on the scheduling process. Naturally, precautions would have to be taken to ensure that two processes do not simultaneously create the same working memory element, thus violating the set semantics.

3.3.5 Domain Facts

Many working memory elements simply represent world knowledge in the form of domain facts that remain immutable during the course of the computation. If the programmer is aware that these working memory elements will not be modified during the course of the computation, then rules which access these working memory elements need need not acquire locks to ensure that they will not be modified during processing. An example of such a situation is the distance tables used in the travelling salesperson problem; the distances between cities are constant and the working memory elements representing the distances do not change. The presence of constant elements is useful when designing partitions for search; constant elements may be stored in a global partition accessible to all processes, thereby reducing the amount of duplication necessary.

3.3.6 Summary: Semantics of Rule-based Systems

This section has discussed a number of ways of interpreting the actions of rules in terms of their semantic intent. A taxonomy of rule uses was described, and methods of designing programs to ensure correct execution for each rule use were discussed. Although the listing of rule functions is far from complete, the flavor of the approach should be clear: if the causes of rule interactions are understood in the context of their role in the overall computation, these rule interactions can usually be avoided or prevented using inexpensive lock mechanisms such as positive locking or the `make-unique` construct.

3.4 Functionally Accurate Computations

The final topic relating to correctness is the question of maintaining high-level data consistency in the course of concurrent activities by independent agents or

tasks. Computations can depart from optimality in a number of ways, such as pursuing an excessive number of redundant solution paths, requiring an excessive amount of time or computation to reach a solution, or producing a solution of lesser quality or high uncertainty. If control decisions are made without full knowledge of the state of the system, as frequently happens during parallel rule firing, then we can expect the departure from optimality to increase. There is a trade-off between performance and communication – as we have seen, if tasks are closely coupled, they must devote resources to preventing interactions due to operations on shared data, while executing tasks independently may reduce the opportunities for increasing the quality of the solution by sharing results. To compensate for the lowering of solution quality in situations of uncertainty or lack of adequate communication between agents, the notion of *functionally accurate* programming has been developed [Lesser and Corkill, 1981]. The notion of reducing locking overhead through functionally accurate means was applied to parallel AI systems as long ago as 1977 [Fennell and Lesser, 1977]. Fennell and Lesser surmised that the overhead of region locking could be eliminated in parallel blackboard systems by implementing mechanisms to detect and correct incorrect program states caused by interacting concurrent activities.

The key assumption behind the Functionally Accurate/Cooperative (FAC) model is that the cost of identifying and repairing errors in a computation is less than the overhead of preventing them originally. "Functionally accurate" is more an attribute of a program rather than a specific algorithm. Some computations are functionally accurate simply by their nature. It is possible to outline some of the necessary characteristics of a program that cause it to be functionally accurate. First, it must be possible to recognize when an incorrect program state exists; that is, there must be some standard by which results can be judged. Second, it must be possible to recover from this state by retracting (or ignoring) the incorrect data, while restimulating the computation required to produce the correct answer. Third, there must be a termination criterion: it must be possible to decide when to accept a result based on the results obtained and the amount of work performed. These characteristics represent design goals in the creation of functionally accurate systems.

3.5 Summary: Correctness in Parallel Rule-Firing Systems

This chapter has discussed the approach proposed for generating correct results during the course of parallel rule-firing. A working memory locking mechanism was described that prevents rules from accessing elements currently being modified by other rules and that prohibits modification of elements currently being accessed by executing rules. This locking scheme has the advantage of low overhead and simplicity, but fails to guarantee that a computation's behavior will be fully serializable because of its inability to prevent interactions due to negated condition

elements. A method was described for extending the locking scheme to detect the addition of working memory elements which are negatively referenced by executing rules. The scheme guaranteed serializable behavior, but at a cost; the overhead was demonstrated to be on the order of 10% for a typical rule-based computation. Because an overhead of this magnitude would limit the number of processors that could concurrently execute rules to 10, it was suggested that programs should be designed to execute without full rule-interaction detection, using the positive working memory locks to resolve unavoidable conflicts. This design approach is based on the observation that rules have specific semantics and that by understanding the role that each rule plays in a computation, the reasons for clashing and disabling behaviors can be understood and avoided. Benchmark programs developed using these design techniques are discussed in Chapter 6.

CHAPTER 4

Control Issues in Parallel Rule-Firing Systems

This chapter is concerned with the control of rule-based systems as it relates to parallel execution. There are many levels at which a system can be controlled and many degrees of control sophistication. Four major areas of control will be discussed.

- Rule-Firing Policies: One of the contributions of this thesis is the development of two novel rule-firing policies that are necessitated by the elimination of synchronization overheads during parallel rule-firing. An asynchronous rule-firing policy is introduced that allows rules to be executed as soon as they become enabled. A task-based rule-firing policy has also been developed that allows rules to be grouped together for purposes of determining quiescence and conflict-resolution protocols in a local context.

- Sequential Control: Ideally, the rule-based paradigm is completely data-driven and reactive; in practice, it is frequently necessary or desirable to impose an explicit sequencing on rule-firing, mimicking the control capabilities of conventional programming languages. Because of the lack of constructs for expressing trivial imperative control idioms, the OPS5 programmer has been reduced to exploiting the specificity and recency criteria of the conflict resolution paradigm in order to impose an order on rule firing and generate loops of rule firings. This use of the recognize-select-act loop to perform low-level control activities incurs a significant overhead, while the explicit sequencing of rule instantiations eliminates opportunities for rule-level parallelism. The approach taken in this research is to add language constructs for explicitly specifying iteration and set operations and to selectively employ parallelism at the action and match levels to reduce the temporal overhead of unavoidably serial constructs.

- Heuristic Control: In most applications for which rule-based programming is appropriate, there is no known formal algorithm for producing a provably correct solution at reasonable cost and in reasonable time. Given a set of rules eligible to fire, the choice of which to execute must be heuristic. Heuristic

control as embodied by conflict resolution requires that all eligible rules be examined in order to select the instantiation(s) to execute. As discussed previously, identifying *all* eligible rules will result in unnecessary synchronization, while a control algorithm that must compare each rule with all others will not be highly parallel. The thesis that is discussed in this chapter is that, in a heuristic problem-solving situation, performing incremental discrimination on rules as they become eligible to fire will not seriously degrade the quality of the solution and will enable control to be performed concurrently with rule firing. A rule-firing policy is described that provides opportunities for imposing heuristic control at several points in the rule-firing cycle, thereby increasing the precision and responsiveness of control activities.

- Resource Utilization: In any parallel system, there is a high correlation between the number of processors that can be kept usefully employed and the degree of speedup. Rule firings should be scheduled so as to maintain a consistent demand for processing resources. When the number of rules to be fired is less than the number of available processors, processing resources should be re-directed to lower level activities that will speed the execution of the currently executing rules. The problem of scheduling rules to maximize resource usage is discussed in the context of a parallel blackboard system by Decker and colleagues [Decker *et al.*, 1991]. They propose the use of heuristic scheduling rules to ensure that knowledge sources that initiate demand for resources or that access resources in disparate sections of the blackboard are prioritized; this scheme maximizes demand for processors and minimizes delays due to locked resources. Although many of the heuristics developed for blackboard systems are also applicable to rule-based systems, because of the lower granularity of rule executions and their more predictable and localized results, it is less profitable to plan for maximum resource usage at runtime and more effective to design for high resource usage. In the applications discussed in this thesis, maximizing resource usage is primarily achieved by monitoring execution runs and "tuning" the system by focused application of action and match-level parallelism and by increasing the default priority of rules with high branching factors. In this chapter, the discussion of resource maximization is incorporated into the discussions of sequential and heuristic control.

4.1 Rule-Firing Policies

The conventional rule-firing policy of serial production systems is synchronous: all rules that are eligible to fire are identified and placed in a conflict set, conflict resolution is carried out, and the "best" rule is executed. Even in a serial system, such a scheme may involve excessive invocation of the conflict resolution routine if

multiple independent instantiations exist in the conflict set. The simplest modification that can be made to the standard rule firing policy is as follows.

1. Identify all eligible rules.
2. Select a set of independent "best" rules.
3. Fire multiple rules (serially or concurrently).
4. Go to step 1.

Independent rules are those that will not be removed from the conflict set by the action of any other rule in the set of rules chosen to execute. If the selected rules are executed concurrently, this algorithm is the standard rule-firing algorithm proposed for parallel rule-firing systems.

The conventional execute-match-select cycle performs conflict resolution after each rule firing, even though there may be no reason for distinguishing between instantiations. Because the above algorithm eliminates $N-1$ calls to the conflict resolution routine for N rule firings, one could expect a small super-linear speedup over a conventional serial rule-firing policy and this has been observed in programs executed using the UMPOPS scheduler. The effect is quite small because the conflict resolution routine in OPS5 performs only superficial processing and parallel rule-firing imposes small overheads of its own on performance. The true impact of conflict resolution is not necessarily its absolute cost, but rather its effect on the timely execution of rules.

The above algorithm is *synchronous*; it requires that all eligible instantiations be identified before selection of rules and rule firing can take place. Unless all rules require the same time to execute and are initiated at exactly the same time, this synchronization can reduce performance by imposing a potentially lengthy synchronization delay on the system, during which eligible rules may remain in the conflict/eligibility set for a considerable time. To avoid this synchronization delay, I have developed an asynchronous rule-firing policy.

4.2 Asynchronous Rule-Firing

An asynchronous rule-firing policy is one in which eligible rules are executed as soon as they become instantiated. Such a policy maximizes processor utilization because executable rules do not remain latent in the conflict set for long periods of time, however, additional correctness criteria must be developed to support asynchronous execution. It is the desire to reduce synchronization overheads in rule-firing that motivates the attempts in this research to eliminate conflict resolution and interaction detection algorithms. These algorithms require comparisons of eligible rules against all other eligible rules and necessarily imply synchronization. The need for asynchronous rule execution was suggested by the first experiments conducted with UMPOPS.

4.2.1 Experiments with Rule-Firing Policies

The first benchmark program developed for the parallel OPS5 was a very simple circuit simulator. While unimpressive in terms of circuit simulation technology, this application displayed the desirable property of task independence due to the high degree of data parallelism represented by the discrete devices. Conceptually, a circuit simulator is event-driven, with each device capable of being simulated in parallel without reference to other entities in the system. For each type of device in the system, the simulator contains a production which "knows" how to simulate the behavior of that device. The circuit is represented by working memory elements describing the devices, the connections, and the signal values seen on inputs and outputs. At each time quantum (which is equivalent to a production execution cycle), each device is simulated by the execution of one production. Each simulation is followed by a propagation phase in which the outputs of each device are propagated to the appropriate device inputs. The problem displays parallelism in that each device can be simulated independently, but has a sequential component in that inputs to devices must be simulated at time t before the outputs at time $t+1$ can be computed. A circuit simulator requires *instance* parallelism since there may be many instances of the same device in a circuit which require the same knowledge for correct simulation.

The test set consisted of fifteen devices containing a total of twenty seven inputs and required fifteen productions to execute in the simulation phase and twenty-seven in the propagation phase. The system was configured to run using only rule-level parallelism. Because of the relative independence of the productions in this system and their ability to execute concurrently, it was predicted that the performance of the system would be very nearly linear, with the speed of execution being proportional to the number of processors with some penalty due to contention for shared resources within the Rete net.

4.2.1.1 Experiment 1: Explicit Synchronization

In the first experiment (see Figure 4.1), the two phases of the program, simulation and propagation, were synchronized using a working memory element of type *mode* – a conventional OPS5 programming technique. Two "demon" productions detected when it was time to change the mode of the system from **simulate** to **propagate** and vice-versa. The assumption was made that all productions in the conflict set could be executed concurrently and therefore no runtime checking for potential conflicts was performed. If the conflict set contained more instantiations than there were processors in the system, the surplus rules were placed on a process queue and executed as processors became available. The rule execution mechanism was synchronous because it explicitly waited until all productions had completed execution and the system had achieved quiescence before re-examining the conflict set.

The results of the first experiment were disappointing, the speedup due to parallelism was only a factor of three and the utilization of processors was poor. Analysis of the system indicated that the fault lay with the mode-changing productions are responsible for deleting and adding the mode working memory elements. . These productions, by necessity, did not share the conflict set with any other productions and could only be executed serially. Inside the Rete net, these elements act as gates that prevent the matching process from proceeding past a given point and adding instantiations into the conflict set. Thus, execution of the mode-switching productions initiates considerable matching activity, causes many productions to be instantiated, and consumes a disproportionate amount of processing resources. While match parallelism reduced the length of the serial bottleneck, it did not eliminate it entirely. A second experiment was devised to increase the level of asynchronous behavior in the program by removing the explicit working memory-based control.

4.2.1.2 Experiment 2: Synchronization via Conflict Set

In the second experiment (Figure 4.2), the conflict set was used as an explicit synchronization mechanism. The observation was made that the computation was logically divided into phases, with all the rules composing one phase capable of executing in parallel. The purpose of mode-changing is to prevent instantiations from one phase from being prematurely inserted into the conflict set and being executed out of order. However, because rule parallelism allows all instantiations in the conflict set to execute simultaneously, the instantiations belonging to the next phase can be safely added to the conflict set, avoiding the necessity for explicit mode-changing. This approach to synchronization allows a greater proportion of the matching process for the next phase of the computation to take place during the current phase. Because each instantiation in the eligibility set contains a separate copy of the working memory elements that caused it to be instantiated, once a production begins execution it cannot be disabled by changes to working memory.

In this second experiment, the speed of processing increased by an additional factor of two, however, processor utilization remained low. Analysis of the results of this experiment revealed that the bottleneck was the delay imposed by the necessity for achieving quiescence before executing the next round of productions, even though they may have been in the conflict set for some time.

Assuming that all productions in the conflict set can be executed simultaneously and that no conflict resolution need be performed, the time that a production remains in the conflict set is equal to the amount of time between its insertion and the time that the system reaches quiescence and all actions affecting working memory are completed. This time depends on a number of factors: the number of productions being concurrently executed; the number and type of righthand side actions to be performed; and the number of processors available.

For example, consider the case of a 16 processor machine, attempting to execute 17 productions concurrently. Assume each production contains roughly the same number of righthand side actions, and therefore takes roughly the same amount of time, t, to execute. Assume also that each production causes one instantiation to be placed within the eligibility set, each of which can be executed without conflicting with any other. In the time interval $[0, t]$, 16 rule instantiations execute, while one remains in the eligibility set. After time t, 16 instantiations are present in the conflict set, the 17th production is being executed, and 15 processors are idle from time $[t, 2t]$. If the righthand side of the productions contain a significant number of actions, produce output, or perform calls to the operating system, the time t could become quite large. If the RHS actions consist solely of working memory changes, applying the surplus processors towards low-level node parallelism can reduce t, however, my experience has been that the degree of effective node parallelism in problems that support multiple production firings is fairly low.

4.2.1.3 Experiment 3: Asynchronous Production Execution

The final version of the experiment (Figure 4.3) was optimized for maximum asynchronous behavior. Since any time that an eligible production spent in the eligibility set is time wasted, a new scheduling policy called "fire when ready" was devised. In this scheme, the eligibility set was continually monitored; whenever a new production entered, it was immediately fired. The OPS5 code implementing the simulator had to be substantially re-written in order to support this level of parallelism. This approach to scheduling did not employ conflict resolution, so it was no longer possible to guarantee that rules would be executed in any given order. Therefore, the rules had to be rewritten to be self-synchronizing, that is, to examine the appropriate working memory elements for ordering information (explicit timetags) to ensure that the simulation of devices proceeded in the correct order.

Because the computation was asynchronous, it was possible that some working memory elements (representing inputs) could be modified before all the devices connected to those inputs had been simulated. To avoid this problem, separate working memory elements representing successive inputs to a device were created, rather than modifying the existing elements. This created a potential for explosive memory growth and required an architecture in which working memory management was knowledge-based and elements were only deleted when it could be guaranteed that they would no longer be needed.

This asynchronous approach to rule firing resulted in high levels of processor utilization (nearly 100%), and a system which could achieve much greater parallel speedups (a factor of 12 with 14 processors executing rules). Considerable extra matching and production execution took place to synchronize the computation and to garbage collect unneeded working memory elements; unlike the previous experiments, additional processors would have resulted in increased performance.

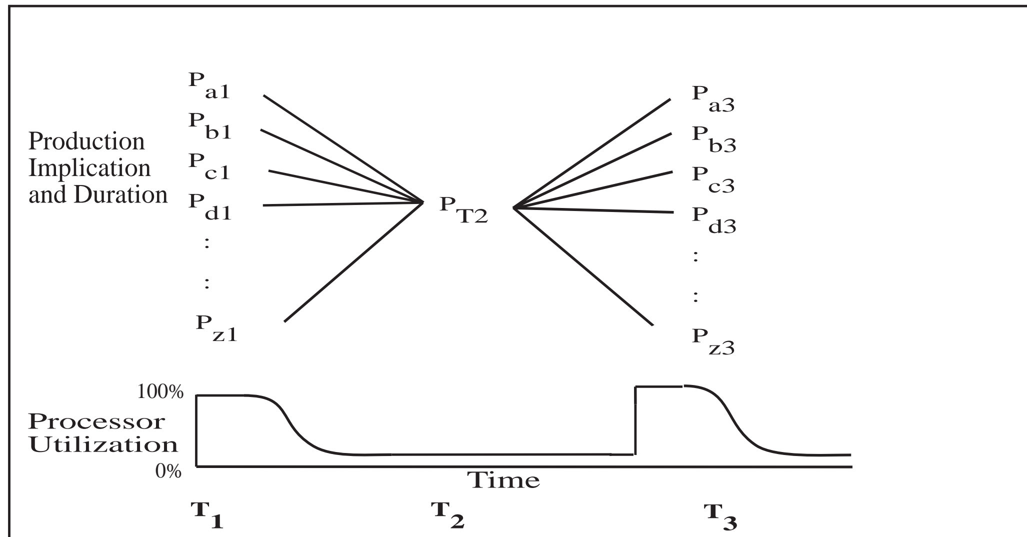


Figure 4.1: Processor utilization for a synchronous rule-firing policy with bottleneck mode-changing rules.

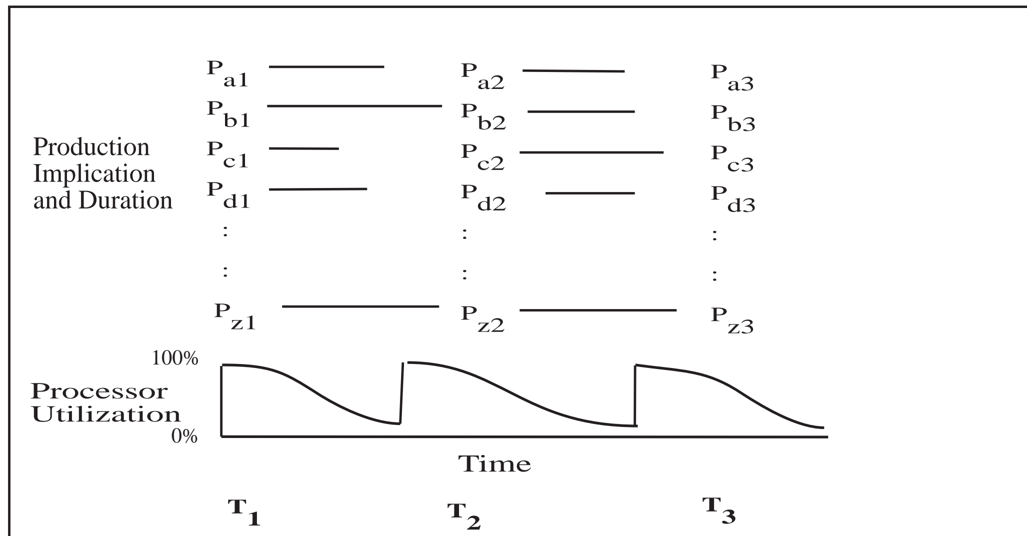


Figure 4.2: Processor utilization for a synchronous rule-firing policy with mode-changing rules eliminated. The delays due to conflict resolution are exaggerated for illustrative purposes.

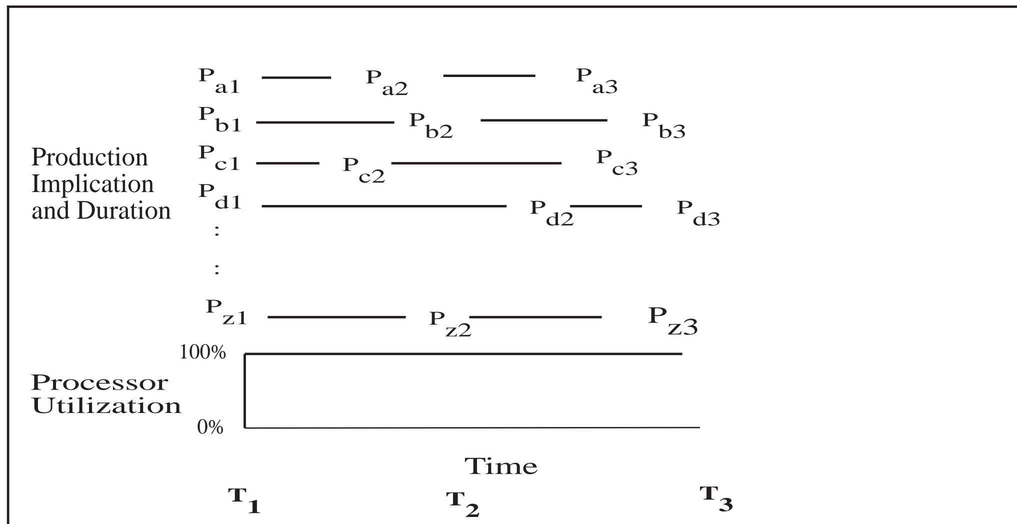


Figure 4.3: Processor utilization for a fully asynchronous rule-firing policy.

4.2.1.4 An Experiment with Unbalanced Rule Execution Times

The influence of the synchronizing effect of conflict resolution is most apparent when the execution time of rules is not uniform, causing instantiations stimulated by faster rules to wait for the slower rules to terminate. The simulation rules in the circuit benchmark are all homogeneous and require approximately the same amount of time to execute. In order to emphasize the effect of synchronization on parallel performance, a delay was inserted into the righthand side of a single rule. The magnitude of the delay was approximately that of the rule's execution time. As can be seen in Figures 4.4 and 4.5, the processor utilizations and speedups obtained by synchronous and asynchronous rule-firing policies vary dramatically.

4.2.2 Summary of Experiments with Asynchronous Rule-Firing

The experiments with the circuit benchmark demonstrated that performance could be greatly increased in a production system by eliminating the conflict resolution stage and executing productions asynchronously in parallel. This improvement was gained at the cost of considerable extra programming effort. The asynchronous version of the benchmark program was significantly more difficult to write and debug than the conventional "lockstep" version. It was the results of these experiments and the desire to simplify the task of programming asynchronous rule-based programs which led to the development of the UMPOPS scheduler described in the following chapter.

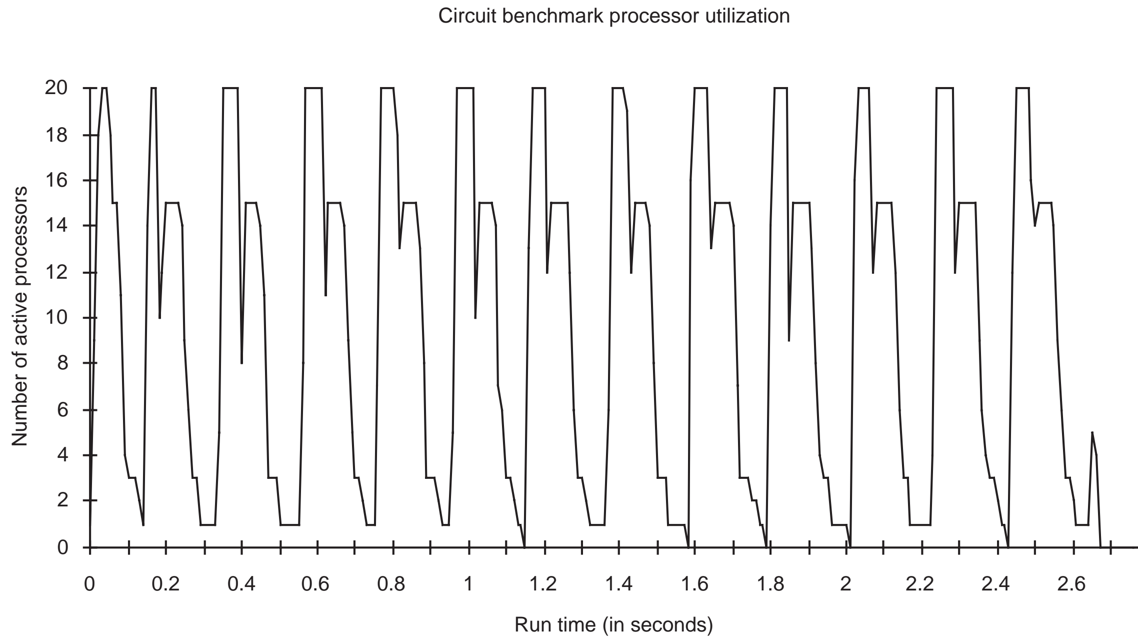


Figure 4.4: Processor utilization for the circuit benchmark with rules of unequal run time, using a synchronous rule-firing policy.

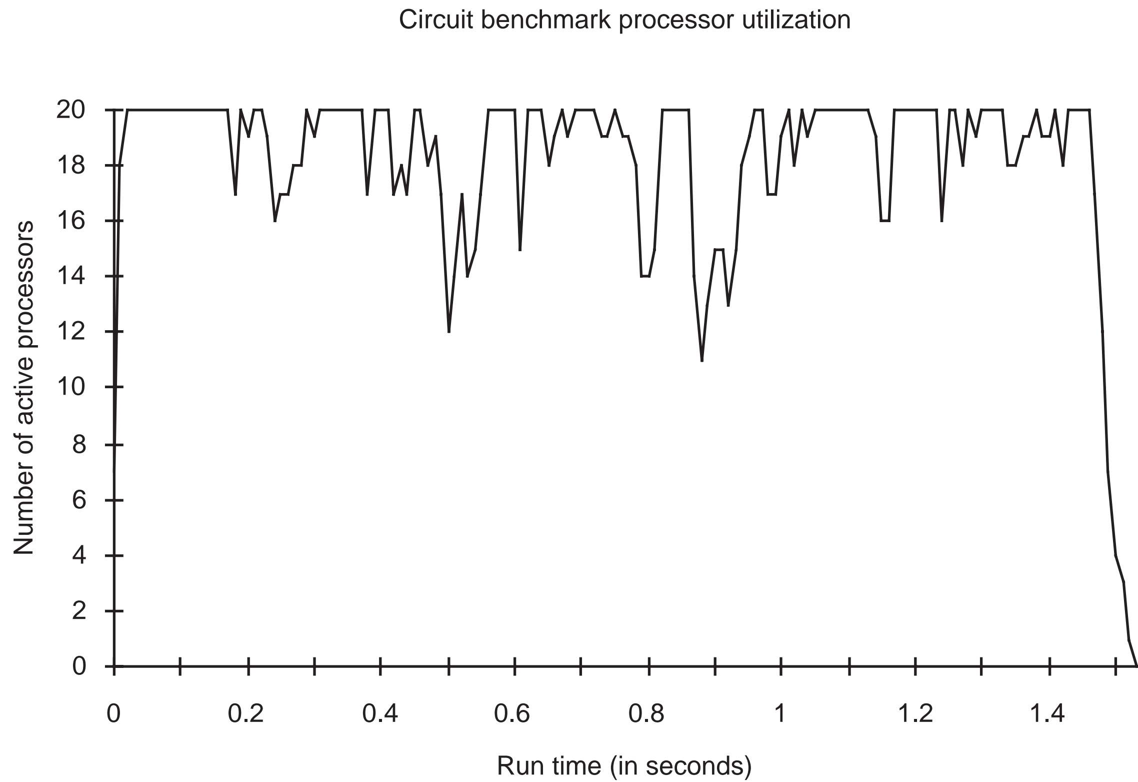


Figure 4.5: Processor utilization for the circuit benchmark with rules of unequal run time, using an asynchronous rule-firing policy.

4.2.3 Monotonicity in the Eligibility Set

Asynchronous rule execution adds a new constraint to program correctness. Since rules are fired as soon as they become eligible, rules must enter the eligibility set monotonically, that is, any rule which becomes eligible to fire due to a working memory change should not be disabled by any *concurrent* matching activity. If the entry of a rule instance into the eligibility set is not *monotonic*, the use of an asynchronous rule-firing policy may result in the execution of transient entries into the eligibility set. A transient entry in the eligibility set is an instantiation that is created due to some transition state in working memory between the deletion/assertion of one working memory element and the assertion/deletion of another. Such transient entries are common in serial or synchronous rule-firing versions of OPS5 but go unnoticed because the eligibility set is only examined after each rule has completely executed and all match activities have terminated.

As an example of a non-monotonic insertion in the eligibility set, consider the following rule. When the accompanying changes to working memory are made, a transient production instantiation appears in the eligibility set. Because this instantiation will be immediately disabled by a concurrent working memory change, it should not be executed.

Rule:

```
(p example-prod
  (mode)
 -(marsupials ^field2 wombat)
  -->
 ...)
```

Working memory changes:

```
; ---> example-prod into eligibility set.
  (remove marsupials ^field1 wallaby ^field2 wombat)
; <---  example-prod out of eligibility set.
  (make marsupials  ^field1 koala ^field2 wombat)
```

standard-ls

The changes in the above example are exactly those which would take place by the execution of the OPS5 statement `(modify <marsupials> ^field1 koala)` in the righthand side of some rule.

Failing monotonicity in the eligibility set, *transient* rule instantiations should be detected and prevented from firing. Working memory locks will prevent the execution of transient rules that would be removed from the eligibility set due to the removal of a working memory, however, for reasons described in chapter 3,

transients caused by the addition of elements that are referred to negatively in the LHS of a rule will not be prevented from executing. Monotonic insertion of rules can often be ensured simply by careful ordering of working memory changes in the RHS and the avoidance of `modify` operators in favor of successive `make` and `remove` operations. However, this requires that the programmer be aware of potential transient instantiations and act to eliminate them. Transients may also occur in the eligibility set during the course of concurrent working memory changes carried out through action parallelism; because the ordering of parallel changes to working memory is nondeterministic, race conditions may occur. To avoid such race conditions, local synchronization operators have been added to UMass Parallel OPS5; these operators are described in detail in Chapter 5.

4.3 Control Tasks

It is likely that the programmer will occasionally wish to perform synchronous conflict resolution within the context of certain *tasks* or groups of rules, while allowing other activities to take place asynchronously. A specific task may require a synchronous rule-firing policy under circumstances in which:

- each operator in the task has a specific preferred order,
- applying one operator invalidates all others in that task, and
- performing search by applying operators in parallel would be prohibitively expensive due to copying or computational costs.

An example of an application which requires such a rule-firing architecture is the Alexsys system developed at Columbia University [Stolfo *et al.*, 1990, Stolfo *et al.*, 1991b]. A discussion of experiments with this system appears in Section 6.3. The construct that UMPOPS provides for specifying multiple independent control tasks is discussed in this section; the syntax and invocation of control tasks is described in Section 5.3.3.

4.3.1 Defining Tasks and Task Quiescence

A task can be defined informally as a named subproblem within the scope of a larger computation. Rules executing in different tasks can fire asynchronously; however they are not guaranteed to access discrete resources, so locking of working memory elements accessed within tasks is necessary. This possible interaction between tasks distinguishes UMPOPS's tasks from Miranker and Kuo's notion of a set of independent clusters firing asynchronously [Miranker *et al.*, 1989, Kuo *et al.*, 1991]. A task is a *control* mechanism that defines the context in which a computation's conflict resolution routines (if any) and rule-firing policy are defined and in which a local quiescence may be determined. Multiple rules within a task may execute concurrently; this is dependent only on the conflict resolution routine assigned to that task; if the routine returns multiple instantiations, they all may

fire. Rule-firing within a task may be asynchronous (in which case no conflict resolution routine is defined) or synchronous (in which case, a situation specific conflict resolution routine may be specified, or a default may be used).

Control activities in tasks requiring synchronous conflict resolution can only occur once the task has become *quiescent*. The quiescence of a task can be defined as a state in which all current computation corresponding to that task has been completed. Because of the data-directed, pattern-matching nature of rule-based systems, determining the quiescence of tasks (or even the scope of tasks) is non-trivial. Certainly if all working memory changes in the entire system have become quiescent, then we can say that a particular task is quiescent and that all relevant operators corresponding to that task have been determined. However, in a system in which tasks are executing asynchronously with respect to each other, it is unreasonable to expect system-wide quiescence to occur. Thus, we have the problem of determining whether a working memory change which is currently occurring is part of the current task. In general, this can only be determined by waiting to see if that element matches against a rule that also matches against an element previously determined to be in the task.

For example, consider the instantiation below:

Rule:

```
  (P find-wombat
    (task ^name find-a-wombat)
    (marsupial ^type wombat ^name <name>)
-->
   ...)
```

WM:

```
  (marsupial ^type wombat ^name Keith)
```

standard-lsIf the working memory element `(marsupial ^type wombat ^name Keith)` is in the process of being asserted, then the task defined by the element `(task ^name find-a-wombat)` can not be said to be quiescent. Dynamic rule conflict detection cannot be used to detect the quiescence of tasks, not only because of the overhead, but because new rules may continuously be created (or about to be created); thus once again, only global synchronization would allow run-time consistency checking to accurately determine whether a task was quiescent.

We must assume that for the large part, working memory is quiescent and that all active working memory changes can be annotated as directly affecting particular tasks. This is done by starting with a quiescent working memory. Goal elements are created in the context of a task; a goal element is simply any element that is specifically annotated as belonging to a task and that stimulates further rule firings.

From that point on, any rule stimulated by the goal element is considered to be executing in the context of that task, and any working memory changes stimulated by that rule are in the context of the task. Quiescence is achieved when all working element changes being performed within the context of a task have terminated.

Because all rules execute in the context of the task that created their stimulated elements, communication or sharing of data is difficult to arrange. (The task mechanism is still experimental and not all of the necessary mechanisms have been developed.) Currently, all initial working memory elements are created in the context of an initial task. All tasks are allowed to access these elements. If any of these elements are modified by a control task, the resulting element remains in the context of the initial task and elements may be asserted into the context of the initial task. This mechanism allows tasks to communicate through working memory, but means that quiescence of tasks is only partial; quiescence of the initial database pool cannot be guaranteed. The problems of identifying quiescence of tasks and managing the communication of data between tasks remains a open research issues.

4.3.2 The Task Implementation

Incorporating tasks into the UMPOPS rule-firing architecture was straightforward. The rule firing routines were modified to annotate rule instances and working memory elements according to their parent tasks so that they would be executed and asserted into the proper context. Each task is given its own conflict resolution routine, which is to be applied only to rule instantiations within the context of that task, thus, each task must be assigned its own conflict/eligibility set. A new eligibility set structure was devised for UMPOPS; the eligibility set is now an array of sets, with the index of each set being uniquely assigned to a task. Various initialization routines were modified to ensure that the new data structures are initialized and reset properly. The conflict set reporting routines were modified so that they correctly interpret and print out the contents of the various conflict sets. The scheduling process was modified to monitor the new conflict set array and perform conflict resolution and scheduling of rule instances within a task when, and only when, that task achieves quiescence. Because tasks do not guarantee data independence between tasks, conflicts may arise between rules in alternate tasks and a single scheduler/lock manager is required (see Figure 4.6). If tasks can be defined in such a way as to guarantee that there will be no interactions between rules executing in the different contexts, then multiple scheduling processes may be used, avoiding potential serial bottlenecks which could occur when multiple tasks became quiescent simultaneously (see Figure 4.7). Locks may still need to be acquired if multiple rules with a task context are allowed to fire concurrently.

4.3.3 Summary: Control Tasks

The control task is a flexible construct which allows a parallel rule-firing system to pursue multiple independent activities, each of which possesses its own conflict resolution routine and appropriate rule-firing policy. The propagation of task

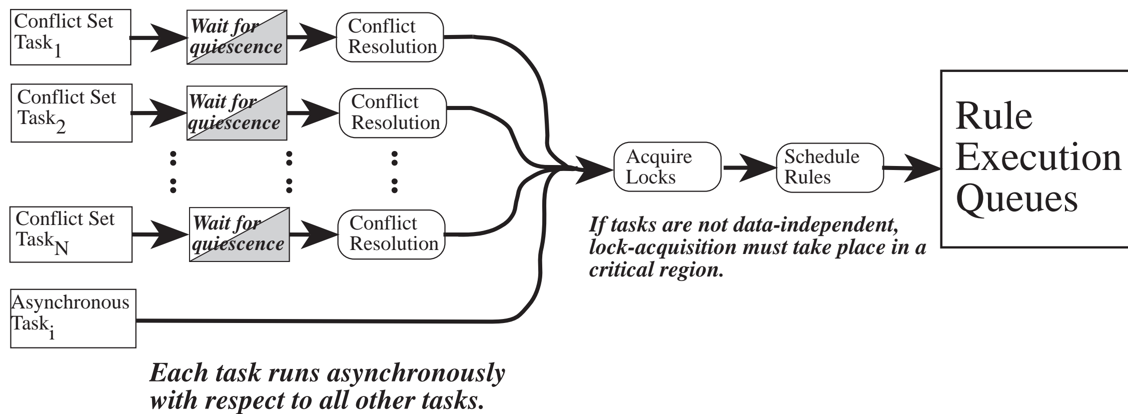


Figure 4.6: The rule-firing architecture required to implement multiple asynchronous control tasks.

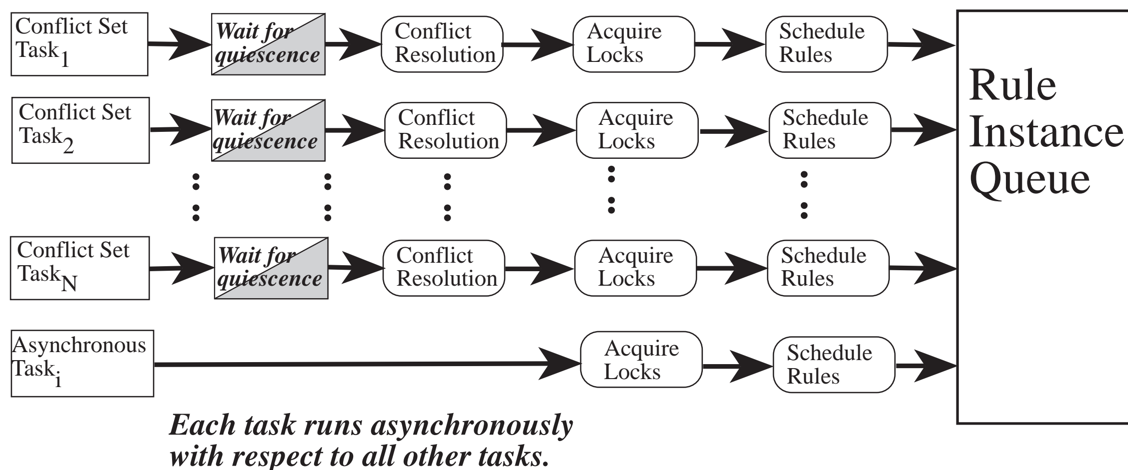


Figure 4.7: If data independence can be insured, then multiple scheduling processes can be assigned to tasks, avoiding potential serial bottlenecks in the scheduling phase.

contexts through the use of working memory elements allows at least a partial quiescence to be defined and determined for each task. If tasks can also be guaranteed to be fully data-independent in a specific application, complete local quiescence of a task can be determined.

4.4 Sequential Control

This thesis does not attempt a complete solution of the rule-sequencing problem, that is, ensuring that rules can be coerced into firing in the correct order at all times. This subject has already been adequately addressed in previous research, e.g. [Georgeff, 1982, Friedman, 1985], and commercial production system languages such as OPS83 [Forgy, 1984] routinely provide access to imperative control constructs. This section is concerned with those aspects of sequential control which are amenable to parallel treatment, in particular, language mechanisms are proposed for compressing multiple rule firings into a single rule execution whose runtime can then be decreased using match or action-level parallelism.

The main objection to enforced rule sequencing is the overhead that it imposes on the computation in terms of extra rule instantiations, matching, and, in conventional production systems, conflict resolution. In parallel-rule-firing systems, the main objection is that forcing rules to fire sequentially removes any potential for parallel activity and reduces processor utilization. To reduce these effects, UMPOPS incorporates modifications to the rule syntax to allow *set-oriented* rule-firing as well as RHS iteration and mapping operators. Simply allowing complex iterative operations to be performed by a single rule execution eliminates considerable overhead because repetitive match, rule-instantiation, and scheduling operations need not be performed between actions. In addition, it is frequently possible to further reduce the serial overhead of such operations by employing *action* parallelism.

Consider the typical mapping or iteration operator that occurs in the righthand side. Assume that the operator simply modifies working memory and performs no output. The iteration operator then proceeds as follows:

1. Set $i = 1, Limit = N$.
2. Process data element i.
3. Assert new element (remove/modify old element).
4. Wait for completion of assertion.
5. Increment i.
6. If not $i = N$, go to 2.

This iteration operation will require a time equal to N working memory modifications $+N$ processing steps. We can assume that in most cases, the match time will far exceed the processing time. It is frequently the case that the order in which elements are asserted to memory is not important, thus, by performing the modifications to working memory in parallel, the execution time of the rule can be

reduced (roughly) to N processing steps $+N$ action-spawning steps $+1$ matching step. In the case of large databases and high matching costs, this will approach an N times speedup. Even though only a single rule is being fired, we can come close to our goal of achieving full processor utilization and corresponding speedup through the application of action parallelism. Depending on the nature of the changes being made to working memory, it may be necessary to synchronize after the execution of this rule to ensure that all matching has been completed before additional rules are allowed to fire.

4.4.1 Set Functions

Set-oriented constructs that use relational DBMS-like semantics to compress many rule firings into one have been suggested by several researchers, notably in the Herbal language [van Biema *et al.*, 1986] and more recently in [Widom and Finkelstein, 1990] and [Gordin and Pasik, 1991].

In an instance-based implementation of a rule-based system, one and only one rule instantiation is created for each set of working memory elements that match a lefthand side. That is, if the lefthand side of a rule has N positive condition elements, then the resulting instantiation will refer to N working memory elements that match those conditions. A set-based rule modifies this scheme by allowing the programmer to specify that certain groups of condition elements on the lefthand side should be considered as sets. Therefore, the resulting instantiation can be considered as a set of all the instantiations which would have normally been matched by an instance-based system. By creating righthand side functions that operate on these functions, many algorithms can be represented with greater efficiency and a reduction in rule-firings. UMPOPS has been modified to support a simple set-oriented syntax. The implications of set-oriented rules on control and rule-firing policies are discussed in this section, while the syntax of the set mechanisms is described in Section 5.3.4.

The principal influence of set operations on synchronous parallel rule firing is the reduction of rule firings devoted to fundamentally serial algorithms. For example, many counting and marking algorithms can be performed by a single set-oriented production. This simplifies programming and eliminates the need for many programming idioms which depend on conflict resolution to succeed. Because the overhead of rule instantiation and invocation is reduced for set rules, the waiting time for any rule that must synchronize with the counting or marking task is reduced. However, individual set-oriented rules, because they must operate on greater amounts of data, may take longer to execute than instance-based rules.

4.4.1.1 Set-oriented Rules and Asynchronous Firing Policies

Implementing set-oriented rules in an asynchronous rule-firing system is somewhat more difficult than in the synchronous case. Set construction proceeds incrementally as one or more working memory elements that stimulate the set rule in question propagate through the network. An asynchronous rule firing scheme may attempt to execute a set-oriented rule before the set has been completely

constructed. This could result in the execution of the set rule multiple times with different data sets, causing inconsistencies within the database. Therefore, it is necessary to determine when the working memory changes that affect a particular instantiation of a set rule are complete and to fire the instantiation only at this time. The problem of associating working memory changes with a particular set-oriented rule instantiation is similar to that of associating working memory elements with particular tasks or of associating rule instances with a particular conflict set.

In the UMPOPS system, this problem is solved by using a signalling mechanism. It is assumed that each set operation has some trigger; that is, that the set rules are in some sense goal-oriented. Before the triggering element (or elements) is (are) added to working memory, the beginning of a *synchronization group* is signalled. After the working memory modifications, the end of the group is signalled. If any set rule is triggered during the period during which the synchronization group is active, it is placed on that group's completion list. The rule instantiation is placed in the execution queue as soon as all the working memory changes invoked within the task become quiescent and completion of the group is signalled, indicating that no more working memory changes will be initiated. If it can be guaranteed that all data relevant to the set activation has been asserted before the triggering element, then the synchronization mechanism can be simplified by checking for quiescence of that single goal element. This concept is explored in more detail in the next section on computational phases.

The idea of task synchronization does some damage to the pure notion of data-directed programming embodied by rule-based systems. In order to create a task group, it is necessary to know in advance when a working memory element being created is likely to trigger a set-based rule. In practice, this syntax is no more contrived than the use of conflict resolution techniques to perform iteration. The main advantage of the synchronization group mechanism is that it eliminates the need for achieving global quiescence within the rule-firing system and replaces it with a mechanism for detecting local quiescence within a set of actions. The overhead of synchronization is therefore limited to just the set of rules affected by the working memory changes within the synchronization group.

4.4.1.2 Acquiring Locks for Set-Rules

The working memory locking scheme must be modified to accommodate set-oriented rules. If a single instantiation within a set of instantiations if prohibited from executing, it should not be necessary to terminate the entire set of instantiations. Instead, the lock algorithm for set-oriented rules is implemented as follows:

1. Identify those working memory elements which are common to instantiations in the set.

2. Attempt to acquire locks for all common elements. If the lock attempt fails, terminate rule.

3. For each instantiation within the set, attempt to acquire locks for elements matched only by that instantiation. If attempt fails, remove the instantiation from the set.

4. Execute the set rule containing all the remaining instantiations.
5. As each set instance terminates, release the read locks required by only that instance.
6. After all set instances have been executed, release all locks acquired in step 2.

4.4.2 A Model of Program Phases

OPS5-like languages typically use *mode* elements to control the enabling of rules in different logical phases of a computation. Although the primary use of mode elements has been to micro-control the execution of productions to implement iteration operators, once these uses have been superseded by more sophisticated RHS operators, a more meaningful definition of program phases becomes possible.

Consider the following problem, given a set of rules which detects and acts upon states of working memory produced in a previous computational phase, when should those rules be allowed to fire? In a purely asynchronous paradigm, the rules are allowed to fire as soon as they become enabled. This firing model is appropriate as long as the implications represented in the rule set are of an existential nature, i.e., if there exists a specific condition, then execute. However, if negated tests are present in the rules, then a test for non-existence of a condition is being made. It may not be possible to determine in advance whether that condition will be asserted by the previous phase. In fact, the only way to ever be certain that it is correct to fire rules with negated elements under these circumstances is to explicitly state that no further elements will be asserted. Thus, we can assign a *semantic* meaning to the assertion of a mode-changing element; the rule which creates that element is stating that no further changes will be made to the database which will affect the subsequent processing phases of the computation.

4.4.3 Mixed-Mode Parallelism and Mode-changing

The division of programs into logical phases marked by the production of closed-world bodies of data makes it difficult to avoid the use of mode-changing productions. These productions fill a necessary role in determining when the matching should occur for groups of productions; if all groups were always activated, rules would be triggered at inappropriate times and there would be no way of focusing matching activities on only the rules of current interest. As an unavoidable consequence of such "switched matching" schemes, there will occur periods in the computation when working memory elements are asserted which will stimulate large amounts of matching activity.

Because the order of matching within the Rete net is determined by the order of the condition elements within the rule, the traditional location of the gating condition element as the first element in the rule prevents partial matching from occurring between other elements within the rule. This causes a significant amount of matching to take place once the gating element finally arrives (see Figure 4.8A).

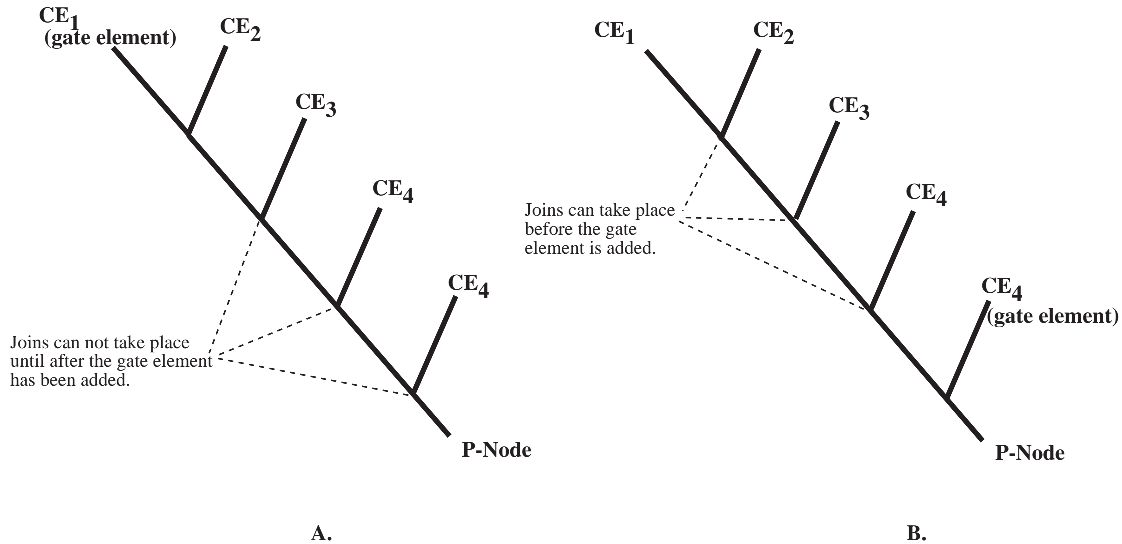


Figure 4.8: The location of the gating element affects the amount of partial matching which can take place in the match process.

The delay due to the gating effect can be minimized by placing the gating element as the final positive condition element in the rule (see Figure 4.8B). This positioning allows more partial matching to occur before the gate element is created. This technique is only applicable if the gating element is not used to pass parameters to the rule, that is, no field in the gating element may be unified with any field in any other condition element of the rule. The problem with this re-ordering of terms is that placing the gating condition as the final element may cause the rule to partially match in situations in which the rule is not applicable. The overhead caused by this unnecessary matching may exceed the advantage gained in minimizing rule activation time by changing the placement of the gating element. The exact nature of the trade-off can only be determined by examining each case separately.

Parallelism can be used to minimize the delays caused by gating in two ways. Because a gating element typically affects many production instantiations, match-level parallelism can be effective in minimizing the time consumed in such match-intensive working memory changes. An asynchronous rule-firing policy will allow productions enabled by the addition of the modal element to be executed incrementally during the execution of the mode-changing rule, thus maintaining processor utilization. These two techniques are mutually conflicting; using processors for matching purposes may result in no free processors being available to execute rules. Because match-level parallelism occurs at a much finer level granularity than rule parallelism and incurs a proportionately higher cost, the primary level of parallel activity should probably revert to the rule level as soon as practical. Managing the trade-off between rapid matching of the elements created by a single bottle-neck rule and concurrently executing rules stimulated by that rule remains a subject for some experimentation; it is not clear whether all resources should be devoted to parallel matching until the bottleneck rule has completed execution or whether some resources should be

diverted to rule execution. It is likely that this is situation-dependent and the architecture should allow the ability to specify either option.

4.4.4 Optimistic Concurrency

One alternative to mode-changing that has been speculated on by Gupta is to allow *optimistic concurrency*; that is, to allow enabled rules to fire before a guarantee of quiescence of the previous processing phase is available. If new data arrives which invalidates a rule firing (or an entire train of inference), the modifications to working memory made by those rule-firings and any current instantiations stimulated by them must be retracted. This scheme results in the maximum degree of responsiveness in a parallel system, but requires a truth-maintenance system of some complexity in order to retract the results of rule firings. A scheme for performing such truth maintenance was described in [Wolfson *et al.*, 1990] and similar techniques have been used in the parallel asynchronous simulation of electronic circuits [Fujimoto, 1990].

4.5 Heuristic control

One of the primary observations arising from this research is that any control mechanism that requires synchronization or examination of all eligible operators will seriously degrade the potential parallelism in the system. Currently, heuristic control in rule-based systems has always been performed as a separate process that takes place after all eligible rules have been identified. In the pursuit of ways of reducing serial overhead, we can question the assumption on which this is based: is it really necessary to identify all alternatives before making heuristic control decisions? The thesis to be explored in this section is that heuristic control can be achieved through an incremental process that takes place concurrently with problem-solving.

The scheduler of UMPOPS has been modified to support heuristic control, that is, rules can be either pruned or ordered according to heuristic evaluation functions. To allow rule firings to be prioritized, multiple execution queues are provided and each is assigned a priority. Rules assigned to a low priority queue are executed first. This is implemented by insuring that the rule "demons" which monitor the queues first look for tasks to perform in the lower numbered queues. Each rule *type* is assigned a static priority at compile time and each instantiation of that rule is placed in the appropriate queue. In order to allow prioritization within rules of the same type, individual rule queues can themselves be declared to be priority queues. Rule *instances* are rated by rule-specific evaluation functions and inserted into the appropriate priority queue. Much of this control information is specified within the rule using the `meta` construct; the exact syntax will be described in the Section 5.2.1.

There are several points in the match-schedule-execute cycle in which heuristic control can be performed. First, heuristic control can be performed *de facto* during the rule matching phase; that is, rules may be heuristic in nature, Writing rules in such a way as to eliminate unnecessary rule instantiations will minimize the number of rules which must be scheduled and processed by the control demon. The next

opportunity for control is during the "pre-execution" phase. This phase needs some explanation. After the rule is placed on the eligibility queue, it is scheduled by a dedicated process. The heuristic scheduling functions need information in order to rate the rule, and much of this information is carried in the variables bound by the rule instantiation. In OPS5, this information would normally not be available without executing the righthand side, however, executing the righthand sides of some rules is just what we are trying to avoid by the incorporation of heuristics. To acquire rating values, UMPOPS does a "pre-evaluation" at instantiation time (i.e. when the rule is inserted onto the eligibility queue) that extracts the variable bindings for the instantiation and stores them. No righthand side actions are actually executed during this phase and the cost is low compared to the actual righthand side execution. If any rule-specific control functions (specified as meta-information) need to be executed, they take place during the pre-evaluation phase. The rule is then placed on the eligibility queue. If the system is in an asynchronous rule-firing mode, the rule instance is immediately scheduled by the control process and placed in the execution queues according to its rating and queue priority. In a synchronous system, conflict resolution is performed on the rule set once quiescence has been achieved.

The final opportunity for heuristic control occurs immediately prior to a rule's execution. Once a rule is placed on the execution queue, it is theoretically executed immediately. However, given the pragmatic limitations on processing resources, in many applications, the number of rules to be executed will temporarily exceed the number of processors available to execute them, and rules will remain on the execution queues for potentially extended periods of time. During this time, the state of the system can change, causing a rule instantiation on the queue to become redundant. Control functions can be attached to rules to determine whether the rule is still valid given the current state of the system. The control functions must be extremely fast and simple if they are not to decrease performance, so they are typically designed (in UMPOPS) to simply check a value being asserted by the rule against a global variable. The control function also resets the value of this global variable if appropriate. (It is a problem in OPS and production systems in general that working memory cannot be easily accessed from procedural code, and data must sometimes be stored redundantly in order to be accessible from both rule-based and imperative code.)

When a rule demon removes an instantiation from the queue, it checks to see whether the rule has an attached control function. If so, the demon executes the control function in the context of the rule instance's righthand side. If the function returns a non-nil value, the rule is executed, otherwise it is killed. Killing a rule does not necessarily mean that no action is taken – there may be some clean up operations associated with the state represented by that rule. The programmer has the option of attaching "kill actions" to the rule which are executed if, and only if, the rule is pruned by a control function.

Control functions are specified as arbitrary Lisp functions. This implementation is not so much as a statement of position on the procedural versus declarative debate as the result of experimentation with both rule-based and function-based control. Programming control constructs in OPS5 proved to be extremely cumbersome; the

pattern-matching syntax of OPS5 is not sufficiently expressive to support arbitrary control functions, and the overhead of the matching/rule-firing cycle results in a non-responsive system. Once again, deliberative control only appears to be suitable for actions of a high level of granularity.

4.5.1 Dynamic Control of Rule-Firing Policies

Simply assigning static priorities to rule *types* and control functions to rules that do not change during the course of a computation is insufficiently flexible to provide truly sophisticated control. It is possible, albeit awkward, to dynamically modify the priorities and control functions associated with rules during execution. For example, special control *meta-rules* can be devised that modify the control characteristics of a system when a particular state occurs. Such an approach was used in the BB1 blackboard architecture which employed a single knowledge-based paradigm for representing both control and domain knowledge [Hayes-Roth, 1985].

Such meta-control rules should, of course, be given the highest execution priorities. Even so, it is likely that queue and execution latencies would render such control rules less responsive to the state of the system than is desirable. That is, if the meta-rules were required to undergo the same locking and scheduling processes as domain rules, they would not execute for some time after the triggering event which created them. A modification to UMPOPS is being contemplated that would allow true control rules to be distinguished. Because the principal activity of such meta-control rules would be to send messages to the scheduler (which is very inexpensive as compared to modifying working memory), control latency could be minimized by sending the control messages during the pre-execution phase (i.e. *immediately* after the control rule becomes enabled) and before the rule enters the eligibility set. Because the new control state of the system would likely have to be mirrored in working memory to prevent iterative firings of the meta-control rules, the meta-control rule would still have to be executed in a conventional fashion. An alternative scheme is to simply bypass the scheduling mechanism and place control meta-rules on the highest priority execution queue.

Another issue in the dynamic control of rule-firing policies is the apportionment of rule demons to the various execution queues. The default assignment is that the highest priority rules get executed first, however this is an *unfair* scheduling policy and it is possible that rules on lower priorities queues could be "starved" if large numbers of high priority rules were to arrive. UMPOPS allows the user to specify a queue examination protocol when invoking rule demons; the demons will examine the rule queues in the given order. If more than one protocol is given, the protocols will be divided evenly among rule demons. It is possible, for example, to assign a single rule demon to monitor a single queue or a range of queues, and to specify the order in which they are visited. The programmer can also specify whether the queue protocol is fair, that is, whether after executing a rule, a demon should begin at the highest priority queue or whether it should visit all queues in its protocol before restarting its traversal of the execution queues.

4.5.2 Interactions between Consistency Maintenance and Heuristic Control

The heuristic control mechanisms can interact with the working memory locking scheme described in Chapter 3 in potentially pathological ways. The first problem occurs when a rule that has acquired the necessary working memory locks is then pruned by a heuristic control mechanism. There will always be an interval between lock acquisition and the pruning. During this time, it is possible that another rule which is capable of satisfying the heuristic will become eligible to fire. Because the first rule has acquired the necessary working memory locks, this competing rule will be prevented from executing and will be removed from the eligibility set by the lock manager. If the first rule is pruned, neither rule will ever execute and the result of this sequence of events will be that the appropriate action never takes place. Simply reversing the order in which lock acquisition and heuristic control takes place will not solve this problem, and performing both operations simultaneously would require delaying lock acquisition until execution time and performing lock management within a critical region. One solution (not yet implemented) is to modify the lock management routines so that rules that are locked out due to competing write operations are not eliminated from the eligibility set until the competing rule has successfully begun execution.

A similar problem arises when an asynchronous rule-firing policy is employed. Because rules are scheduled on a first-come-first-served basis, standard conflict resolution techniques in which all eligible rules are ordered and the "best" rule is selected cannot be applied. It is possible, therefore, that a rule will be selected to be fired and acquire all working memory locks only to have a heuristically superior rule arrive in the eligibility queue. If execution queue latencies are short, then the second rule will simply be disabled by the first rule as it changes working memory; if the rules are designed correctly, the change to working memory will restimulate a similar instantiation and the superior result will eventually be reasserted into working memory at the cost of some delay. If execution queue latencies are long, the rule that asserts an inferior answer may remain in the execution queue for a long time. This will block the assertion of the superior result and reduce the efficiency of the heuristic pruning mechanisms by allowing more inferior solution paths to be explored during the extended period in which the better solution is blocked. A solution (also unimplemented) to this problem is to allow heuristic override of locks. That is, one could record the identity of rule instances that have acquired locks to working memory elements. If a rule attempts to acquire a lock on that element and finds that it is possessed by another rule, then the following algorithm could be performed.

1. Check to see if the blocking rule is currently executing. If so, fail.

2. If the blocking rule is still in the execution queue, mark it as temporarily non-executable.

3. Compare the blocked and blocking rule using a situation specific conflict resolution function.

4. If the blocked rule is superior, mark the blocking rule as "killed", release its locks, and schedule the new rule. If the blocking rule is superior, mark it as executable and remove the blocked rule from the eligibility queue.

Both of the mechanisms discussed in this section for reducing the interaction between working memory locks and heuristic pruning would require that the scheduler perform additional bookkeeping and would consume additional memory; in the interests of keeping the scheduler simple and fast, neither mechanism has been incorporated into UMPOPS.

4.6 Conclusion: Control of Parallel Production Systems

This chapter has reviewed the major areas of control in parallel rule-firing systems: rule-firing policies, sequencing of rules and processing phases, heuristic control, and, incorporated into the discussion of the other areas, the optimization of resource usage through the use of mixed-mode parallelism and the appropriate assignment of rule priorities. The uses of the control constructs discussed in this chapter are illustrated in the benchmark programs of Chapter 6.

CHAPTER 5

UMass Parallel OPS5

This chapter describes UMass Parallel OPS5 (UMPOPS), a Lisp-based version of OPS5 which has been modified to support a number of levels of parallel activity: *rule* parallelism which allows rules to be fired concurrently, *matching* parallelism which allows the pattern matching to be performed in parallel, and *action* parallelism which allows individual working memory changes to be made in parallel. UMPOPS contains many changes to the basic syntax and features of OPS5 which were added to support parallel activity and to allow more versatile control of rule execution and sequencing.

5.1 The Rule-Firing Architecture

The architecture of OPS5 had to be modified considerably to support parallel rule-firing. Although simple in concept, the parallel rule-firing architecture had to fulfill the following requirements:

- The architecture was required to support multiple rule-firing policies including parallel asynchronous, parallel synchronous, task-based with multiple conflict sets, and serial.
- It was necessary to support heuristic scheduling and pruning of rules at multiple points in the rule-firing cycle.
- A lock management scheme had to be incorporated so that eligible rules could be checked for interactions before execution.
- In order to satisfy the requirements of heuristic control and allow for efficient utilization of processing resources, it was necessary to be able to prioritize rule executions so that more important rules would be executed first during conditions of processor saturation.
- The architecture was required to be flexible so that experiments could be performed with varying scheduling and locking policies.

- The rule-firing mechanism was required to be instrumented so that information about processor usage, contention for resources, and rule execution times could be extracted.

In accordance with these requirements, the rule-firing architecture of UMPOPS was implemented as a priority queue-based system in which queue "demons" serve requests generated by a central scheduler (see Figure 5.1).[1] One processor is dedicated to scheduling rules and managing working memory locks; all others are allocated to demon processes. Each demon process will execute requests at any level of parallelism, rule, match, or action. The basic functioning of the rule execution cycle is described below.

As each rule instantiation becomes enabled, it is placed in a queue of eligible rules (the eligibility set). Prior to being placed on the queue, each rule instance is optionally rated according to a rule-specific (and possibly situation-specific) function defined by the programmer. The scheduling process takes each rule instance off the queue in turn and attempts to acquire locks associated with the working memory elements positively referenced or modified by the instance in accordance with the scheme described in Chapter 3. If the locks can be acquired, the rule is scheduled using a combination of its rating or a rule-specific priority assigned by the programmer at compile-time. The rule-specific priority is used to determine the execution queue in which the rule should be placed. The "rule demons" examine the execution queues in a user-specifiable order so that it is possible to define certain types of rules to be more important than others. Generally, rules placed in a lower numbered queue will be executed with higher priority, however the user is given the capability to modify the order in which queues are examined by the rule demons. To allow heuristic discriminations to be performed between rules, each individual execution queue may be declared to be a priority queue so that rules can be prioritized according to their ratings.

The rule instances are removed from the execution queues by *rule demons* which then proceed to execute the rules. If heuristic pruning is enabled, the rule demons will first execute a control function attached to the rule type in order to determine whether the rule instance should be pruned. This rule pruning is necessary to maintain the responsiveness of the system, because the state of the system may change dramatically between the time when a rule becomes eligible to fire and when a processor becomes free to execute it. This is particularly true if rules are prioritized – less important rules may remain on the execution queues for significant lengths of time; long enough to become redundant or unnecessary to the computation.

The rule-demon approach was adopted over a previous approach of forking off rule executions using the thread construct when it became clear that the thread mechanism did not allow sufficient flexibility in ordering and pruning rule executions

[1]The queue-based server architecture was inspired in part by the method in which match-level parallelism is implemented in CParaOPS5 [Kalp *et al.*, 1988].

in response to changes within the system. The rule demon system also makes it possible to instrument and measure the behavior of the scheduling queues.

5.1.1 Rule Demons

Although the queue servers are called "rule demons", this is actually a misnomer dating back to previous versions of UMPOPS. The demons are responsible for executing action and match-level parallel activities as well as rules. The control structure of the rule demons is structured according to the model that low-level activities should be served before higher granularity activities. This model is based on the observation that delaying a rule execution in favor of a match activity will not seriously degrade the performance of the rule, while delaying a match (or action) activity in favor of a rule may incur a performance penalty of several times the activity's lifetime. In accordance with this model, the central loop of each demon operates in the following steps:

1. While any match level operations are on the match queues, remove one match operation and process it.

2. If no match level operation is enqueued, while any action level operations are on the action queues, remove and process one action.

3. While no match or action requests are extant, monitor the rule queues, beginning from the lowest numbered (highest priority) and scanning towards the highest (lowest priority). If a rule is found, it is removed from the queue and executed.

4. Go to step 1 and repeat.

The above algorithm is greedy in respect to match and action parallelism; rules may not get sufficient resources to execute if many low-level activities are active. There is also the slight possibility of deadlock if many rules invoke synchronous match or action-parallelism activities simultaneously and then "spin", waiting for completion. The problem of deadlock is avoided by modifying the synchronizing operators in the rules' righthand sides to check for match or action activities that can be executed instead of wasting cycles spinning. This change increases processor utility at the cost of potentially delaying the completion of the rules when synchronization occurs (because the monitoring process may be executing an unrelated match or action-level task). The problem of rule delay due to processor saturation can be avoided by configuring a proportion of the rule demons to give higher priority to rules rather than action or match-level activities. This will allow high priority rules to be executed more expeditiously at the cost of prolonging the lower-level parallel activities.

The dynamic behavior of the system can be modified by changing the order in which rule queues are visited by the queue demons; UMPOPS allows this to

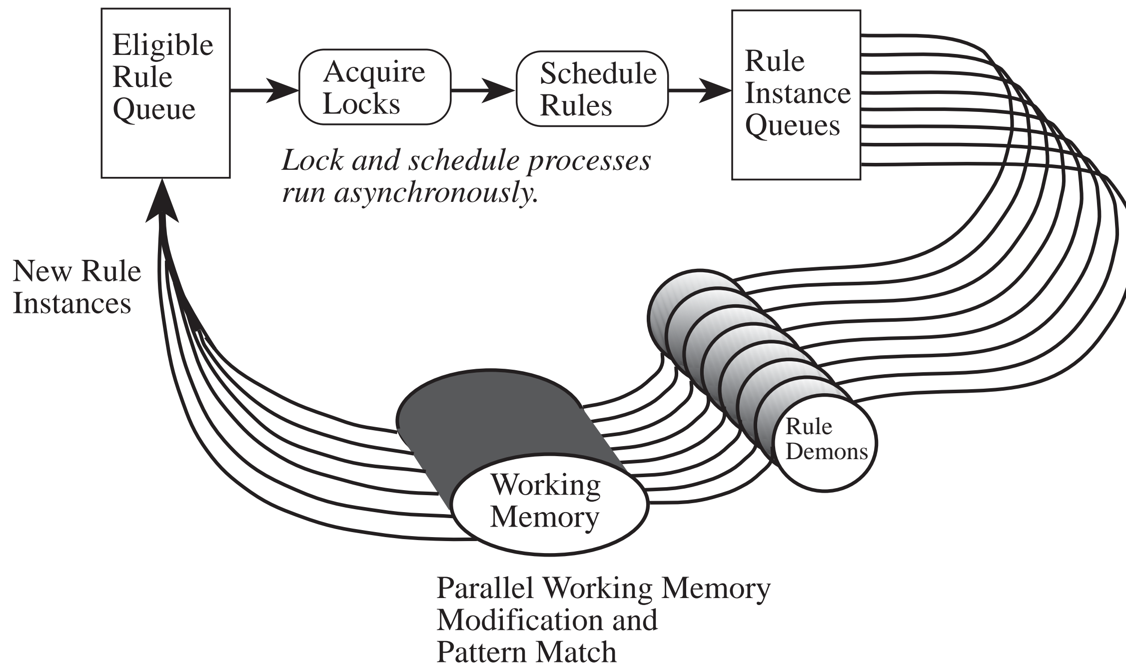


Figure 5.1: The architecture of the parallel rule-firing system.

be specified at invocation time. Because there may be increased contention for queues at low levels of granularity, UMPOPS by default provides two queues each for action or match level requests; this number may be increased by the user if monitoring indicates that contention for the scheduling queues has become a bottleneck. Measurement of delays in accessing rule-execution queues revealed surprisingly low levels of contention (even for priority queues with their increased insertion time) and only a single execution queue for each priority is usually necessary.

Experiments were performed with variations of the UMPOPS architecture, including multiple lock managers and varying numbers of execution queues. It appears that the limiting factor on performance is the amount of time that rules spend in the execution queues; performance seems fairly insensitive to scheduling times. This was in accordance with expectations; the 16 processor Sequent upon which the experiments were performed did not have enough processors to make the scheduling process a bottleneck.

Run Time Statistics: UMPOPS was implemented to serve as an experimental tool for exploring the characteristics of parallel rule-based systems and the tradeoffs imposed by various architectural choices. The rule-firing architecture is therefore instrumented to return timing information, both for the overall statistics of each run and individual statistics relating to each rule firing.

Execution Statistics The following statistics are gathered each time the inference engine is invoked.

- Total run time: The total time taken to run a program.
- Individual processor times: The total time spent by each processor in executing rules, as well as average, minimum and maximum times spent by each processor in executing rules.
- The total number of rules scheduled, executed, locked out, or deleted by heuristic functions.

Rule Execution Statistics: For each *rule*, the system keeps a record of the number of executions and the averages, over all the rule's instantiations, of the time to execute the righthand side, the time to "pre-evaluate" the rule, the time spent in the "conflict set", the time spent waiting in the execution queue, and the number of attempts and the time required to acquire working memory locks. For each of these statistics, the system records average, minimum, and maximum figures as well as standard deviations. Because rule instantiations are stored after execution, these statistics can also be retrieved for specific rule instances.

Queue statistics: For each scheduling queue, the system records the time at which each addition or deletion takes place; this allows the size of each of the priority queues to be charted over the course of a run.

Critical Regions: Mechanisms are available to measure the amount of time which is spent waiting to acquire locks on each critical region, however it was found that the presence of these mechanisms affected the timing of the system, and they are only employed when it is suspected that there is serious contention for a resource.

Working Memory Elements: For each working memory element, the system records the time of creation and the time required to perform the match for that element. (A deficiency in the recording of statistics is that there is no current method for recording the match time required to delete a working memory element.)

5.2 Modifications to LHS Syntax in UMPOPS

This section discusses the modifications made to the OPS5 language to support parallel activities and allow the elimination of sequential programming idioms.

5.2.1 LHS Meta-level Notation

The lefthand syntax of the rules in UMass parallel OPS5 has been modified to allow the specification of "meta-level" information. The notation has the syntax

```
(meta (meta-type value) (meta-type value) ...).
```

Although the `meta` construct is present in the lefthand side, it does not generate patterns or change the match in any way; it is simply used to send messages to the rule compilation routines. Originally, the purpose of the `meta` notation was to allow

the specification of control information, however, in actual use, the **meta** notation has been used as a catch-all for any language modification which would otherwise require a modification to the original OPS5 syntax. The current usages of the meta notation are summarized below.

5.2.2 Annotating Mode-changing Productions

Rule-based programs are usually organized in phases. Each production contains a reference to particular working memory element of a class such as *mode* or *stage*. The production is only enabled when the mode is set to a particular value. In order to change the mode, special rules such as the one below are used.

```
(p go-to-next-phase
        (stage ^is current-phase)
       -->
        (modify 1  ^is next-phase))
```

In a serial OPS5 system, the standard conflict resolution strategy is used to ensure that the mode-changing production only fires after all other eligible productions have done so. In a system which fires all eligible productions in parallel, the mode-changing production may execute prematurely. To prevent this, mode-changing productions should be explicitly annotated in the following way:

```
(p go-to-next-phase
        (meta
           (rtype mode-changer))
        (stage ^is current-phase)
       -->
        (modify 1  ^is next-phase))
```

"Mode-changing" rules annotated in this manner are handled specially by the scheduler, they are prevented from firing until all other rules in the conflict set have fired and all working memory changes have been processed. This guarantees that control rules will never conflict with or disable domain rules.

5.2.3 Other Uses of the `Meta` *Notation*

The `meta` notation has been used as a general purpose mechanism to specify information about the rules that would otherwise require changes to the existing OPS5 rule syntax. These usages are summarized below:

- **Priority:** A number between 0 and $N-1$ where N is the number of priority queues. All rules of this type will be placed within that queue.

- **Priority-queue:** If this flag is set to be non-nil, then rule instantiations of this type are placed on priority queue indicated by the priority keyword.

- **Priority-fn:** A function that is executed during the pre-eval phase in order to determine the rating of rules placed in a priority queue.

- **Lock-not-required:** If it is certain that a rule will never interact with other rules (e.g., it works in its own space), then locking is not required. Setting `lock-not-required` to be non-nil informs the controller that locks need not be acquired for this rule.

- **Control-fn:** A control function that, when executed in the environment of the righthand side, will determine whether or not the rule should fire. Variables bound by the rule can be accessed using the **`$varbind`** function of OPS5 (i.e. **`($varbind '<foo>)`** returns the current binding of **`<foo>`**). Usually the control function compares some combination of instantiation variables with a global variable describing the current state of the solution.

- **Control-generator:** A function that allows a control value to be associated with a keyword at instantiation time that is then stored in the rule instantiation for later reference by control functions. The user specifies a form, i.e. `((gen-control-data indicator exp))`, and the cons-cell `(indicator . exp)` is placed on an association list associated with the rule instance. It can then be accessed by various control functions using a standard `assoc` call, i.e. `(assoc '*tsp-distance* (rule-instance-control-data instance))`.

- **Rhs-kill-actions:** A list of righthand side actions to be taken if the rule is killed by a control rule. These are usually "clean up" actions that delete the current state so as to reduce the overall size of working memory.

Examples of the use of the `meta` construct can be seen in the programming examples in Section 6 and the appendices.

5.3 New Righthand Side Functions in UMPOPS

This section describes the additions to the OPS5 righthand side instruction set and syntax.

5.3.1 Invoking Action and Match-level Parallelism in the RHS

In a previous version of UMass parallel OPS5, action and match level parallelism were enabled by global flags. Using this technique, it was not possible to selectively employ these levels of parallelism in specific rules. Because action and match parallelism are not appropriate in all situations and may cause saturation of processing capability, it proved necessary to develop RHS language constructs to allow action or

match parallelism to be specified at the level of individual working memory changes. To avoid having to maintain separate versions of programs with and without action and match level parallelism, the global switches `*node-parallelism*` and `*action-parallelism*` are retained. If these flags are set to `nil`, parallelism is disabled and all actions and match activities will take place serially.

A disadvantage of using explicit constructs to implement low-level parallelism is that it is not possible to simply activate these levels of parallelism to measure the speedup achievable on existing OPS5 programs using only action or match-level parallelism; the programs must be modified to explicitly invoke parallelism in their righthand sides.

Currently, UMPOPS does not support the nested use of both match and action level parallelism constructs because any use of match parallelism tends to saturate the available number of processors.

5.3.1.1 Action Parallelism

There are two constructs, `in-parallel` and `in-parallel-sync` that invoke action parallelism. Each of these constructs takes one or more RHS working memory modifications or assertions as arguments. Working memory changes carried out within the scope of these commands are executed concurrently with all other working memory changes. If the `in-parallel` construct is used, the flow of control continues on to the actions following the construct, if any, as soon as the working memory changes are initiated. It is frequently necessary to ensure that all working memory changes have completed before a rule terminates, for example, when performing an initialization routine. In such cases, the `in-parallel-sync` construct is used; this

construct initiates all the RHS actions included within its body, then waits until they have all been completed before any further actions are taken.

5.3.1.2 Match-Level Parallelism

Match-level parallelism does not normally yield great speedups because of the small granularity of the match operations, the relatively high overhead of invoking parallel operations at that level of granularity, and the small number of rules affected by the average working memory change [Gupta, 1987]. There are, however, certain cases in which a significant improvement can be achieved. The most common of these is the mode-changing production. When a working memory element that triggers a new phase of the computation is added, unusually large amounts of matching activity occur and many rules are triggered. Under these circumstances, match-level parallelism can greatly reduce the rule execution time.

To invoke match-level parallelism, new righthand side actions are provided. These are **`make-match-parallel`**, **`modify-match-parallel`**, and **`remove-match-parallel`**. The syntax of these actions is identical to their serial counterparts, however, the matching of the working memory changes triggered by these commands will take place in parallel. The commands do not terminate until all the matching processes have been completed.

5.3.2 Make-unique

UMPOPS provides working memory locks to enforce consistency in working memory during concurrent activities. But the working memory locking scheme is not adequate to ensure correct rule firings for rules that contain negated elements

on their lefthand sides as it is not possible to acquire a lock on an element that does not yet exist. However, it is possible, through the **`make-unique`** function, to require that a rule "ask permission" before creating an element that it (or another rule) references through a negative condition element.

The **`make-unique`** function (actually a compiler macro) is used to create a working memory element with certain values once and only once. This mechanism, very useful when performing initializations, allows the user to specify a working memory element of a specific class with given key values. Before creation, a check is performed to see if such an element previously exists. If so, the rule is not allowed to execute. Otherwise the rule is allowed to create that element, and all other instantiations are prevented from doing so. This is called acquiring a "unique lock". An extended example of the use of the **`make-unique`** mechanism is given in Section 6.2.3.

The element that is to be created must be *declared* to be unique using the **`unique-attribute`** command. This command, which must precede any rule definitions, usually immediately follows the **`literalize`** command defining the working memory element. For example, to define a unique solution element for each of a number of tasks, one could use the following syntax:

```
(literalize task-solution task value)
(unique-attribute task-solution task)
```

After this declaration, only one element with the class 'task-solution' and a given value of the task field can be created by a call to **`make-unique`**; thus each task will have a unique solution element and clashing cannot occur. Uniqueness is only

guaranteed if the element is created using the `make-unique` call; UMPOPS does not prevent the programmer from later changing the key fields or values of a unique element. The syntax of the `make-unique` call (and, in fact, the function itself) is identical to that of the OPS5 `make`. The elements required to be created uniquely are annotated at compile time and the `make-unique` call is retained as syntactic sugar to highlight the elements that are intended to be created as unique elements.

Once obtained, unique locks are never released; this allows the programmer to implement "one-shot" rules that fire once and once only. The function `clear-unique-trees` must be executed between runs of a program in order to release all existing unique locks. (The underlying mechanism behind the unique locks is a discrimination tree upon whose leaf nodes the locks are hung.)

The `make-unique` function differs from the more general region locks in that it allows the user to specify a particular create operation to be singled out for special attention. Instead of having to check all new working memory elements against all currently defined regions, only elements declared to be unique are examined; this reduces the overhead of the locking mechanism significantly.

5.3.3 Control Task Syntax

A high-level mechanism for defining multiple independently executing tasks, each containing its own rule-firing policies and conflict resolution routines was described in Section 4.3. The commands for defining and invoking tasks are described in this section.

Defining tasks: Tasks must be defined before they can be created. A task definition consists of a task name, a rule firing policy (asynchronous or synchronous)

and, if synchronous, a conflict resolution routine specified as a function call taking an eligibility set.

```
(deftask <task-name>
         :conflict-resolution-routine <CR-function-name>
         :type {synchronous | asynchronous}
         )
```

Initiating tasks: A task must be explicitly invoked with one exception; all initial rule firings take place within the scope of an initial default task. (The user may define this default task to be asynchronous or synchronous.) The control task syntax is similar to that used in UMPOPS for specifying action-level parallelism:

```
(with-new-task(<task-name> body))
```

All RHS actions contained within the body are executed within the context of the task. Otherwise, all RHS actions within a rule are executed within the context of the task which stimulated that rule.

Modifying or Terminating Tasks: When phases of a computation change, the nature of the task may have to change as well. The **(redefine-task <task-type>)** operator allows rules to change the rule-firing policy and conflict resolution routine of a task in mid-computation. The **(kill-task)** function allows a rule to terminate the task context in which it is executing. This allows the resources used by the task, specifically the eligibility set, to be reclaimed and assigned to other tasks.

5.3.4 Set Functions and Synchronization Groups

The use of set-oriented productions as a method for eliminating sequential serial rule firings was discussed in Section 4.4.1. UMPOPS supports a simple implementation of set-oriented productions loosely based on that described in [Gordin and Pasik, 1991]. The synchronization group operators that are required to avoid premature firings of set rules are described in the following section as an example of primitive functions for performing local synchronization tasks.

5.3.4.1 Syntax of the Set Notation

The addition of set-oriented rules to OPS5 required changes to both the left and righthand side rule syntax. Set-oriented rules must have one or more *set patterns* in their lefthand side. A set pattern is a condition element surrounded by square brackets, e.g. `[(block ^color <x>)]`. Any such condition element matches *all* elements that satisfy the pattern. If more than one set pattern is present in the LHS, then one instance of a set production matching the cross product of all these elements will be produced. One can think of a set rule as simply compressing all the rule firings that would occur in standard OPS5 into a single rule firing. The **map-set** function allows the programmer to map RHS operations across the set of rule instantiations.

```
(map-set
    (rhs-action)
    (rhs-action)
       :     :
    (rhs-action))
```

The righthand side actions that are encased in the `map-set` function are mapped across the set of instances. Any variables or condition element variables are bound to the appropriate values in turn. Any righthand side actions that should only be executed once can either precede or follow the `map-set` construct. The set notation can also be used to count occurrences of working memory elements; when a set-oriented rule is executed, the variable `<set-count>` is bound to the number of instances contained in the set.

5.3.4.2 Synchronization Groups

The following righthand side actions are used to implement *synchronization groups*:

Generate-sync-group: Returns a pointer to a synchronization group. If invoked within a righthand side, all working memory elements subsequently created by this rule will be considered as members of that synchronization group.

End-group: Terminates a synchronization group. Any rules enabled by the working memory elements within the group will become enabled as soon as all elements have completed matching.

Signal-quiescence-to-group: Used by the working memory match routines to signal to a group that a working memory element has become quiescent. This simply decrements a counter within a critical region and then, if the counter becomes zero and the group has been terminated, allows enabled rules to fire.

Signal-wme-active-to-group: Adds a new working memory element to the synchronization group.

Set-group-completion-demon: Allows a function to be attached to a group to be executed when the group terminates and becomes quiescent.

The synchronization group mechanism is sufficiently flexible to be used for purposes other than synchronizing set-rules, for example, it is used for avoiding race conditions when performing modify actions in parallel.

5.3.5 Map-vector

OPS5 is oriented towards using rules as the basic unit of iteration. This is particularly inefficient because each rule firing consumes an unavoidable amount of overhead in terms of matching, rule instantiation, conflict resolution (if any), and variable binding. The use of rules to implement trivial loops tends to eliminate the possibility for rule parallelism as the computation cannot proceed until the loop has terminated, and rules fire sequentially within a loop.

The `map-vector` command is an example of the kind of syntactic mechanism that can be incorporated into a rule-based language to allow iterative operations to be performed within the scope of a single rule firing. Much more sophisticated structures than vectors have been incorporated into rule-based languages since OPS5 was written (for example, OPS83 allows working memory fields to contain structured records) and the `map-vector` command should be considered significant not for its expressive power (which is simply making up for a language deficiency) but for the reduction in sequential rule firings that it allows.

The `map-vector` command is used to map RHS operations over elements in a vector. A vector is a field of a working memory element corresponding to a list of values; it can be thought of as a one-dimensional array of arbitrary length. The

vector is not a particularly flexible mechanism and the mechanisms for manipulating them are crude. Iterating over a vector typically involves maintaining a working memory element with a counter and a vector and using a succession of rule firings to perform operations on each item of the vector in turn. Each rule firing extracts a value from the vector using a **substr** command, then deletes the element and reasserts a modified version with an incremented counter. This is far from efficient, even in a serial system. The **map-vector** command allows a single rule to map operations over a vector. The **map-vector** command has the syntax:

```
(map-vector {<ce-variable> | <ce-index>} vector-name
            {<field-bindings> | nil}
            body)
```

The <**ce-variable**> or <**ce-index**> is simply a pointer to the element containing the vector. The **vector-name** is the name of the field in which the vector is stored. The <field-bindings> is a list of the form (<variable-name> key <variable-name> key) where *key* is one of prev | item | rest | index | length | vector-less-item. Body, of course, is the set of RHS actions to be executed in the context of the **map-vector**. The <field-bindings> need some explanation. During the execution of the **map-vector**, each element of the vector is considered in turn. Local variables are bound to the list of elements previously seen(prev), the list of elements yet to be seen (rest), the current element (item), the index of the current element(index), the length of the vector (index), or the entire vector except for the current item (vector-less-item). Because **map-vectors** can be nested, the programmer is given the ability to bind these values to appropriate variable names. The ability to refer to the previous and subsequent items in the vector allows permutations of vector

elements to be produced. A typical use of **map-vector** from a travelling salesperson example is shown below:

```
(p start-city
  {<start> (start ^start-city <sc> ^length <length> )   }
           (initialized ^value t)
   -->
  (remove 2)
  (map-vector <start> city-list (item <city> vector-less-item <vli>)
       (bind <tag>)
     (in-parallel
       (make connect-goal ^tag <tag> ^city1 <sc> ^city2 <city>
                          ^length (compute <length> - 1)
                          ^city-list  <vli>)
       (make so-far ^tag <tag> ^distance 0 ^cities-seen <sc>)))
   )
```

5.4 Multiple Worlds

One of the advantages of parallelism in complex system is the ability to explore multiple alternatives simultaneously. If the search is partitioned appropriately, then problems of rules interacting no longer apply. So we can trade the cost of detecting syntactic rule interactions against the expense of creating partitioned (and possibly redundant) states. Copying states can be done informally in situations in which the entire problem-solving state can be represented as one or two working memory elements; in these cases, the partitioned state can be created by copying the elements in question and annotating them with a unique tag (see the Travelling Salesperson example in Chapter 6 for an example of this technique). For larger, more complex states, possibly consisting of linked data structures, the problem of accessing and copying states becomes more problematic. Not only is it difficult to explicitly

reference each appropriate working memory element in the lefthand side of the rule and explicitly copy it on the right, but the use of explicit tags to denote state places an undue strain on the pattern matcher and can lead to matching inefficiencies.

A simple approach to partitioning is to copy each new state into a logically separate version of the Rete net. That is, we can imagine each node of the Rete net being sliced into an infinite number of dimensions. Each dimension represents a new state, and as each new state is created, the working memory elements associated with that state are placed in the appropriate world, or dimension (see Figure 5.2). Thus, creation of a new state consists simply of the transformation $WME_i -> WME_i + 1$ where the arrow denotes a copying/transformation operator. The advantage of this approach is that working memory elements in one state can only be compared with working memory elements in the same state and therefore identical elements can appear in multiple states, thus no re-naming or tag generation is necessary. The principal disadvantage, and it is potentially a large one, is that the copying operation is likely to be very expensive (it effectively violates the *temporal redundancy* requirement of the OPS5 Rete net.) The response to this objection is two-fold; first, given sufficient resources to implement action parallelism, copying of working memory elements can be done concurrently and the creation of new states can be reduced to $O(n)$ where n is the depth of the network. The second response is that, with a sufficiently clever copying algorithm, one state can be copied directly into another from one slice of each node of the Rete net to the next, thus maintaining the partial match state contained in the Rete net. This copying operation is potentially suitable for large scale SIMD parallelism and could potentially be carried out in $O(1)$ time. A second disadvantage is that of space

usage – the copying scheme inevitably will result in redundant copies of working memory elements which could be inefficient to maintain and garbage collect. We can somewhat reduce this problem by maintaining a *base* space; that is, a space in which all facts not actually modified during the course of the computation are stored. The ultimate solution to the problem of space (and to a certain extent, that of copying overhead) is to employ a scheme in which successor states simply inherit the working memory elements from predecessor states via pointers augmented with truth maintenance-style IN/OUT lists.

An experimental version of UMPOPS has been developed with a partitioned Rete net and operators for performing parallel search in multiple worlds. This multiple worlds implementation is described more fully in [Neiman, 1992b].

5.5 Implementation of Parallel Matching in UMPOPS

This chapter discusses the data structures and algorithms that are used to implement the parallel OPS5 matching process and the changes that were necessary to allow parallel activity. During the reimplementation process, it was discovered that there are several assumptions concerning the order in which activities take place within the matcher that are no longer valid in a parallel system – the potential errors and their solutions are also described in this section.

Many of the details of the implementation were inspired by Gupta's study of the issues involved in parallelizing the Rete net [Gupta, 1987]. Because this work is undoubtedly familiar to the interested reader, this report concentrates primarily

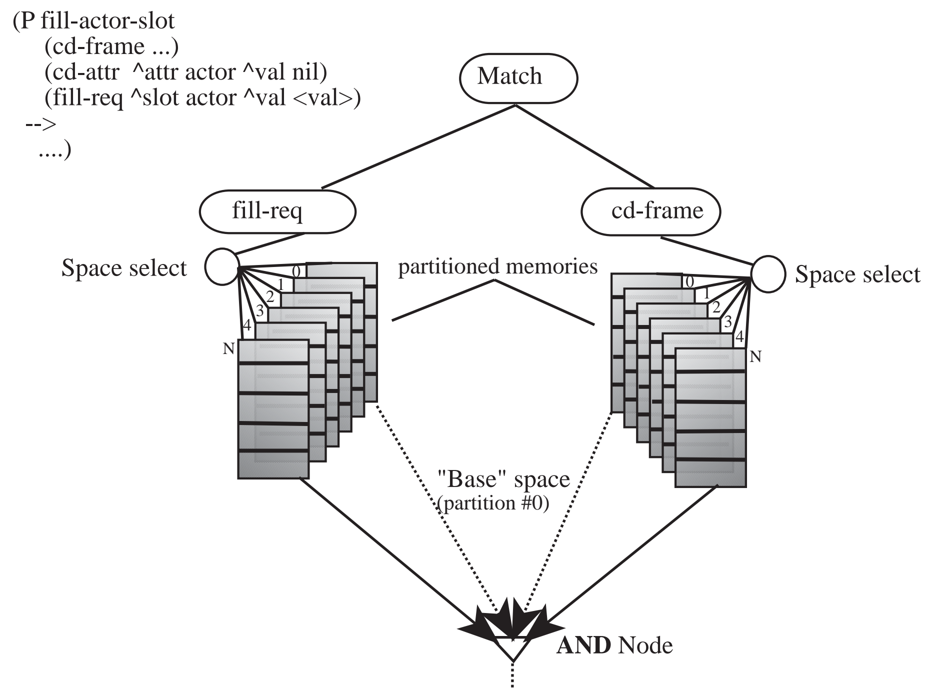


Figure 5.2: By partitioning the memories of the Rete net, a multiple world implementation suitable for parallel search can be transparently achieved. To minimize copying, a “base” space or partition can be defined that contains knowledge guaranteed to remain stable over the course of the search.

on the implementation details that are unique to parallel OPS5, particularly the synchronization of two-input nodes.

5.5.1 The Rete Net

Because the following discussion hinges on an understanding of the internals of the OPS5 pattern matching process, a short overview is given of the processing that takes place within the Rete net, the principle data structure in OPS5. In production systems, most of the processing time is spent determining which rules are eligible to fire. In OPS5, this process consists of matching the lefthand sides of productions against working memory. When a set of working memory elements is found such that there is a working memory element for every non-negated condition element in the lefthand side and there exist no elements that match negated condition elements, the rule is eligible to fire. As a principal bottleneck in rule firing, this matching process should be as fast as possible.

5.5.1.1 Rete Net Overview

The Rete-net matching process works by passing tokens consisting of one or more working memory elements through the net, performing tests on them at each node. The "top" of the Rete net is composed of *alpha* nodes that consist of simple tests on the class of the working memory element and specific fields. This part of the network possesses no memory and resembles a conventional discrimination net; tokens are passed to succeeding nodes in the network only if the tests at the current node succeed. Alpha tests are not very time-consuming and parallelizing their execution does not lead to large improvements in performance.

Beta tests are responsible for unifying variable values between two condition elements (inter-element tests). Each of the beta nodes has two inputs and two memories, one associated with each input. As a token arrives at a beta node, it is stored in memory and tested against the *opposite* memory to see if one or more consistent bindings can be achieved. If so, a new token is constructed from the incoming token and the stored token. This new token is then propagated through the beta node's out list (a list of successor nodes). The memories associated with the beta nodes store partial matches, making it unnecessary to repeat the entire computationally expensive unification process after each working memory modification. The cost of executing a beta node is proportional to the size of the memory against which the incoming token is tested. The two primary beta nodes are the AND and NOT nodes. Beta nodes present numerous opportunities for parallelism; for example, multiple beta nodes can be executed in parallel, or, if the architecture supports sufficiently fine-grained processing, an incoming token can be compared to each corresponding token in memory simultaneously. Beta nodes also present a number of obstacles to implementing parallelism. First, they contain memory nodes that must remain consistent despite possible parallel accesses. Secondly, each beta node refers to at least two tokens that can change asynchronously during the match process. Finally, new data may arrive during a match episode; synchronization constructs are needed to ensure that the new data does not stimulate spurious matches or none at all.

At the bottom of the Rete net is a series of *production* nodes; when a token arrives at one of these nodes, the production corresponding to the node is placed in the conflict set, instantiated with variable bindings from the incoming token. The

production node has no memory, thus only one production firing ever results from a given combination of working memory elements.

AND Nodes: The operation of an AND node is illustrated in Figure 5.3. In part A of the figure, a token is shown arriving at the memory node of the AND. The token (which represents a partial match) is inserted into the memory of the AND node and then processed (part B). (In a serial system, it does not matter whether the node is placed in memory before or after processing, although this is not the case when node parallelism is allowed.) Part C of the figure shows the processing of the AND node. The incoming token is compared to each token in the *opposite* memory according to the list of tests contained within the node. A typical test might compare the value of the third slot of the second element of the incoming token to the fifth slot of the first element of the memory token.

Pairs of tokens that satisfy the tests are concatenated into a new token and passed to the succeeding nodes in the network. Because an AND node is basically symmetrical, this description covers the case of tokens arriving from both the left and right sides. In the case of a *negated* token (that is, a token resulting from a `remove` working memory command), the AND node functions in the same way except that the token is removed from the memory node and, if the token satisfies the tests, a new *negated* token is passed to succeeding nodes so that partial matches will be deleted from memory nodes lower down in the network. If the succeeding node is a production node, then the negated token is used to remove the corresponding instantiation from the conflict set.[2]

[2] A negated token should not be confused with a negated condition element. A negated token is simply a token tagged for removal while a negated condition element specifies a working memory element that must not exist if a rule containing it is to fire.

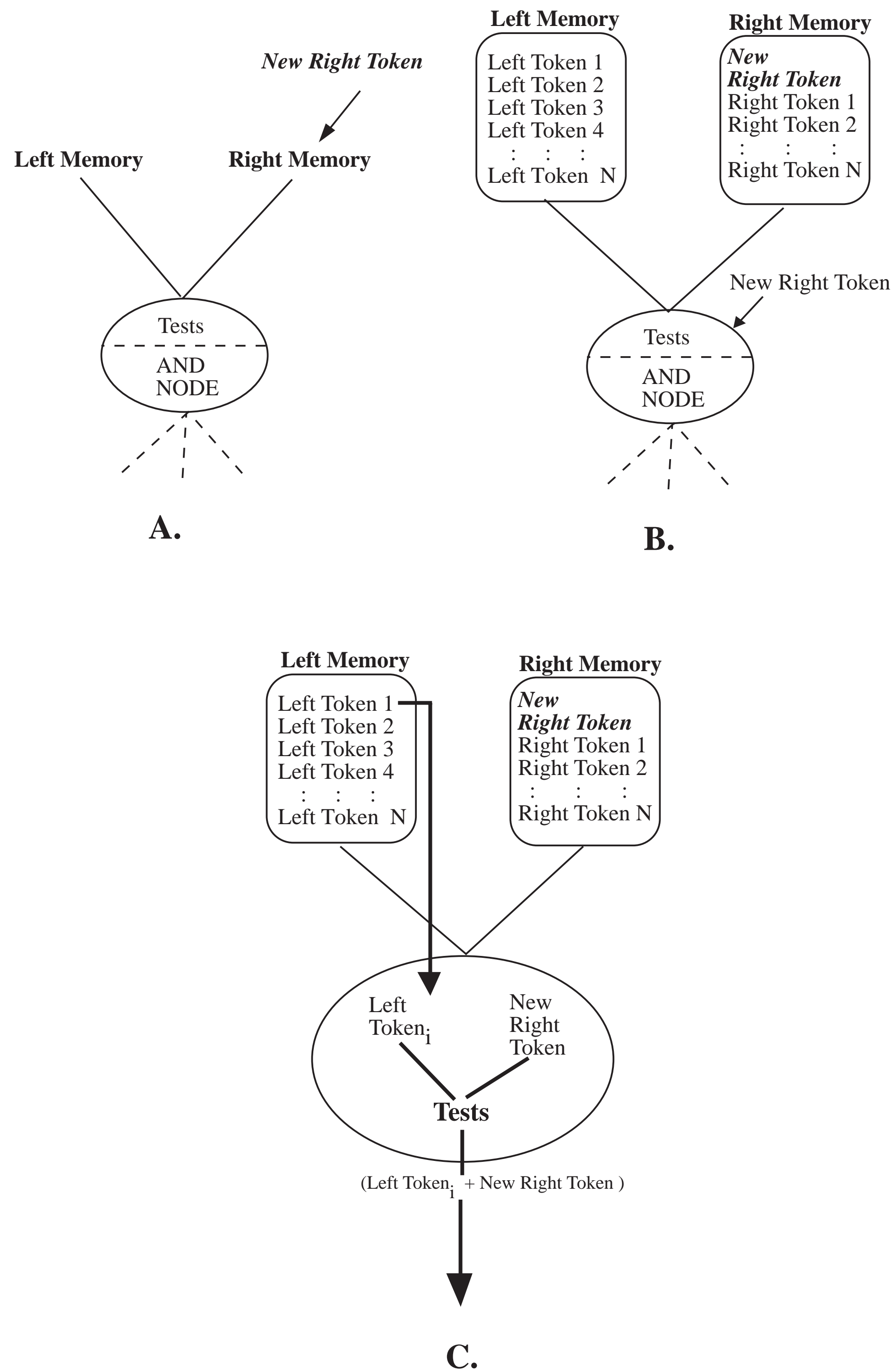


Figure 5.3: A token arrives at an AND node.

NOT Nodes: A NOT node is used to implement negated clauses in a production lefthand side. The NOT nodes are structurally similar to AND nodes, but the processing is quite different. A NOT node must ensure that for a given negated clause, there is no working memory element that matches that clause in such a way that there are consistent variable bindings with the working memory elements matching the preceding LHS clauses. Like the AND node, the NOT node has two memories. One memory is devoted to working memory elements that potentially match the negated clause. The other memory contains a list of tokens corresponding to the non-negated condition elements of the LHS and, associated with each token, a count of the number of matches that occur in the opposite memory. The processing of tokens arriving at a NOT node differs according to whether the token arrives from the left or righthand side.

A token arriving from the left (the choice of sides is arbitrary) represents a list of elements that match the lefthand side condition elements of the production; this token will be propagated through the net only if no token is present in the opposite memory that satisfies the tests of the NOT node. The arriving token is placed in the lefthand memory of the NOT node and assigned a counter value of zero (Figure 5.4, parts A and B). For each element in the right memory, the test is performed; if successful, the counter is incremented. If the count is zero after the entire righthand memory has been examined, the token is propagated.

A token arriving from the right (Figure 5.5, parts A and B) is placed in the righthand memory. Then, for every token in the lefthand memory, if the tests are satisfied, then the corresponding counter is incremented. If the counter was formerly 0, then the new token has disabled the production; in this case, the lefthand token is

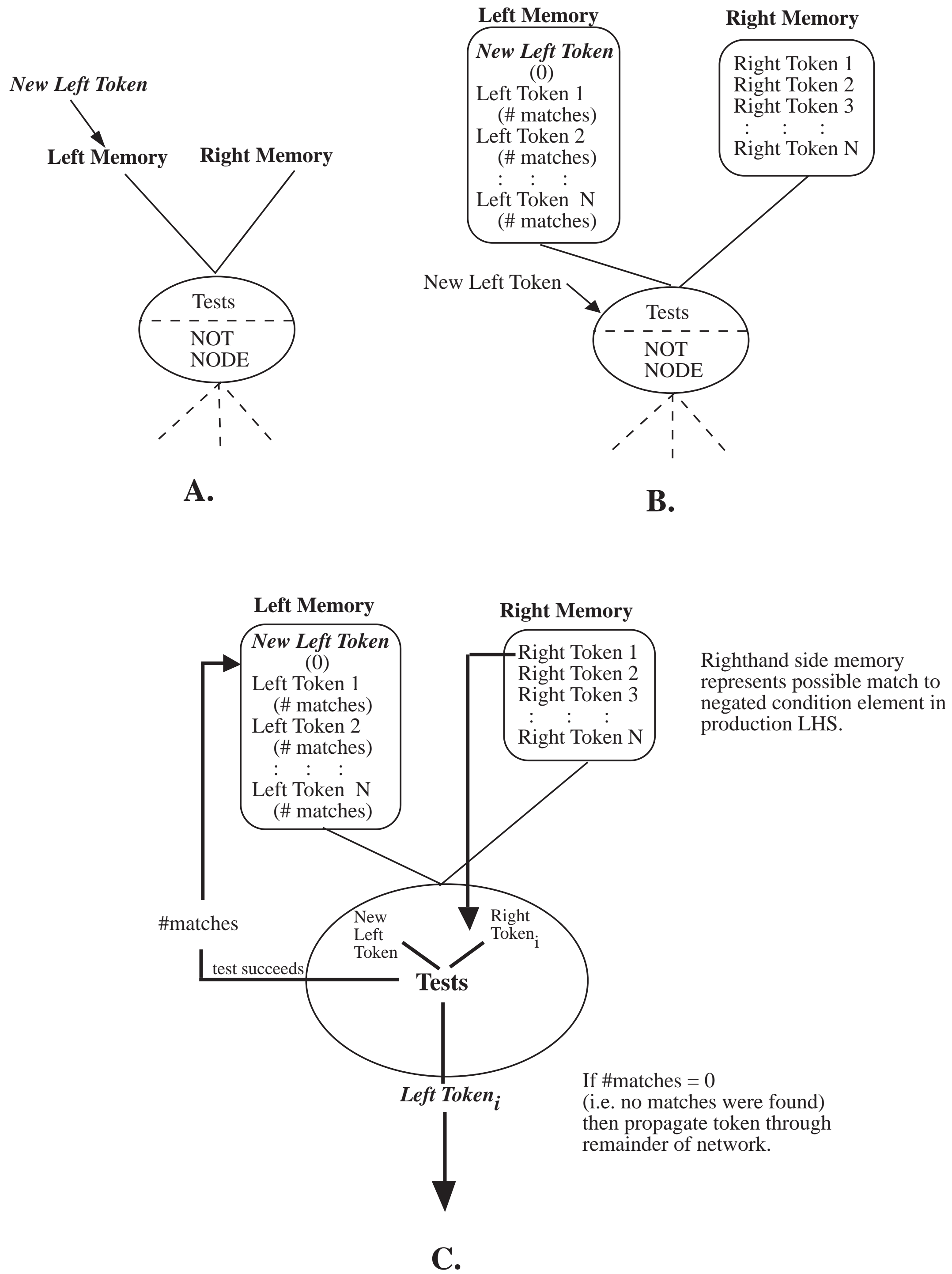


Figure 5.4: A token arrives at the lefthand input of a NOT node.

negated and propagated to remove it from memory nodes further down the net. For negated (deleted) tokens arriving at the NOT node, the process is the same except that the token is removed from the memory and the counters are decremented. If any counter becomes 0, then the lefthand token is propagated.

5.6 Implementing Match-level Parallelism

Any degree of parallelism in a system that uses a Rete net pattern matcher implies the presence of (or at least the capability for) node and intra-node parallelism. Technically, the node level of parallelism allows more than one test node in the network to be active at the same time while intra-node parallelism allows multiple activations of a single node. The first attribute increases the speed of a single match episode because multiple paths of the network can be traversed in parallel. The second attribute allows multiple match episodes to take place at once; a situation that arises when multiple actions in a RHS are executed concurrently, or when the righthand sides of multiple productions are executed concurrently.

Match-level parallelism occurs when node parallelism is used to increase the matching speed of a single working memory element change. Implementing match parallelism is relatively simple. Each node possesses an *out-list*, that is, a list of the nodes that succeed it in the network. In a serial system, this out-list is traversed in a depth-first fashion; to parallelize the net traversal, each item in the out-list is traversed in parallel. This approach to match parallelism spawns one new process for each node in the net traversed by a given token. Depending on the structure of the Rete net, the branching factor, the amount of computation performed at

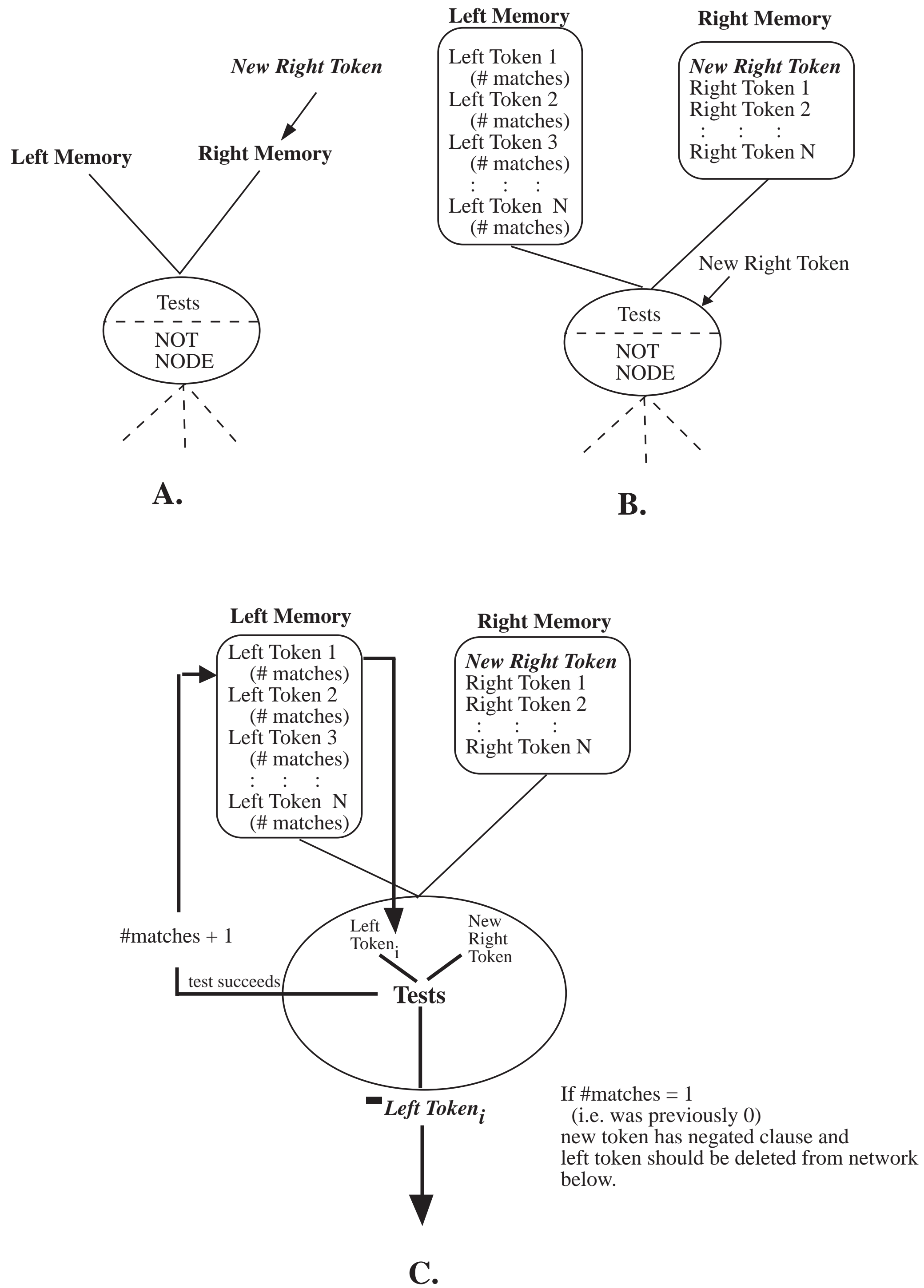


Figure 5.5: A token arrives at the righthand input of a NOT node.

each node, and the overhead of invoking parallel processes, this might involve more overhead than is gained by the parallelism. Variations on the scheme involve only invoking node parallelism when the out-list is large, not invoking node parallelism for the simple alpha nodes, only creating parallel processes at the first level of beta nodes, or creating less than N processes for an N-element out-list, with each process then traversing part of the out-list in a depth-first fashion.

When node parallelism is employed, there is a chance that multiple production nodes may be simultaneously active, causing multiple instantiations to be entered into the conflict set at the same time. For this reason, the conflict set (or, if the implementation does not require conflict resolution, the eligibility set) must be considered a critical resource, and the add and delete functions must take place within a critical region.

Intra-node parallelism, in which multiple tokens can be processed by multiple activations of the same node at the same time, is required for action- or rule-level parallelism. The major difficulty is maintaining the consistency of the associated memory (for beta nodes) during simultaneous accesses. If two tokens are added to memory at the same time, then the memory list could end in an inconsistent state. To avoid this problem, each memory node is assigned a unique lock that allows token insertion and deletion to be performed within a critical region. For AND nodes, the memories do not have to be locked during the actual token processing as the synchronization mechanism described in the following section ensures that the state of the network remains consistent.

5.7 Synchronization of 2-input Nodes

The Rete net makes the implicit assumption that only one token is processed by a two input node at one time. When the system supports action or production parallelism, this assumption is no longer true; multiple tokens might arrive at a two-input node at any time, and at either input [Forgy, 1979]. It is inevitable that eventually a token will arrive at either the left or right input while a matching token is still being processed on the opposite side. This can cause serious synchronization problems.

There are two possible failure modes, depending on when the token is added to the node's memory. Figure 5.6 depicts the case in which the implementation adds tokens to the memory *before* passing the token to the AND node. When tokens arrive simultaneously, it is possible that the left token will match against the right token, and the right token will match against the left token. This will result in two identical tokens being propagated through the network. The inevitable result is that the conflict set will eventually contain multiple identical instantiations, multiple copies of tokens will proliferate in memory, and the state of the network will be corrupted.

If the tokens are added to the node's memory *after* the matching process takes place, then it is possible that neither token will match. The righthand matching process will examine the lefthand memory and not find a matching token and the lefthand process will examine the righthand memory and not find a matching token. Then both tokens will be added to memory on their respective sides. This would

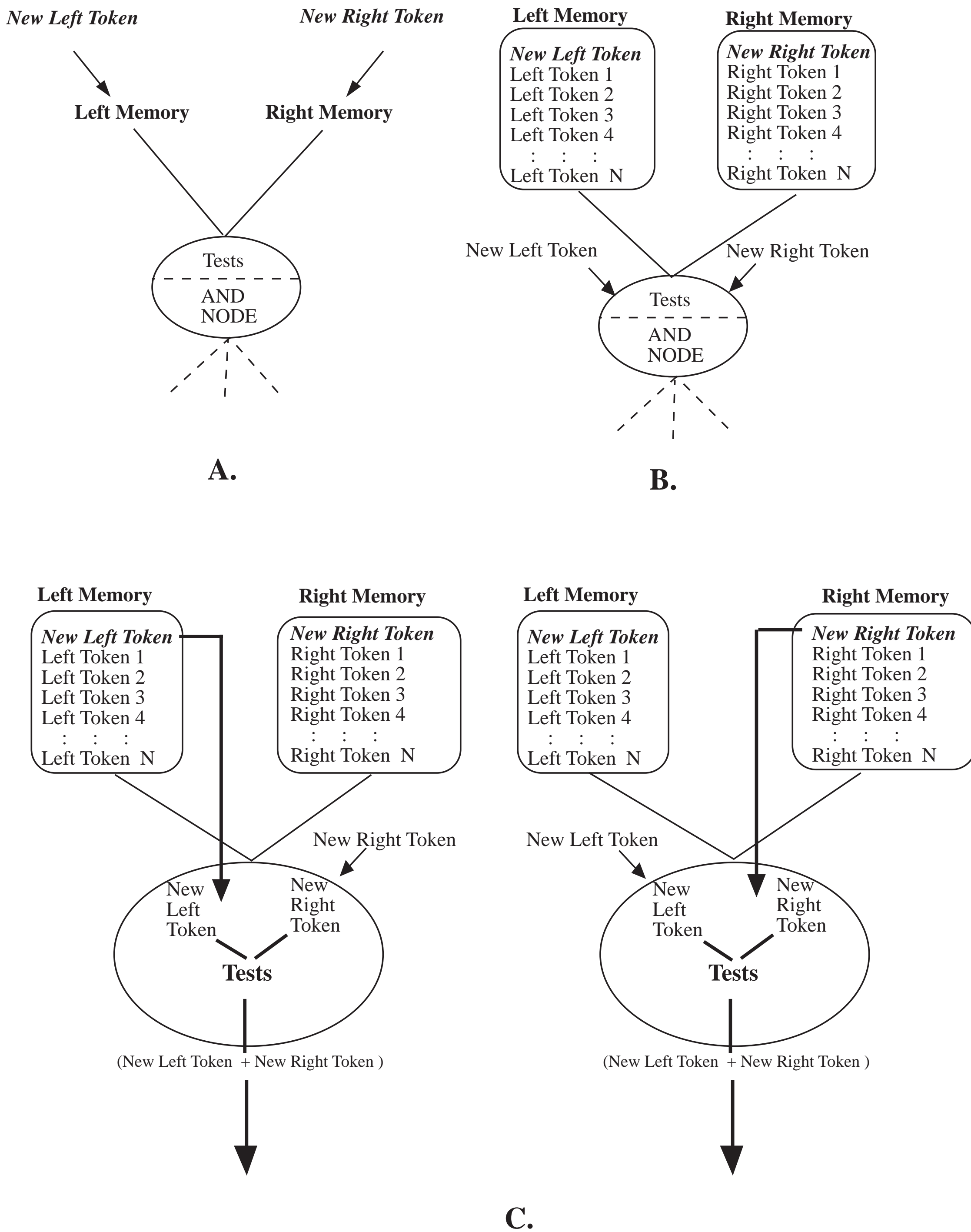


Figure 5.6: When matching tokens arrive at an AND node simultaneously, synchronization errors can occur.

result in a situation in which the node's memories contain two tokens satisfying all tests but which have not been passed further down the net.

One possible solution to this problem, which was adopted by Gupta, is to *lock* one side of a node when a token arrives from the opposite side so that the problem of simultaneous arrival never occurs. This is unduly restrictive, however, for the occurrence of a simultaneous arrival of matching tokens is very rare. Non-matching tokens arriving at the opposite inputs can be processed without difficulty and a locking approach would unduly reduce throughput through the node. UMPOPS adopts the following solution in lieu of locking memory nodes.

In UMPOPS, tokens are added to memory before they are passed to the AND node, so the case in which two identical tokens are propagated must be detected. The solution taken to this synchronization problem was to add a *completion flag* and a *match list* field to each token being passed through the network. As each token enters a two-input node, the flag is set to false. It is not set to true until all tests have been completed on that token and it has been added to memory; obviously, if a node's match flag is set, then its matching process is complete and it is not going to generate any more matches unless a matching token arrives on the opposite side.

When a test in the two-input node succeeds, the matching token is checked to see if its completion flag is set. If the flag is set, then the token is propagated as usual. If not, then that token is currently being processed and a case of simultaneous activation exists. The matching token is stored on the incoming token's match list. After the incoming token has been compared against the entire opposite memory, both it and its match list are passed to a synchronization process. The synchronization process iterates over the match list examining the completion flags

of each item. If the flag becomes set, then the two tokens are concatenated and propagated. If the matched token's flag is not set, then its match list is examined to see if it contains the current token. If so, then the two nodes are mutually matching and a synchronization error exists. In this case, the result of one match is suppressed by removing it from the token's match list (The choice of which side the token is removed from is arbitrary). Once the matching token is removed, the match list for the incoming token becomes empty and its completion flag is set. This allows the opposite synchronization process to propagate the concatenated token further down the net.

The overhead for this synchronization check is not high because it is rare for two tokens to arrive at a node simultaneously, therefore the usual overhead is the creation of the token data structure, and the checking and setting of the completion flags. Figure 5.7 demonstrates the synchronization process.

To prove that this mechanism correctly solves the simultaneous token problem, consider the following cases.

Case A: The left token(T_L) arrives and completes matching before the right token T_R arrives. This is the same as the serial case. The left token does not find a righthand match and is not propagated, but sets its completion flag. The righthand token then arrives, successfully matches against the lefthand token, and, because the completion flag for T_L is set, the result is propagated.

Case B: T_L does not complete matching before T_R arrives, but, because T_R is concatenated to the front of the memory list, T_L does not match against T_R. In this case, the completion flag for T_L is not yet set, so T_L is placed on T_R's match list. The synchronization mechanism ensures that T_R cannot complete until the match list is

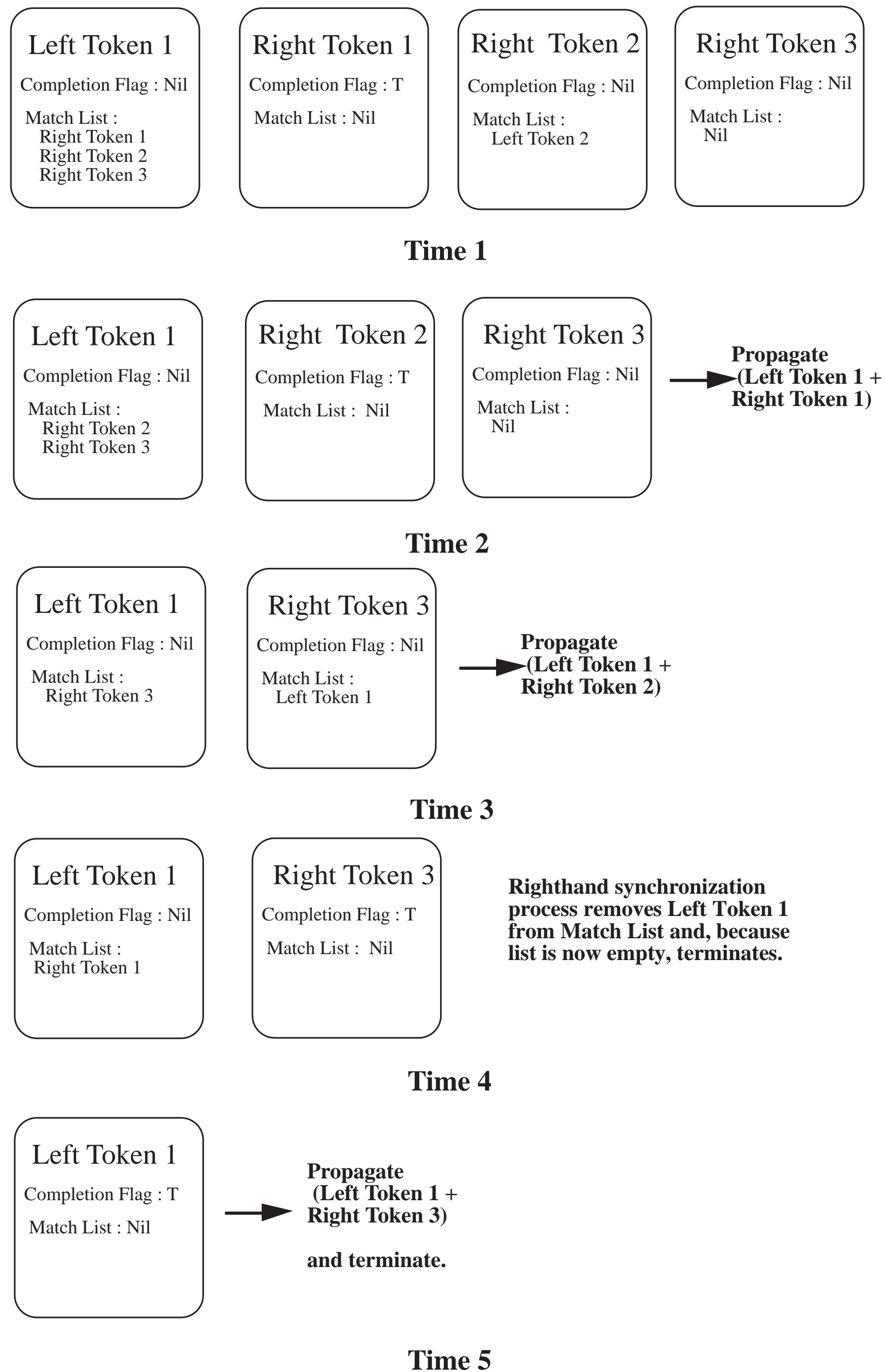


Figure 5.7: The synchronization process for an AND node.

empty. The token T_L, however, has an empty match list and completes, setting its completion flag. T_R can then propagate the result of concatenating T_L and T_R.

Case C: T_L and T_R arrive at the two input node simultaneously. Both are entered in to memory, and each successfully matches against the other. Left uncorrected, two identical tokens ($T_L + T_R$) would be propagated through the network. This is the pathological case which the synchronization mechanism was designed to avoid. The completion flag can not be set on either token because in order to do so, the opposing token would have to have its flag set. Therefore, the matching token is placed on each incoming token's match list, that is, T_R stores T_L and T_L stores T_R on its list. Each token is then passed to the synchronization routine. The synchronization routine observes that the token T_R has T_L on its match list which in turn has T_R on its match list. It arbitrarily deletes T_R from T_L's match list. T_L then has a null match list and the match process terminates, setting the completion flag for T_L. Once the flag is set, the synchronization process monitoring T_R can then propagate T_L+T_R and remove T_L from T_R's match list, allowing its match process to terminate. Only one copy of the outgoing token is propagated.

Because of the symmetry of the two-input AND node, the tokens T_L and T_R can be reversed in the above discussion. There are no other cases. It remains only to consider the case of deadlock. Is it possible for a token to never set its completion flag, thus resulting in a synchronization process that never terminates? The answer, briefly, is no, for the only way for a token to never complete is for it to match a token whose completion flag is never set. But the only way for this to happen is for the two tokens to be mutually matching, and this deadlock is arbitrarily broken by the synchronization routine.

The synchronization problem may also appear in NOT nodes, however this was solved by another mechanism. Because the NOT node modifies its memory nodes during processing (by incrementing counters), it proved easier to simply lock both memories while the node was being executed. Therefore, the simultaneous synchronization problem does not arise in this implementation. However, locking the memory nodes dramatically reduces throughput, and a less restrictive algorithm should eventually be developed.

5.7.1 Synchronization and Sharing of Memory Nodes

The approach taken to synchronization effectively prevents the sharing of memory nodes in UMPOPS. If a memory node has an *out-list* of more than one beta node, then a token's synchronization flag might be set in any of these nodes. This could cause synchronization of any of the other sibling nodes to take place improperly. Because the proliferation of memory nodes is inefficient in terms of space usage and increases the number of critical regions that must be acquired during processing, this restriction should be removed. Methods such as arrays of synchronization flags in which the arity of the array is the same as the arity of the *out-list* should be applicable, but have not yet been implemented.

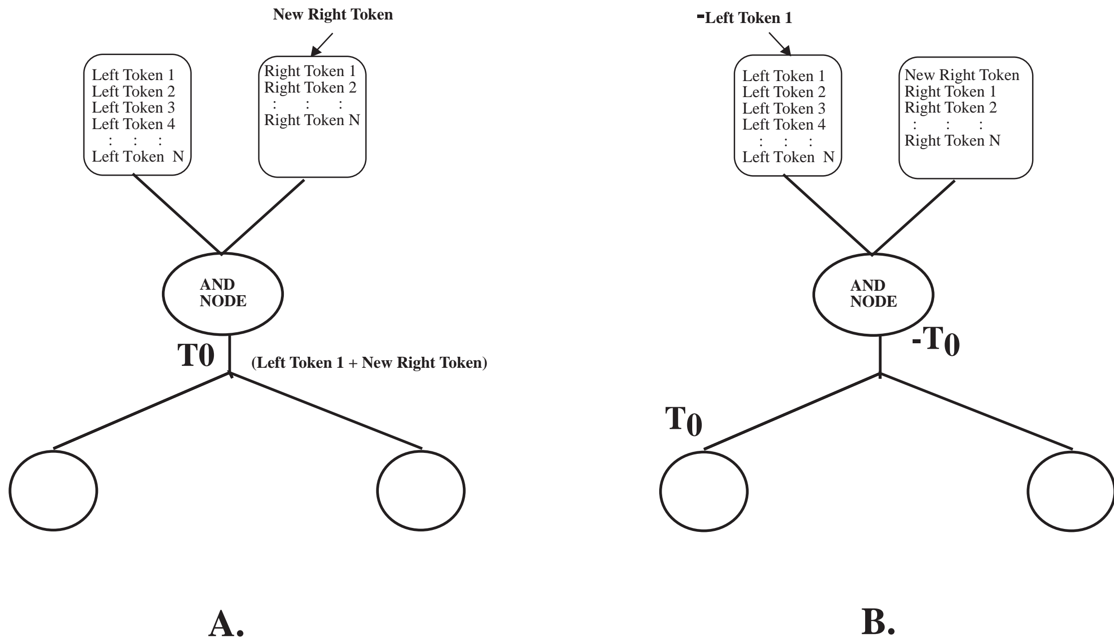
5.8 Race Conditions

There is one additional hazard due to intra-node parallelism that must be guarded against. Consider the case in which a token T_1 enters a two input node and matches with a token contained on the opposite side T_2. A new token consisting of the two tokens concatenated together, $(T_1 + T_2)$ is propagated through the tree.

Now suppose that a remove working memory element episode takes place, that causes T_2 to be removed from the node memory. This causes the token -(T_1 + T_2) to be propagated, where the minus sign represents a flag specifying deletion. For any number of reasons, it is possible that this negated token could arrive at a beta node or production node before the original token. If this happens, the deletion will fail, and the positive token will remain in memory despite the fact that one of its supporting working memory elements has vanished (Figure 5.8). Currently, race conditions are avoided by using the task synchronization mechanisms provided by UMPOPS; no action stimulated by an embedded add or delete operation in a `modify` command is allowed to execute until the opposing match operation has completed.

5.8.1 Avoiding Critical Regions

When running in parallel, it is necessary to reduce the time that processes spend in critical regions as this tends to serialize performance. The most notable critical regions in UMPOPS are the eligibility (conflict) set and the node memories. Conflict for the node memories is greatly reduced by hashing. Inserting instantiations into the eligibility set is inexpensive, however deletion is costly because the traversal of the list and the actual deletion must take place within a lock. To avoid this delay, instantiations that are deleted from the eligibility set are simply marked as "killed", but are not actually removed from the list. As the scheduling process removes instantiations from the list, it checks to see if the instantiations have been killed; if so, they are simply discarded. A variation of this scheme could be used to reduce the overhead of deleting tokens from node memories, however this would require a



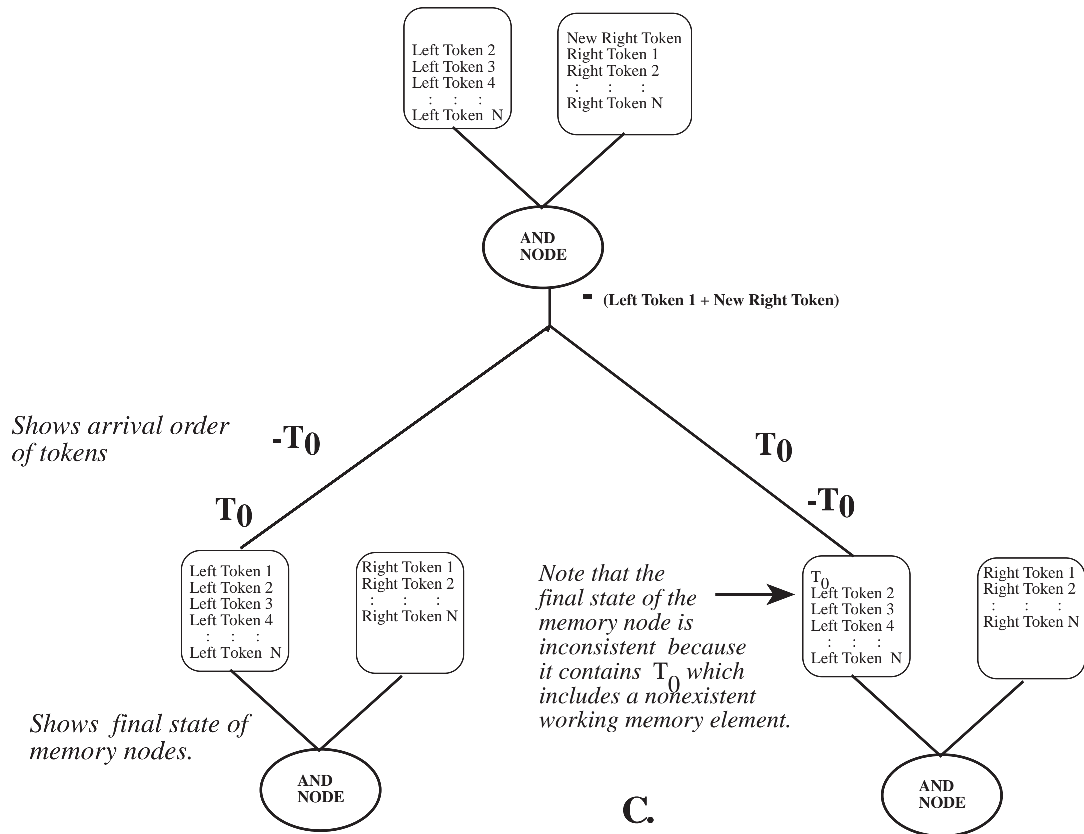


Figure 5.8: Race conditions due to intra-node parallelism.

garbage collection process to periodically examine the node memories and remove unneeded tokens.

5.9 Implementing Action-level Parallelism

Action-level parallelism occurs when multiple working memory changes stimulated by a single rule take place concurrently. From a matching standpoint, once the working memory changes are asserted, they are handled no differently than changes invoked by separate rule instances. The challenge of implementing action parallelism comes from the implementation of OPS5 that defines or compiles the righthand side into monolithic code in which the individual actions are not accessible to be invoked concurrently. In order to access the individual actions, the **in-parallel** and **in-parallel-sync** constructs were devised. These constructs, actually compile-time macros, examine their arguments (which consist of **make**, **modify**, or **remove** functions) and modify them so that they spawn off individual match processes.

In effect, each working memory operation is expanded into a function that pushes a change operation and arguments onto an *action* queue where it is then executed by a rule/action demon. If the ***action-parallelism*** flag is not set, the action-queue-push operation executes the operations instead of pushing them onto the action queue.

The initial naive implementation of action-level parallelism allowed the right-hand side of the rule to compute the token to be added to (or deleted from) memory. This token was then placed on the action queue with a flag indicating whether it was an add or delete operation. It turned out, however, that the process of constructing

the token to be matched against working memory is fairly expensive relative to the matching process. Thus, the righthand side of the rule was only able to push about four actions onto the queue before the first had finished executing, and the benefit due to action parallelism was thus limited to four-fold. This rather contradicts the widely quoted statistic that matching consumes at least 90% of the work of executing a rule. One reason for this may be that action parallelism is most useful during initialization routines in which there is not a lot of matching performed against the elements being asserted.

To increase the potential speedup due to action parallelism, the righthand side was modified so that instead of creating the token to be matched, it simply placed the appropriate **`make, modify or remove`** function, its arguments, and the necessary environment variables onto the action queue. The action demons now were responsible for both constructing the tokens and matching them against working memory. Although this increased the time that the individual action demons were active, the rule instances were able to push actions onto the queue much more quickly and the speedup due to action parallelism increased to at least eight-fold. The (obvious) lesson to be learned from this is that when spawning off processes sequentially (especially at a low level of granularity), the launch time must be minimized. Therefore, all initialization and processing work that can be passed on to the parallel processes should be.

5.10 Summary: UMPOPS

UMass Parallel OPS5 has proven to be a flexible tool for the study of parallel rule-firing systems. However, many improvements could be made to the syntax and

capabilities of the language and the implementation. A number of these changes are being contemplated and may be incorporated in later versions of UMPOPS. Of these, the most crucial are optimizations to the Rete net which will reduce the contention for locks within the Rete net caused by the complete locking of &NOT nodes. As will be seen in the following chapter, this contention can limit the throughput of tokens in the pattern matcher. A related problem is the loss of node-sharing necessitated by the synchronization mechanism described in Section 5.7; the sharing of patterns between rules is one of the virtues of the Rete net algorithm which should be reinstated in UMPOPS. At the level of pattern-matching constructs, UMPOPS should support a richer lefthand syntax, including support for sorting and counting operators and user-defined predicates. The facilities that UMPOPS for parallel programming support such as synchronization groups and the **make-unique** construct are very low-level. As our experience in parallel rule-firing grows, we can expect to be able to add more sophisticated constructs and support facilities including automated analysis programs which can examine rule bases and identify potential sources of rule interactions or inefficient parallel constructs.

CHAPTER 6

EXPERIMENTS

This chapter discusses the main experimental results of my research. The first section is devoted to describing the construction of three parallel rule-based programs in terms of their potential for rule, action, and match-level parallelism and their potential for asynchronous rule execution. The design of these programs demonstrates the techniques discussed in Chapter 3 for avoiding or resolving rule interactions by taking advantage of the semantics of the computation.

The first benchmark, a rule-based implementation of the Waltz line filtering algorithm [Waltz, 1975], was originally written by Toru Ishida; I have substantially modified this program to increase the clarity of the rules and to incorporate UMPOPS constructs, however it remains functionally identical to the original. This benchmark will be referred to as Toru-Waltz. Toru-Waltz exhibits both data parallelism and parallel inference, and contains examples of both initialization and mode-changing rules. The second program (TSP) is an implementation of the travelling salesperson problem; it illustrates the issues underlying the implementation of heuristic control in a parallel asynchronous rule-based program.[1] The final program, Alexsys, a combinatorial optimization program in the domain of high finance

[1]The text of these benchmarks is included in Appendices A and B.

which was developed at Columbia University, demonstrates the parallelization of a process which consists of multiple independent tasks, each of which can be executed asynchronously with respect to each other, while firing rules sequentially within each task.

The second half of this chapter discusses the performance of these benchmarks in terms the overall speedup obtained, the contributions of action and match level parallelism, and the processor utilization achieved. The effects of possible limiting factors such as contention for queues and resources within the pattern matcher as well as scheduling and lock times are measured. Finally, the implications of the results for parallel rule-firing systems are discussed.

6.1 Analyzing the Toru-Waltz Benchmark for Rule Parallelism

The Toru-Waltz program is amenable to parallel rule execution for a fundamental reason – during the course of the program, conflict resolution is never[2] used for the purpose of distinguishing between two valid rules. In most cases, if a rule appears in the conflict set, it is either executable, superfluous, or transient. The principal problems in adapting this program to parallelism are removing extraneous rules from the conflict set and firing the eligible rules at the earliest possible time without accidentally executing a transient instantiation.

The Toru-Waltz benchmark is divided into four stages, each of which supports a considerable degree of concurrent rule firing or matching activities:

- Initialize: Creates a database of the legal junction labels.

[2]Well...hardly ever.

- Make-data: Loads the scene to be analyzed into working memory.
- Enumerate-Possible-Candidates: Lists all the labellings for all the junctions.
- Reduce-Candidates: Eliminates all illegal line labellings.

Both the `initialize` and `make-data` phases of the computation simply consist of adding data to working memory. Rule parallelism can be employed by executing both phases concurrently. To further reduce the initialization overhead, action parallelism can be used to assert the initial working memory elements concurrently. In Toru-Waltz, initialization time was reduced by approximately a factor of eight by combining rule and action parallelism (see Figure 6.1).

In the `enumerate-possible-candidates` phase of the computation, each junction is assigned all possible legal labellings defined by the database created by the initialization rules. Because each junction can be labelled independent of all others, this phase is ideal for asynchronous rule-level parallelism. Each instantiation corresponds to a unique junction and labelling, so rules never conflict. Each instantiation is monotonically enabled by the creation of a junction in the `make-data` phase of the computation, which allows asynchronous rule-firing. The righthand side of the `make-data` and `enumerate` rules perform no `modify` commands, so no transient instantiations appear in the conflict set.

A brief digression is in order here. In the discussion of the rationale underlying the selection of the locking mechanism in Chapter 3, it was asserted that attempts to automatically ensure serializable behavior require both a compile-time and run-time component. Then why should we feel that it is possible to *design* a parallel rule-set which executes correctly when design, of course, precedes run-time? The answer is given above: because the designer is granted a certain knowledge of the

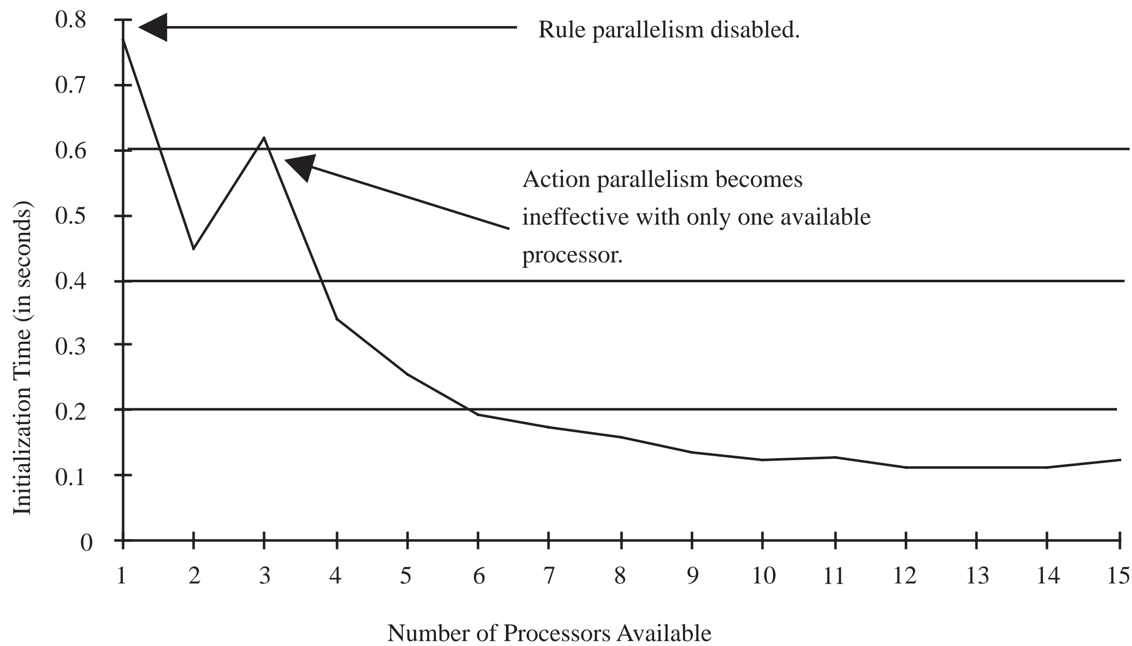


Figure 6.1: The time required by the initialization phase of the Toru-Waltz program can be reduced by the use of rule and action-level parallelism.

nature of the data that is input to the program, it is possible to make deductions about the uniqueness of each rule instantiation. For example, in Toru-Waltz, we know that each junction is assigned a unique label, and therefore there will only be one rule instantiation for each junction/possible-junction-label combination. Thus, while an automated interaction detection mechanism must rely on run-time analysis of rules with instantiated variables to conclude that each instance of the **`enumerate-possible-line-label`** rule can be run in parallel, the designer, by means of a certain omniscience, can conclude the rules will not interact. This, of course, raises the potential of including syntactic methods of conveying this information to compile-time analysis methods, but given the philosophy of designing

for parallel activity, the principal virtue of such a mechanism would be to verify the correctness of the programmer's design.

Because the enumeration rules are added to the eligibility set monotonically (that is, once added, a rule instance will never be removed from the conflict set by another rule firing), it is possible to combine the `initialization` and `enumeration` phases so that the `enumerate` rules can fire asynchronously as soon as they are enabled. Because the scheduler, which manages the mode-changing mechanism, continuously monitors the rule demons for quiescence and the eligibility set for contents, there is no possibility that the system will move on to the next phase of processing before the initialization and enumeration phases have terminated.

The `reduce-candidates` phase consists of rules that detect junctions whose labels are not consistent with any possible labelling of adjacent vertices; these junctions are then deleted. Deleting elements that represent junctions is the only action that takes place during the reduce-candidates phase of processing. Because it is possible for rules to mutually disable each other by deleting existing `possible-line-label` elements, it takes a certain amount of inspection to conclude that the rules in this phase can actually execute concurrently. The operation that is taking place during this phase of the program is constraint propagation. The rules that perform this task can be interpreted as performing logical inference operations; from the discussion in Chapter 3, we know that the problem with a monotonic inference is the possibility of redundant working memory elements. In Toru-Waltz, the result of the inference is to conclude that a junction labelling is incorrect and to remove it. By examination, it can be seen that the rule interaction in the reduction phase is harmless; the effect of executing both rule instantiations concurrently is to execute a redundant `remove` operation; such operations are

semantically valid in OPS5. Although the execution of redundant reduction rules will not affect the outcome of the computation, because the labelling-candidate element is referenced via a positive condition element, the working memory locking mechanism of UMPOPS automatically detects the interacting rule instances and prevents one of them from firing.

6.1.1 Mode Changes in Toru-Waltz

One serializing feature in Toru-Waltz (and most other rule-based programs) is the use of the modal or gating type of working memory element. Modes are typically used to distinguish between the major stages of a computation. In cases where there is no significant parallelism between the stages, the use of mode elements serve a useful purpose in denoting an explicit partitioning among rules. Specific semantics can be assigned to each mode change; for example, in the Toru-Waltz program, the **`reduce-candidates`** mode declares that no new junctions will be created, therefore the relaxation phase can begin. If stages of the computation can overlap or be pipelined, the use of mode elements can cause unnecessary serialization of the computation; in these cases, the rules in the overlapping stages should be placed in the same partition so that they may fire asynchronously.

The use of modal working memory elements can seriously slow down a computation because otherwise eligible productions can not enter the conflict set until the mode of the element is changed. Typically, many productions are enabled by a modification to a modal element and creation of instantiations is relatively expensive, so the overhead of a mode-changing rule may be very high. To give an idea of the magnitude of the problem posed by mode-changing rules, the single rule **`go-to-reduce-candidates`** can consume anywhere from 33% to 50% of the run-time

of Toru-Waltz (out of a total of 370 rule firings) depending on whether match-level parallelism and asynchronous rule-firing are enabled. The time consumed by the **go-to-reduce-candidates** mode-changing rule is reduced by a factor of five by using match-level parallelism (see Figure 6.2). It turns out, however, that even match-level parallelism is not sufficiently fine-grained to efficiently reduce the overhead of matching gating elements. The problem is that the modal element must be matched against all working memory elements that can potentially match the second condition element in the rule's lefthand side. The match process that is assigned this task must create a potentially large number of tokens and place them on the execution queue so they can be propagated by other match processes. Parallelizing the match at a slightly finer level so that multiple match processes could share the overhead of joining the mode element with elements matching the second condition would further reduce the serial bottleneck represented by mode-changing rules.

6.2 The Travelling Salesperson Benchmark

The Toru-Waltz program was simple to parallelize because it presented opportunities for *data* parallelism in which individual rule instances could fire on unique data items. No control mechanisms were required because one instance fired for each potential junction labelling. The travelling salesperson (TSP) benchmark is a more complex example because the search process has to be managed so that there is no interaction between working memory elements in different search states, and rules must be heuristically pruned in order to reduce execution time to a reasonable

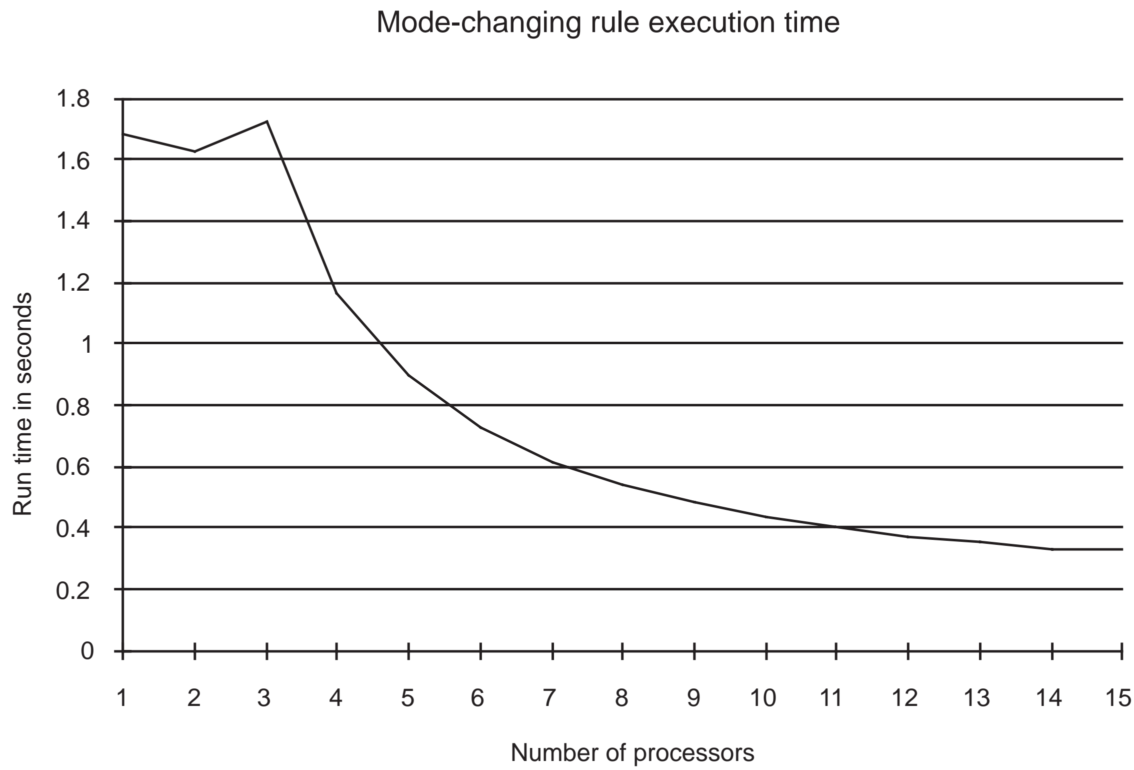


Figure 6.2: The time required to execute the mode-changing production in Toru-Waltz can be reduced by the use of match-level parallelism.

number of rule firings. Finally, solutions from different paths must be compared and the best solution selected.

Search spaces are managed in TSP by assigning all elements in the same state a unique identifier. In the example shown below, a new state is created by generating a new tag value using the `ngenatom` command. (The `ngenatom` function generates a unique integer rather than a symbol in order to avoid the overhead of interning a variable, which is quite high in a shared-memory system). The tag is used to annotate the working memory elements that comprise a state in the search space, i.e. the `so-far` and `connect-goal` elements. The `so-far` element records the distance travelled and the cities seen, while the `connect-goal` element records the

next city to be visited. Because each combination of elements in a state are unique and there is only one operator in TSP, there will be only one rule instance firing per state. This allows the use of the `(meta (lock-not-required t))` notation; because of the partitioning, no rules can conflict during the propagation phase and the overhead of locking working memory elements is not necessary. As will be seen shortly, this is not the case when asserting solutions.

The `map-vector` command, which allows iteration across vectors to be performed in the righthand side, is used in TSP propagation rules to avoid having to use multiple rule firings to spawn all the descendants of the node in the search tree represented by each rule instantiation. Because the nature of the search process in Travelling Salesperson is that few rules are initially active, the `propagate-cities` rules were divided into two types. The initial `start-cities` rule and rule instances matching states with more than five cities[3] left to visit have action parallelism without synchronization invoked in their righthand sides so that matching need not be completed before the next state is constructed and visited. As more states are created deeper in the search space more rules become active and action parallelism is no longer used. With fewer successor nodes, the righthand sides do not take as long to execute and, with more active states, all available processors are already usefully employed in performing search. The rules `propagate-city-5` and `propagate-city-lt-5` are identical from a knowledge-engineering point of view and are distinguished only by the invocation of action parallelism in their righthand sides. This use of duplicate rules to maximize processor utilization is quite unaesthetic and

[3]The benchmark program discussed in this chapter solves a six-city problem which is large enough to be interesting and small enough not to require garbage collection while processing.

is another artifact of the lack of expressiveness in the OPS5 syntax that does not allow conditionals in the righthand side.

The meta-information in the rule shown in Figure 6.3 states that its basic priority is 1 (and all instances of this rule will be placed in queue number 1), that this is a priority queue, the function that computes the priority is `propagate-city-priority-fn`, the heuristic control function associated with this rule is `compare-with-solution`, and if the rule is pruned by the heuristic control function, it should remove the working memory elements indicated by the condition elements <sofar> and <cg>. Both the priority function and the heuristic control function are encoded in Lisp, but are executed within the context of the rule's righthand side and can therefore access not only global Lisp variables, but can also reference any OPS variable bound in the rule's lefthand side through use of the OPS5 `$varbind` function.

6.2.1 Heuristic Control in TSP

The TSP problem is NP-hard, and if all possible solutions were examined, the search space would grow unmanageably for even small values of N. There are a number of well-known admissible heuristics for ordering the traversal of the search space in TSP; the one chosen for this example is the minimum spanning tree (MST) [Pearl, 1984]. Rules are placed in the execution priority queue according to the value returned by the MST heuristic. A record is kept of the best (lowest distance) solution developed so far, and if the value of the heuristic exceeds this value for any search space, the corresponding rule instance is not executed; instead,

```
(p propagate-city-5
   (meta (priority 1) (control-fn compare-with-solution)
         (priority-queue t)
         (lock-not-required t)
         (priority-fn propagate-city-priority-fn)
         (rhs-kill-actions ((oremove <sofar>)
                            (oremove <cg>)))
         )
   <sofar> (so-far ^tag <tag> ^distance <d-so-far>
        ^home-city <home> )
   <cg> (connect-goal ^tag <tag>
                      ^city1 <city1> ^city2 <city2>
                      ^length  >= 5 <l>  )
   (distance ^city1 <city1> ^city2 <city2> ^distance <d>)

-->
   (bind <cities-seen> (litval cities-seen))
   (oremove <sofar>)
      (oremove <cg>)
   (map-vector <cg> city-list
        (item <new-city> vector-less-item <vli>)
      (bind <newtag> (ngenatom))
     (in-parallel
      (make connect-goal ^tag <newtag> ^city1 <city2>
                        ^city2 <new-city>
           ^length (compute <l> - 1)
           ^city-list  <vli> )
      (make so-far ^tag <newtag> ^home-city <home>
                  ^distance (compute <d-so-far> +  <d>)
             ^cities-seen
                (substr <sofar> <cities-seen> inf)
                        <city2> ) ;and the new city
   )))
```

Figure 6.3: The propagate rule from TSP.

the rule's *kill-actions* are executed to remove the corresponding working memory elements and reduce the size of memory.

The heuristic functions are implemented procedurally in Lisp and are specified using the LHS `meta` notation. An early version of this program attempted to compute the MST heuristic using rule firings, however, this resulted in more rule firings to compute the heuristic than to perform the search. For the same reason, rules are pruned using procedural functions in the rule demons; at the level of granularity of search operations in TSP (one rule per state), meta-rule implementations of control methods are not efficient.

6.2.2 Asynchronous Rule-Firing in TSP

An asynchronous rule-firing policy is used in the travelling salesperson program. Because rules are scheduled and executed as soon as they enter the eligibility queue, there is never a time when the system is quiescent and it is therefore impossible to select the "best" of all possible solution paths as indicated by the heuristic evaluation function. It is reasonable to ask whether this causes any degradation in the quality of the solution process as measured by the number of nodes expanded in the search space. In fact, the number of nodes expanded by the TSP program is approximately the same whether the program is executed serially or asynchronously in parallel. There are a number of reasons why this is the case. First, solutions tend to be placed in the execution queues faster than they can be handled by the rule demons. Because the `propagation` rules are placed on a priority queue, the lower quality rules tend not to be executed until after a solution has been found; they can then be pruned without being executed. The second and most important reason why asynchronous rule firing produces acceptable results is that the heuristics used for

TSP are only reasonably precise. Each invalid solution path must be developed to a certain depth before the estimate of the quality of the developing solution becomes good enough that the path can be pruned. Even a strictly best-first algorithm for solving TSP using the minimum spanning tree heuristic must develop a minimum number of nodes in order to ensure that the current solution is indeed the minimum distance. Executing rules asynchronously just means that a certain amount of this work takes place before the solution is found. An analysis of this aspect of the parallel nature of the travelling salesperson problem can be found in [Kumar *et al.*, 1988].

6.2.3 Merging Solutions

The Travelling Salesperson problem was developed primarily to illustrate two points about parallel rule-firing; the elimination of the need for rule interference detection by partitioning the problem into independent states (this idea is developed further in [Neiman, 1992b]) and the idiom for merging results from parallel search processes. Eventually, each parallel search path that has not been pruned terminates and posts a possible solution. Only one solution is acceptable and when competing rules post solutions there is a chance for conflict; either the posting of multiple solutions or the posting of an inferior solution.

Consider the first case, in order to create a **`solution`** element, one needs a rule of the form shown in Figure 6.4. If rules were executed instantaneously, then there would be no difficulties with this rule. However, there is an unavoidable delay between the instantiation of a rule and its firing. This leads to the following possible scenario. An instantiation of the **`init-solution`** rule arrives with a goal to create

a solution of say, 10,000. It is scheduled to fire; once in the execution queue, the rule cannot be disabled, even if a better solution goal arrives. Suppose then, a better solution (produced by some parallel search process) does arrive while the first instantiation is still in the execution queue. It is also scheduled and placed on the execution queue. The result of this scenario is that two solution elements are posted in working memory and a possibly erroneous result is produced.

(If there is a goal to create a solution,
and there is no goal to create a better solution
and there is no current solution
Then
create a solution element.)

```
(p init-solution
   (meta (priority 0))
   <new> (solution-goal ^distance <dist> ^tag <tag> )
         -(solution-goal ^distance < <dist>)
         -(solution)
  -->
 (bind  <cities-seen> (litval cities-seen))
  (make-unique solution ^tag <tag> ^distance <dist>
        ^cities-seen (substr <new> <cities-seen> inf)))
```

Figure 6.4: The `init-solution` rule which initializes a data-merging episode in TSP.

How can this be prevented? The locking scheme cannot prevent this situation because the element to be locked, the `solution` element, does not exist until after the execution of the initialization rule. Instead, we provide a mechanism for creating a *unique* working memory element. A unique element is defined as a class of working memory element with one or more optional key fields. Only one element with a given combination of working memory class and key field values is allowed to exist. In

(If there is a goal to create a solution,
and there is no goal to create a better solution
and the current solution is inferior
Then
assert a new solution element.)

```
(p new-and-improved
 (meta (priority 0))
 <new> (solution-goal ^distance <dist> ^tag <tag> )
       -(solution-goal ^distance < <dist>)
 <old> (solution ^distance > <dist>)
-->
 (bind  <cities-seen> (litval cities-seen))
 (make solution ^tag <tag> ^distance <dist>
       ^cities-seen (substr <new> <cities-seen> inf))
 (remove <old>))
```

Figure 6.5: The `new-and-improved` rule which implements a merge operation for TSP.

the above scenario, if the `solution` class is declared to be unique, the second rule instantiation will not be allowed to fire, because an instance of the `solution` element is already being created. Thus, the rule instance will be disabled and the assertion of the `solution` element will trigger an instantiation of the `new-and-improved` rule, causing the correct solution to eventually be asserted. A similar possibility for interactions occurs with the `new-and-improved` rule (Figure 6.5); two solutions may compete to modify the existing solution. Once again, multiple copies of the `solution` element could potentially be created. In this case, however, an existing element is being replaced and the rule instantiation replacing it must first acquire a write lock on that element. Thus, only one instance of the `new-and-improved` rule is ever scheduled to fire at a given time.

There are a number of points of interest in the solution merging rules: First, the correctness of the solution merging is guaranteed. If a superior solution is ever locked out, the assertion of the new `solution` element will re-trigger the instantiation; thus, the data-directed nature of the rule-based system serves to automatically correct temporary errors. Although the correct solution will always eventually be asserted, it is important that the solution-merging rules perform a check to ensure that the solution being asserted is from the best known solution goal. Otherwise, if many solution-goals were present, each time the solution was modified, it would take many cycles of rule firings before it was ensured that the correct solution was achieved; and many rules would be locked out during each of these cycles.

Finally, note the order in which the actions take place in the **`new-and-improved`** rule. The new solution is asserted and then the previous solution is removed from memory. This is to avoid the creation of transient instantiations of the **`init-solution`** rule. Although the unique-lock mechanism would prevent the **`init-solution`** rule instantiations from firing, avoiding the overhead of creating the instantiations is a good idea.

6.2.4 Queue Latencies in TSP

TSP was chosen as a benchmark because even small problems generate large enough search spaces to provide opportunities for rule-level parallelism and the necessity for heuristic pruning. Each node in the search space (except for the leaf and penultimate nodes) generates multiple successor states and thus the rate of rule generation will exceed the execution rate for any reasonable number of processors. The scheduled but unexecuted rules (that represent the *open* list) remain on the

execution queues until processors become available to execute them. This queue latency time can greatly exceed the actual time of rule execution. It is interesting to briefly examine the effects of queue latency on the performance of the benchmark.

Because the `solution` element is used for performing heuristic pruning, it is necessary to assert new solutions as rapidly as possible. If there were only one execution queue, new solutions would remain in the execution queue for extended periods of time. Propagation and solution rules would compete for resources and inferior search paths would be traversed during the interval between the scheduling of a solution rule and its execution. For this reason, rules that assert solutions are given a higher priority than propagation rules, ensuring that the information used for heuristic pruning is as current as possible.

Once an initial solution has been found (not necessarily the final solution), heuristic pruning can take place. At this point, it would appear possible that the node that would generate the final solution could potentially languish in the execution queue for a long time while lesser solutions were developed, and this could adversely affect the performance of the benchmark. In fact, the combination of the use of priority queues and a reasonable heuristic cause the solution to be developed rather expeditiously. Because the search problem then has to develop many additional nodes in order to eliminate the possibility of a better solution, it turns out that queue latency is not a major problem in a heuristic program that requires a “best” solution. In fact, the contents of the execution queues can be divided up into rules representing nodes that must be “opened” and nodes that will eventually be pruned; it is only if a significant number of the latter are executed while waiting for the former that queue (and scheduling) latency becomes a problem.

6.3 Alexsys: Parallelization of a "Real-World" Rule-based System

This section investigates the issues involved in parallelizing Alexsys, a rule-based system developed at Columbia University in conjunction with Citicorp to handle the problem of fulfilling contracts to trade pools of mortgages in such a way as to maximize profit [Stolfo *et al.*, 1990]. Alexsys represents an expert system which has been developed for use in the "real" world with actual data and differs markedly from the small benchmark programs discussed previously in this work in terms of the complexity of the rules and the size of the working memory database. However, it can be shown that the computations performed by Alexsys can be analyzed in terms of rule parallelism and programmed using the mechanisms provided by UMPOPS, augmented with the control task construct described in Section 4.3. Time constraints have prevented a similar analysis of Alexsys for match and action level parallelism.

6.3.1 The Alexsys program

Alexsys allocates pools of mortgages to contracts in such a way as to maximize profit and minimize loss. Each contract is assigned a "tail" which indicates whether money will be made or lost on that contract. Because PSA (Public Securities Administration) rules allow contracts to be filled plus-or-minus a fixed percentage, profit can be maximized by assigning the maximum allowed value of pools to contracts which are profitable and and the minimum allowed value of pools to unprofitable contracts. In part, the program can be evaluated in terms of how near it comes to achieving the theoretical maximum profit while allocating pools to contracts.

(Pool allocation is an NP-hard combinatorial optimization problem, so a heuristic approach is justified.) In order to fill a contract, entire pools can be assigned to contracts; if no single pool satisfies the requirements, pools may be combined or split under certain guidelines. There are a number of heuristic rules obtained from experts which describe the order in which splitting and combining should take place; these heuristics are intended to result in legal allocations (according to PSA rules) while maximizing profit and minimizing the number of surplus unallocatable pools and unassigned contracts. There are many subtle aspects of the allocation process which are not germane to the discussion at hand; for a more complete description of the Alexsys system and its potential for parallel speedup, see [Stolfo *et al.*, 1990, Stolfo *et al.*, 1991a].

In Alexsys, the control structure of the process by which pools are assigned to contracts is divided into three major parts: initialization, pool allocation, and report generation. During the initialization phase, the most profitable unallocated contract is selected and assigned the appropriate working memory data structures. During the pool allocation phase, pools are selected according to heuristic rules and assigned to a contract. The serial Alexsys, of course, will allocate pools to only a single contract at a time. The approach taken to parallelizing Alexsys was to execute multiple contract allocation tasks simultaneously. When the contract is fully allocated or no more pools are available to be assigned, the control moves to the report phase.

During the report generation phase, the system reports the details of the pool configuration chosen for a particular contract. In the original Alexsys program, this output was generated in a form suitable for a subsequent processing module; output

was also sent to a control terminal. The version of Alexsys discussed here writes all output to a list-based data structure; I/O operations are avoided in order to provide consistency when gathering timing information. Like the allocation phase, the report phase can also support parallel activity; multiple reporting rules may fire concurrently as long as the data structure used for recording results is protected against multiple concurrent writes to the same substructures.

The control structure of Alexsys is predicated partly on the nature of the pool allocation problem and partly on the limitations of the OPS5 language. The heuristics developed to satisfy the PSA restrictions and the requirements of the allocation process impose certain preferences on rule selection and ordering, thus requiring a *synchronous* conflict resolution policy to be applied for each contract allocation task. Because of the inadequacies of the OPS5 conflict resolution routine which cannot specify preferences between specific rule types, the original Alexsys cycled progressively through sets of possible configuration rules of declining utility, using mode-changing rules to investigate each possibility in turn. A true conflict resolution mechanism in which the user was able to state preferences between rules would eliminate much of the need to perform these computationally expensive mode-changing operations. Stolfo and colleagues have investigated the use of the meta-rule constructs of the PARULEL system (see Section 2.5.2) to eliminate the need for mode-changing operations; their results are reported in [Stolfo *et al.*, 1991b]. In the version of Alexsys described in this section, specially designed conflict resolution routines are assigned to control tasks to accomplish the same function.

6.3.2 Parallelizing Alexsys

There are two major sequential influences in Alexsys; contracts are allocated most profitable first, one at a time, and only one allocation rule is applied to a contract per cycle. This last sequentiality, as was pointed out by Stolfo and colleagues, is caused by the need to sequentially update working memory elements acting as state variables to record the progress of the solution, in particular, the amount of "millions" allocated to a contract, and the amount still required. Thus, within a specific contract allocation task, rules execute sequentially, and there is no easily available parallelism (although a divide-and-conquer scheme may be appropriate for allocating pools to very large contracts). For this reason, I have chosen to parallelize Alexsys by allocating multiple contracts in parallel.

The multiple concurrent allocation of contracts raises a number of issues: ensuring that heuristic constraints are not violated, managing resource allocation for multiple tasks, avoiding resource conflicts, avoiding deadlock due to contention for resources, and determining that good solutions can be obtained.

If contracts are allocated in parallel, there are several ways in which the heuristics governing the problem will be violated. If the available pool of mortgages is exhausted during the allocation process and multiple contracts are vying for the last pool in a greedy fashion, then it may not be the most profitable contract which is assigned the pool. Similarly, if a pool satisfies the constraints of two contract allocation tasks, it may not be the most profitable contract which acquires access to it. If multiple allocations are carried out in parallel, then one task may execute a rule which allocates a pool according to a heuristic of lower priority than one enabled by another competing task which references the same rule. For example,

a pool might be eligible to be assigned to a `two-pool-combo` in one allocation and to a `three-pool-combo` in another, where the `three-pool-combo` is heuristically preferred. Because the actual usage of the pool depends on the order in which the two rules become instantiated (assuming an asynchronous firing policy between tasks), it can not be guaranteed that the heuristics will be obeyed.

One of the purposes of the experiments with the Alexsys system was to determine just how pathological the execution of rules in less than ideal heuristic order would be. The intuition is that the results would be acceptable – the Alexsys program is known to be satisficing rather than optimal because of the NP-hard nature of the combinatorial optimization task – violating the heuristics might degrade the solution quality somewhat, but perhaps not dramatically. There are justifications for this belief. The original serial rule-firing Alexsys was extremely slow and the Alexsys research group experimented with methods of increasing the speed of processing. One of the techniques they employed was to partition the pools of mortgages into blocks. The assignment of pools to blocks was performed in a random round-robin fashion, and blocks were then assigned to contracts sequentially. Thus, pools in block $N+1$ would be processed only after all pools in block N, even though there might be pools in block $N+1$ which would satisfy a more highly rated heuristic than any of those in block N. Thus, there is precedent for violating the heuristics; the results that were achieved using the blocking scheme were considered acceptable by the Alexsys evaluators based on the number and size of remaining pools, the number of unfilled contracts, and the percentage of the theoretically achievable profit actually obtained.

The problem of potentially filling a less profitable contract while a more profitable contract goes unfilled remains is not attacked in parallel Alexsys. Stolfo

and colleagues suggest a post-processing phase which tweaks the final solution by swapping "good millions" (units allocated by Alexsys) between contracts to maximize profit.

6.3.3 Modifications to Alexsys

Parallel Alexsys runs in an experimental version of UMPOPS which has been modified to support Alexsys' use of user-defined predicates in the lefthand side. The first step in the conversion of Alexsys to parallel operation was the removal of the separate processing phases required in the serial version to implement the rule-choice heuristics. Each `fill` and `report` rule was given a mode element which served to bind the value of the contract currently being assigned; the instantiations of all rules which could potentially fill a contract were therefore coexistent in the conflict set. A conflict resolution function was written (in Lisp) for each of the two main categories of rules in the `fill` phase, high and low contracts, and another for the `report` phase. Because certain `report` rules could co-execute (because I/O was directed to a data structure rather than a terminal), the conflict resolution routine for the `report` phase was modified to return multiple rules for execution if possible. Because of the inherent sequentiality of the allocation task, only one rule was ever returned by the `fill` conflict resolution routines. The `fill` conflict resolution routines discriminated only on rule *type* – no attempt was made to select the most profitable instantiation for each type, although this would undoubtably have increased the performance of the program. The UMPOPS *task* construct was used to define a context for each contract allocation; each task was specified to be synchronous and was assigned the appropriate conflict resolution function. The addition of common gating elements

to all `fill` rules meant that if a contract *could* be filled, then there would be an instantiation in the task's conflict set after quiescence had been reached. A possible disadvantage was that it was possible that rules would become instantiated at a variable rate and that there would be a synchronization delay while the task waited for quiescence, even though a heuristic rule of the highest rating was available. Because tasks are allowed to execute asynchronously, this synchronization delay is not necessarily significant; if sufficient tasks are executing, then all the processing resources would be engaged in useful activity regardless.

6.3.4 Data Management in Alexsys

The Alexsys program has two main data types, contracts and pools. In the data set supplied with the Alexsys distribution, there are 107 contracts to be filled and 540 pools divided into 107 blocks. Managing this amount of data efficiently required a certain amount of experimentation. The pool allocation process expects contracts to be executed in a specific order, from most profitable to least. The initial approach was to store all contracts in working memory and allocate them as requested using a sorting rule.

If there are fewer active tasks than
the maximum allowed
and there is a contract more profitable
than any others
then
allocate pools for that contract...

```
(p allocate-contract
   (max-tasks ^max <num>)
   (active-tasks ^num < <num>)
   (contract ^tail <val>)
  -(contract ^tail > <val>)
-->
```

```
Create task and add contract to task.
...)
```

Because OPS5 does not perform sorting operations efficiently (the memories of the Rete net nodes are not sorted), rules which order elements do not execute quickly. Therefore, the sorting rule proved to act as a serial bottleneck, slowing down the allocation of contracts to tasks. Allowing multiple tasks to attempt to acquire contracts also proved futile – each task would attempt to grab the most profitable contract and would acquire a write lock on that element. All other instantiations accessing that element would be locked out, and the time required to generate them would have been wasted. The problem was that, functionally, a working memory element was being used as a semaphore variable for controlling access to a critical region. But the latency involved in modifying a working memory element and performing the associated pattern-matching caused unacceptable delays. Finally, it was decided to simply sort the contracts off-line and to define an external function to allocate each contract in sequence. Because a single variable could now be used to implement the critical region, the serial bottleneck associated with contract allocation was reduced to insignificance. There is a lesson to be learned here – the use of working memory elements to represent frequently accessed shared variables between concurrent tasks should be avoided because the latencies inherent in working memory modification and pattern matching will result in serious performance degradation.

A similar problem was encountered when allocating blocks of pools to each task. The first design simply asserted pools into working memory and allowed contract allocation tasks to compete for acceptable pools. This resulted in a large number of "fill" rules competing for the same pool and many fill rules were locked out, resulting in a low ratio of instantiation generation to instantiation generation. The parallel rules for Alexsys were redesigned to incorporate data parallelism – each task was assigned a discrete block of pools. Each pool was explicitly assigned to a task using a tagging scheme for partitioning working memory.[4] All other types working memory elements operating within the context of a task were also annotated so that they matched only other elements in the same task. If pools were left over after a contract was allocated, they were matched by a set rule and returned to the free pool list. As was the case with contracts, because of matching overheads, it proved cumbersome and slow to perform data management of pools using working memory techniques and external functions were defined to import and export pools.

The use of external functions also simplified the task of deadlock detection and avoidance and detecting when no pools remained to satisfy tasks. When no

[4]The multiple worlds construct described in Section 5.4 would have been ideal for partitioning working memory according to contract allocation tasks, however, the version of UMPOPS which incorporated multiple worlds could not support the user-defined predicates required by Alexsys.

pools remain, contracts are annotated as partially specified and sent to the report generation phase – these contracts are said to have *failed.* With multiple tasks vying for pools, there existed the possibility that each task would request a certain number of pools which could not be satisfied unless other tasks released pools they had already reserved for an allocation task. The pool allocation routines maintained a list of tasks waiting for pools and a counter of active tasks – if the number of tasks waiting equaled the number of active tasks, one task was arbitrarily terminated and its pools returned to the free pool list. Pools already allocated to the contract are *not* returned, although this would have been fairly simply to arrange. So it is possible that multiple contract allocation tasks might *fail* although there were initially enough pools to satisfy some of them. This problem was partially avoided by reserving a sufficient block of pools for each contract during the first request for pools.

6.3.5 Experimental Results with Alexsys

The results achieved with the UMPOPS version of Alexsys are not directly comparable with the original OPS5 version due to the extensive modifications to the architecture and rule-firing policy of the program. For the record, the version of Alexsys reported on in [Stolfo *et al.*, 1990] which used the same data set reported on here executed in 4.15 minutes, achieving a profit of 95.6% of the theoretical maximum. This performance was achieved on a Sun 4/60 workstation running at 12.5 MIPS using a version of the program which produced I/O and garbage collected as it ran. The version of parallel Alexsys discussed here did not generate any I/O or garbage collect and executed in 7.6 minutes using a single processor on a Sequent which provides approximately 1 MIPS/processor. The profit achieved was 91.9% of the theoretical maximum. The parallel speedup obtained using the multiple contract allocation scheme is demonstrated in Figure 6.6. The maximum speedup obtained was approximately 9-fold using 19 processors to execute rules and allocating a maximum of 18 contracts simultaneously. The limited speedup is due to the tremendous variance (two orders of magnitude) in the size of contracts in the data set used. Some contracts are quite small and can be satisfied with a single rule firing while others require the allocation of many "good millions" and require many repetitive firing of `fill` rules before the allocation is complete. Because `fill` rules implicitly perform a combinatorial search to generate possible instantiations, they match slowly, and long series of `fill` instantiations are time-consuming. The use of action and match level parallelism to reduce the overhead of `fill` rule execution could profitably be examined, as could the use of conquer-and-divide algorithms for partitioning large contracts into multiple smaller allocation tasks. The overhead due to large contract allocation tasks can be seen in Figure 6.7 which displays the processor utilization for a parallel Alexsys run in which a maximum of 18 contracts are simultaneously allocated. All contract allocation tasks are initiated by time 20, and processor utilization remains very high up to that point; during the remaining

30 seconds, processor utilization gradually drops off as allocation tasks terminate (the spikes in the processor utilization graph indicate the places where an allocation task terminates and multiple report rules fire).

The most surprising aspect of the parallel Alexsys experiments is that the quality of the solution as measured by percentage of maximum profit obtained actually increases steadily as more contracts are allocated in parallel (see Figure 6.8). In all cases, all contracts were filled and only one or two pools remained unallocated. This is contrary to expectations; it was thought that allocating less profitable contracts in parallel with more parallel contracts would lead to reduced profitability. It would be interesting to determine whether this trend persisted for other data sets than the one provided with the Alexsys distribution.

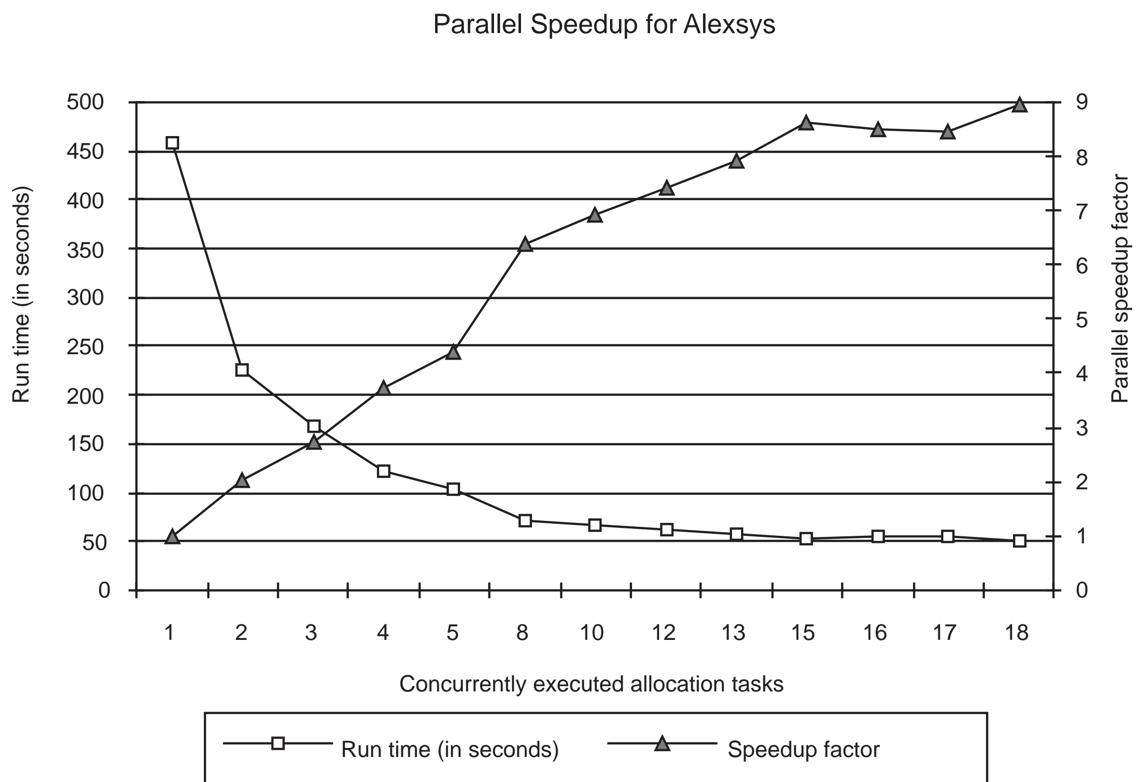


Figure 6.6: The parallel speedup achieved for Alexsys.

6.3.6 *Conclusions: Alexsys*

The motivation for the experiments with Alexsys were to demonstrate the use of rule-level parallelism for a large, complex, "real-world" program and the use of control tasks to allow multiple sequential tasks to execute asynchronously in parallel. The results of these experiments were quite promising – the parallel speedup of

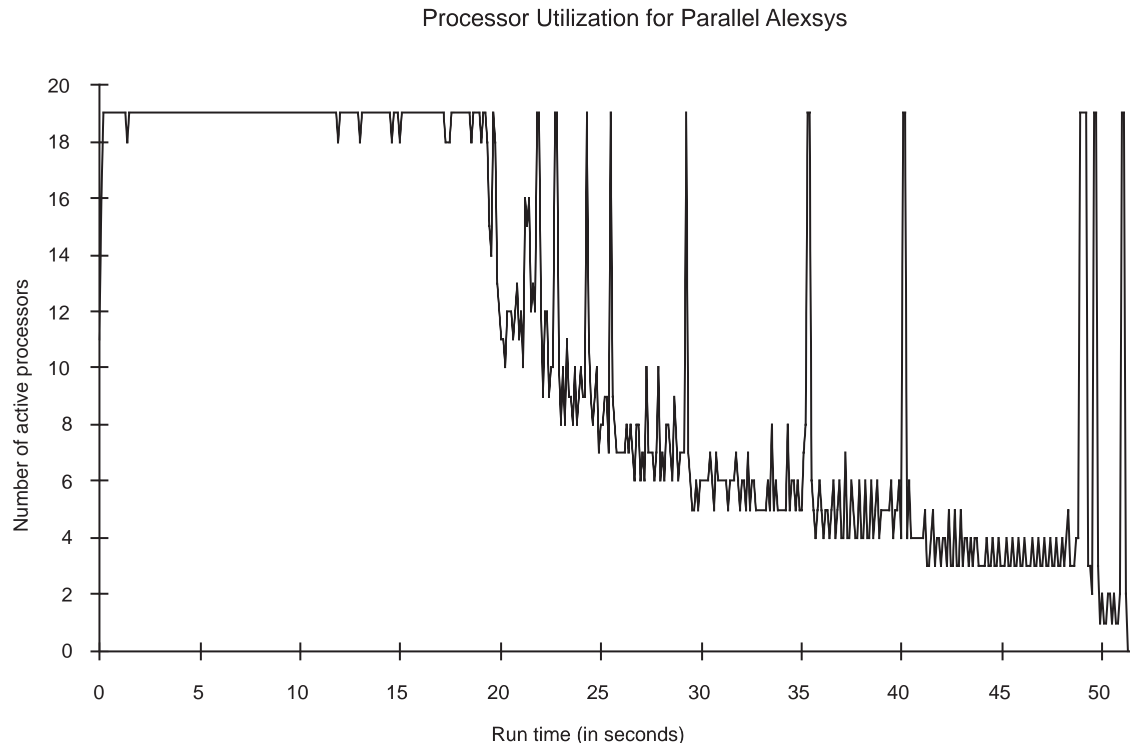


Figure 6.7: The processor utilization graph for parallel Alexsys with a maximum of 18 concurrent allocation tasks.

Alexsys was quite good, considering the unbalanced nature of the contracts to be allocated, and there exists a potential for further parallel speedups due to action and match level parallelism and partitioning of large contracts. The use of control tasks allowed a high level of processor utilization for much of the allocation process despite the fact that each individual allocation task was required to achieve quiescence before a rule could be selected to fire. The primary difficulty in developing parallel Alexsys was in managing large amounts of data and allocating this data in a fair manner to competing processes; perhaps the current research in integrating production and database systems (e.g. [Miranker, 1990a]) will ease this problem in future rule-based systems.

6.4 Performance Analysis of Toru-Waltz and TSP

The rationale behind parallelizing a rule-based system is to increase the performance of the system. Ideally, the nature of the speedup should be linear or near-linear to the number of available processors. In practice, there are a number of effects that limit the performance of UMPOPS and they are discussed in this section in the context of the results obtained with the benchmark programs discussed in the previous section.

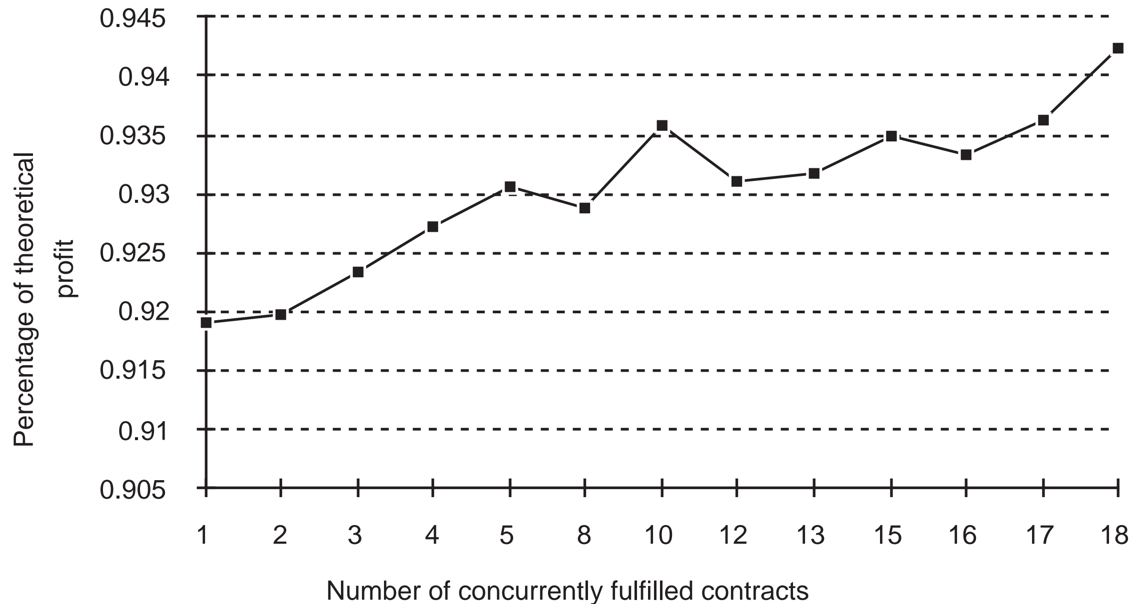


Figure 6.8: The solution qualities produced by the parallel Alexsys allocation process graphed against maximum concurrent allocation tasks.

6.4.1 Performance Measurements: Toru-Waltz

The speedup due to all levels of parallelism in Toru-Waltz is roughly a factor of eight or nine, independent of the number of processors available beyond that number (see Figure 6.9). This ceiling is due to a number of factors: the presence of the mode-changing rule that represents a serial bottleneck and limits rule parallelism; the "tapering-off" effect which is observed as the program grows close to a solution and requires fewer processing resources; and contention for resources within the pattern matcher itself.

The performance of Toru-Waltz is measured relative to the speed of the benchmark using a single rule demon. The speedup is more dramatic (10-fold) when measured against the serial version of the program that employs conflict resolution; because conflict resolution is not necessary in Toru-Waltz, the speedup observed during asynchronous rule-firing is due partly to its elimination and not to parallel activity. In general, this holds true for all benchmarks which have been explored using UMPOPS (see table 6.1). For this reason, all speedups reported in this dissertation are reported relative to the single rule-demon case using the UMPOPS scheduler.

Because both action and match parallelism are employed in Toru-Waltz to reduce the overhead due to initialization and mode-changing, the overall speedup

Table 6.1: Performance of the nondeterministic UMPOPS scheduler using a single rule-demon compared with the performance of a serial OPS5 scheduler using conflict resolution for three benchmark programs.

Benchmark	Serial run time (UMPOPS scheduler)	Serial run time (OPS5 conflict resolution)
Travelling Salesperson	31.50	31.9
Toru-Waltz	10.6	11.6
Circuit (3000 cycles)	54.55	61.1

is not expected to be linear. As a general rule, each successively lower level of granularity of parallelism, rule, action, and match, generates less of an overall speedup due to the proportionate overhead required in generating the parallel action. Not only is the overhead of smaller granularity operations greater relative to the cost of the operation, but, because there are typically many more small granularity operations, the absolute cost of the overhead becomes significant.

The speedup for action parallelism in a single rule is approximately 8-fold in UMPOPS. Match-level parallelism yields a speedup factor of five in the single mode-changing rule in which it is employed.[5] For an insight into the performance of Toru-Waltz, we can examine the processor utilization graph for this benchmark (see Figure 6.10.).

During the initialization phase, Toru-Waltz makes use of rule and action parallelism. Two rules fire simultaneously to add data to the system and each rule employs action parallelism to reduce run time. Because all actions must be completed before moving to the next phase, two processors must be reserved to perform synchronization and the maximum level of action parallelism is $N-2$ where N is the total number of processors devoted to processing demons. During the mode-change from the `enumerate` phase to the `reduce candidates` phase, only a single rule can fire and match parallelism is employed to reduce the serial bottleneck. Finally, as the algorithm approaches completion, the amount of work to be performed (and the demand for processors) falls off significantly. This is due to the nature of the constraint propagation algorithm; each junction is connected to at most three other junctions and thus, each rule firing can initiate at most three rule firings. Because the Toru-Waltz benchmark is fairly small, the significance of these three limiting phases is relatively large, and the potential speedup is limited.

The principal bottleneck in Toru-Waltz is the overhead due to the `go-to-reduce` mode-changing production. If this rule is included, the maximum rule execution

[5] There is no reason to think that this speedup is an absolute limit, but optimization of match-level parallelism is not a high-priority in this research project.

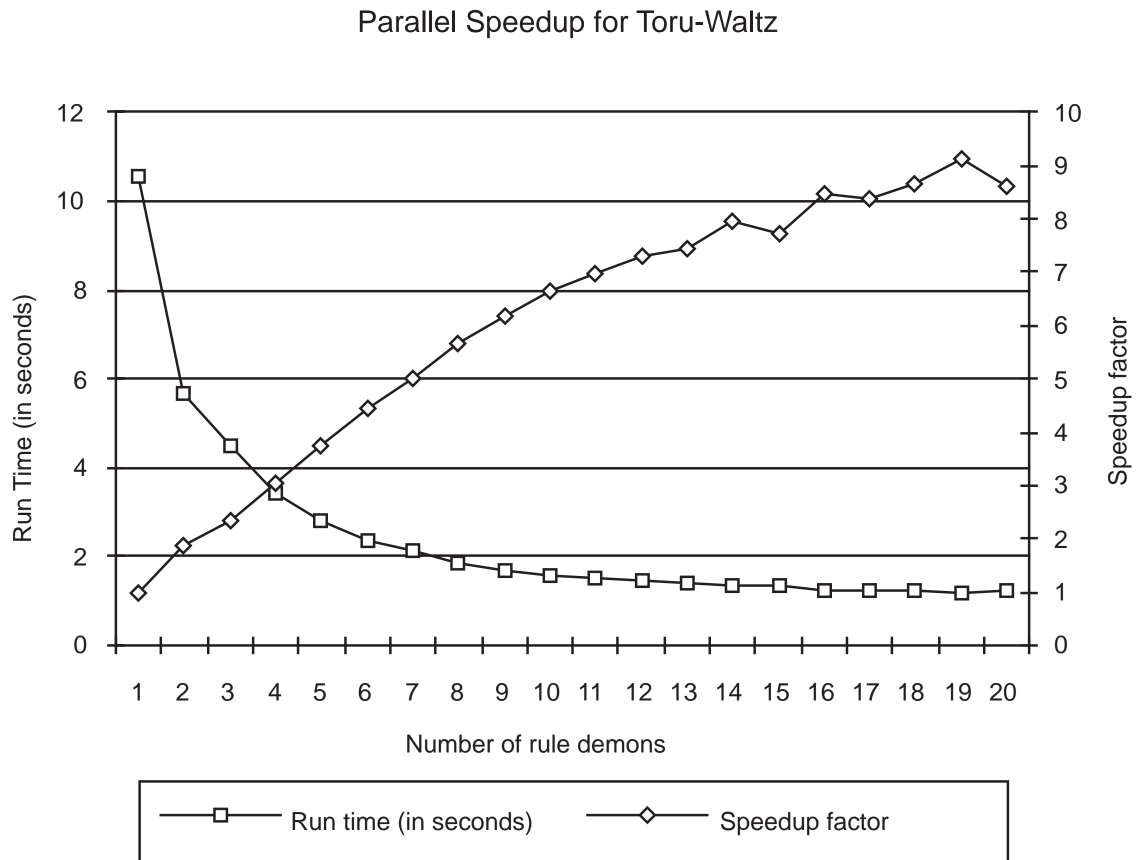


Figure 6.9: The parallel speedup graph for Toru-Waltz with 20 processors. The left axis shows run-time in seconds. The right axis shows the speedup factor achieved.

rate for Toru-Waltz is about 300 firings per second, while if this rule is disregarded, the average rule execution speed approaches 500 firings/second (see Figure 6.11). Clearly, the optimization of match parallelism would further decrease the overhead of mode-changing rules.

6.4.2 Performance Measurements: TSP

The processor utilization for the Travelling Salesperson benchmark is shown in Figure 6.12. As can be seen in this graph, the level of rule parallelism is very high in this benchmark after the initialization phase has been completed. Theoretically, the speedup due to rule parallelism in the Travelling Salesperson Problem should be linear and Figure 6.13 demonstrates that this is approximately the case. The

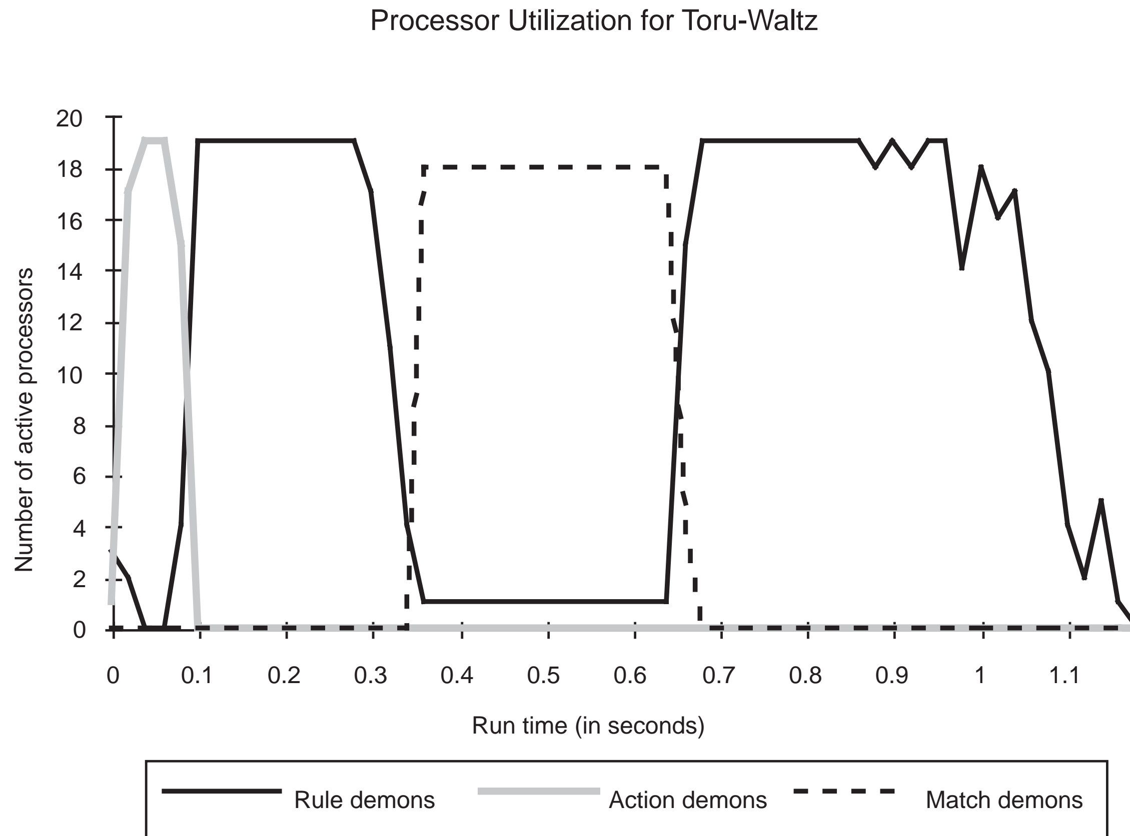


Figure 6.10: The processor utilization graph for Toru-Waltz with 19 "demon" processors. In the initialization phase, a large number of action demons become active while date is added to working memory. In the `enumerate` phase, 19 rule demons become active. During the mode change from the `enumerate` phase to the `reduce` phase, 18 match demons are active. One rule demon must be active during this time to perform synchronization. Finally, rule parallelism becomes dominant during the `reduce` phase and tapers off as less work becomes available.

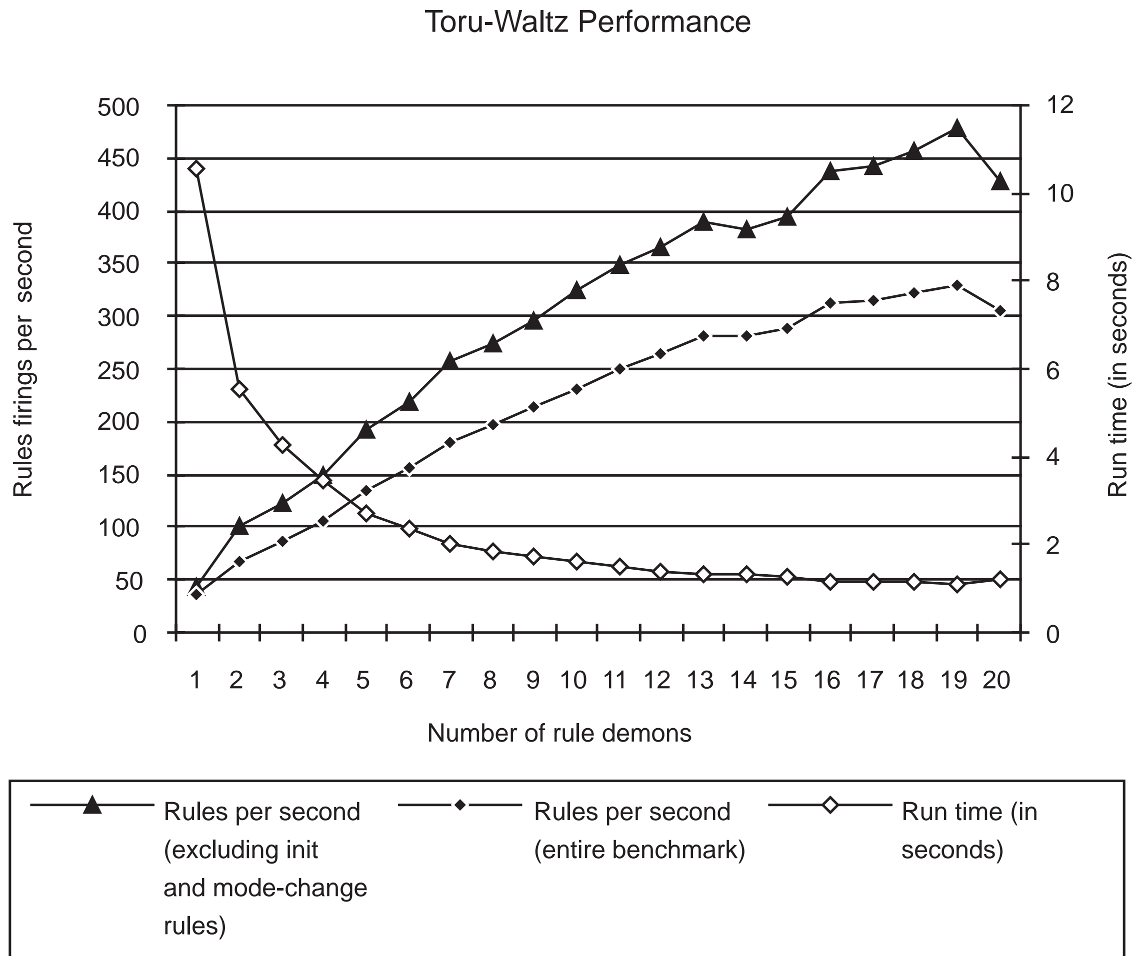


Figure 6.11: The parallel speedup for Toru-Waltz in terms of rule firings per second. With periods of reduced parallel firing due to initialization and mode-changing eliminated, rule firing rates approach 500 rules per second.

observed speedup is approximately 17.5 using 20 processors[6]. The observed speedup departs slightly from the linear for reasons discussed in the following section.

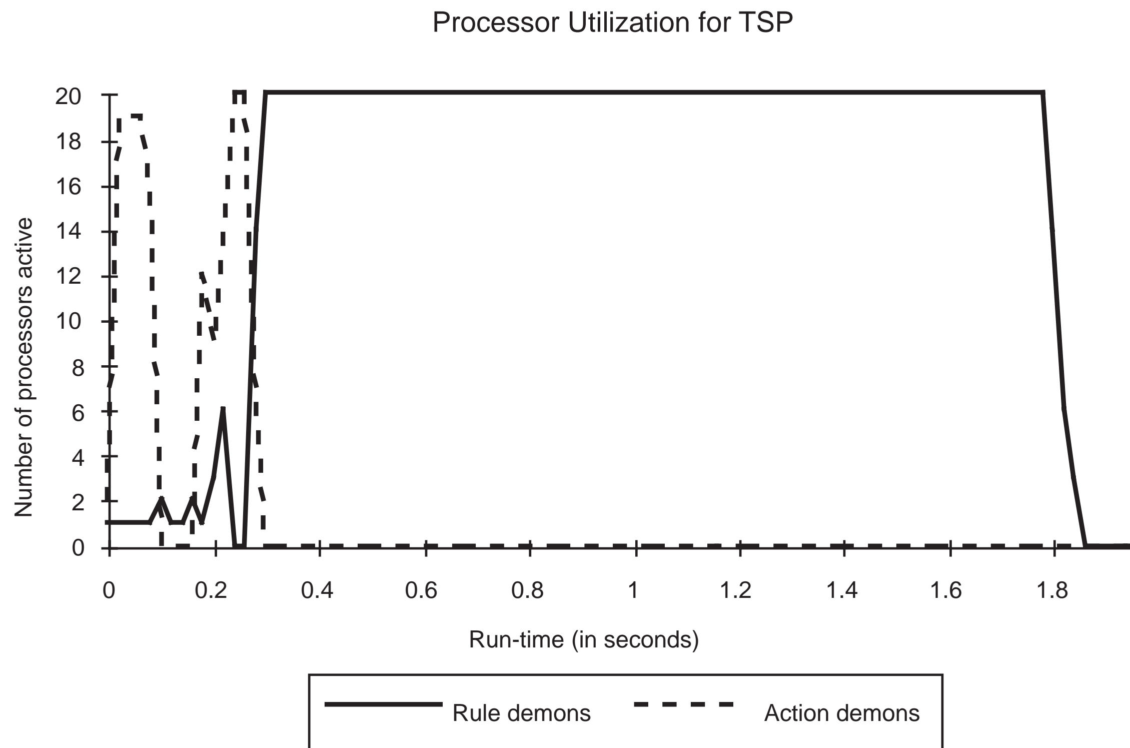


Figure 6.12: The processor utilization for the TSP benchmark with 20 processors.

6.4.3 Measurements of Contention in the Rete Net

In the experiments with TSP and Toru-Waltz, we have seen a steady decrease in run time and increase in performance as more processors are applied to the problem, however, the increase departs from the linear as the number of processors increase. In particular, the speedup for the Toru-Waltz program is disappointing. The departure from linear performance was acceptable when experiments were run using 15 processors to execute rules, but when, late in the research program, another 5 processors were added, the effect became more pronounced and it became necessary to examine the sources of slowdown more carefully. There are a number of possible explanations for the departure from linear performance, including the character of the Sequent's scheduler which causes processors to be time-shared on a random basis,

[6]In UMPOPS, one processor is always reserved for the scheduler and therefore, the maximum number of rule demons is 20 on a 21 processor machine. To provide consistency in benchmarking, one processor is usually reserved for system functions leaving 19 rule demons available.

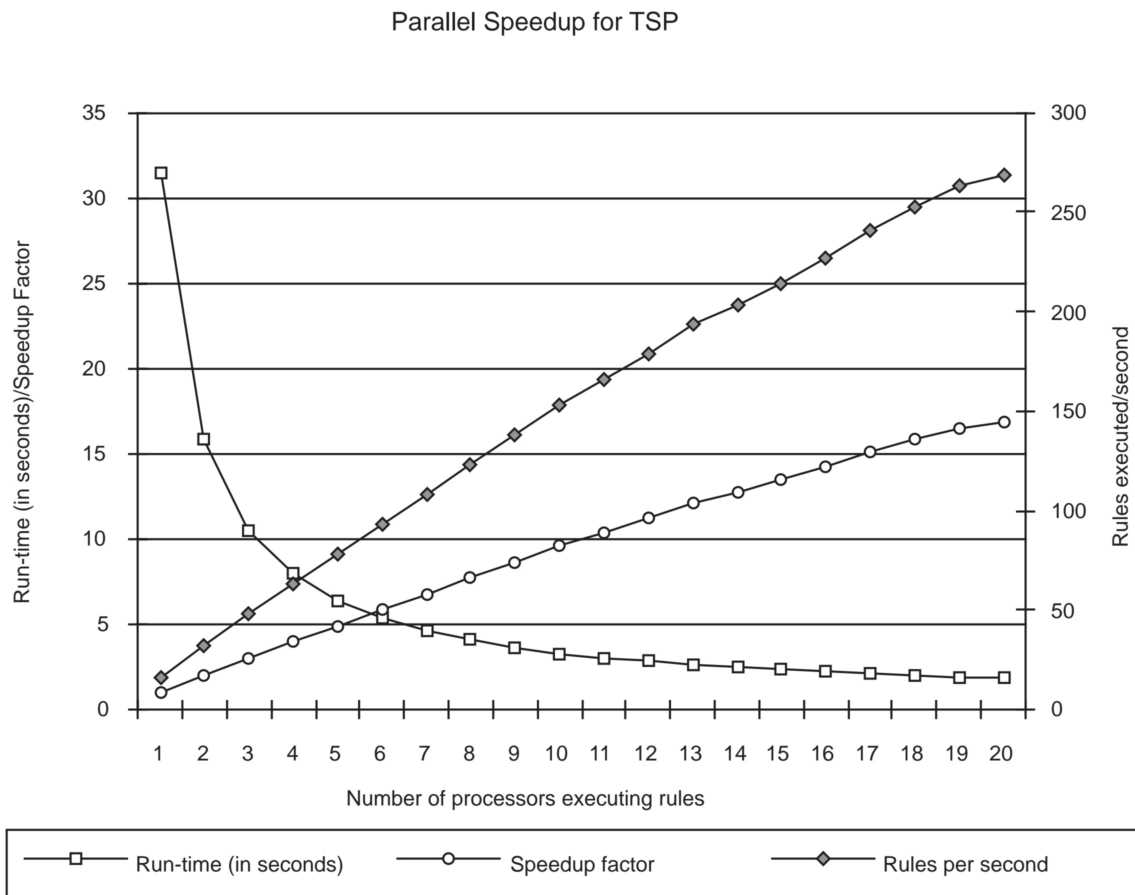


Figure 6.13: The parallel speedup for the TSP benchmark with 20 processors. The graph depicts the decrease in execution time as the number of processors increases, the ratio of parallel run-time to serial run-time, and the number of rules fired per second.

cache faults within the shared memory architecture, and contention for resources within the Rete net,. Because the Sequent provides no support for measuring the first two, this section will concentrate on the latter.

Although the contention of the various queue demons for the rule, match, and action queues would appear to be a potential bottleneck, measurements of processor utilization indicate that the processors spend very little time idle waiting for access to queues. Because the rule demons are active virtually all the time, but parallel speedup is non-linear, the conclusion is that the average time required to actually execute a rule must increase as the number of processors increases. The average execution times for various rules have been measured for the Toru-Waltz and TSP benchmarks and are presented in Figures 6.14 and 6.15 respectively.

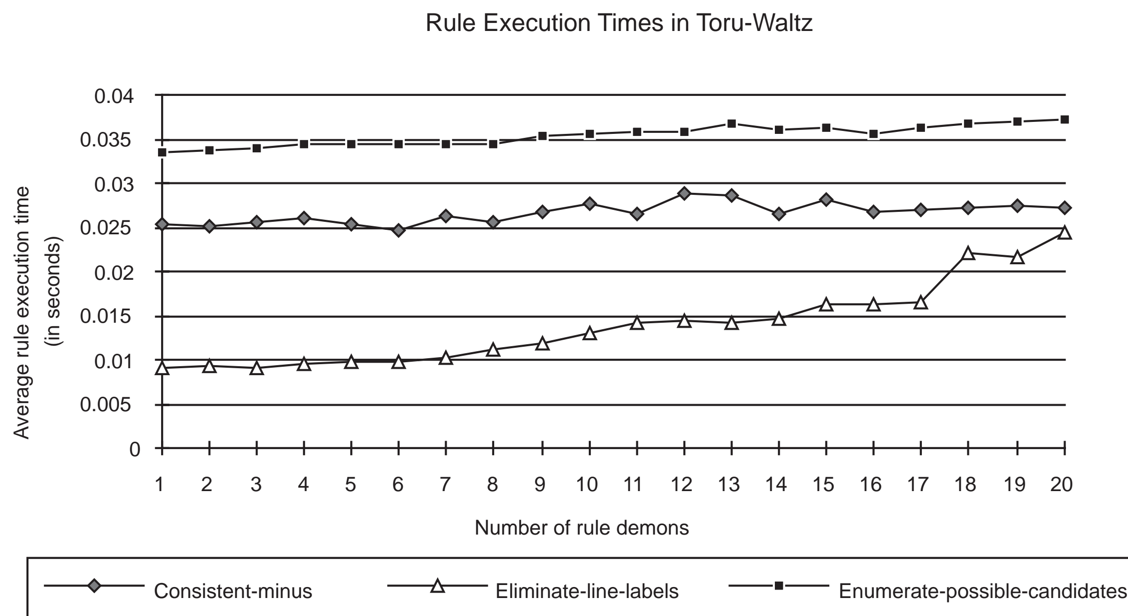


Figure 6.14: The average time required to execute certain rules in Toru-Waltz, plotted against number of processors.

Examining these graphs, we can see that the Travelling Salesperson benchmark, which displays very good parallel speedups, demonstrates only a very small increase in average rule execution times ($< 5\%$) as the number of processors increases. Toru-Waltz, on the other hand, shows a more marked increase in rule execution times, with one rule, `eliminate-line-labels`, increasing by 250% when 20 rule-demons are active. Examination of the rule sets reveals the source of the different behaviors. TSP contains few rules with negated condition elements; while all the rules in the `reduce` phase of Toru-Waltz contain negated elements. As was noted in Section 5.7, it is necessary to lock both memories of &NOT nodes during processing in order to eliminate synchronization problems. &AND nodes only require that

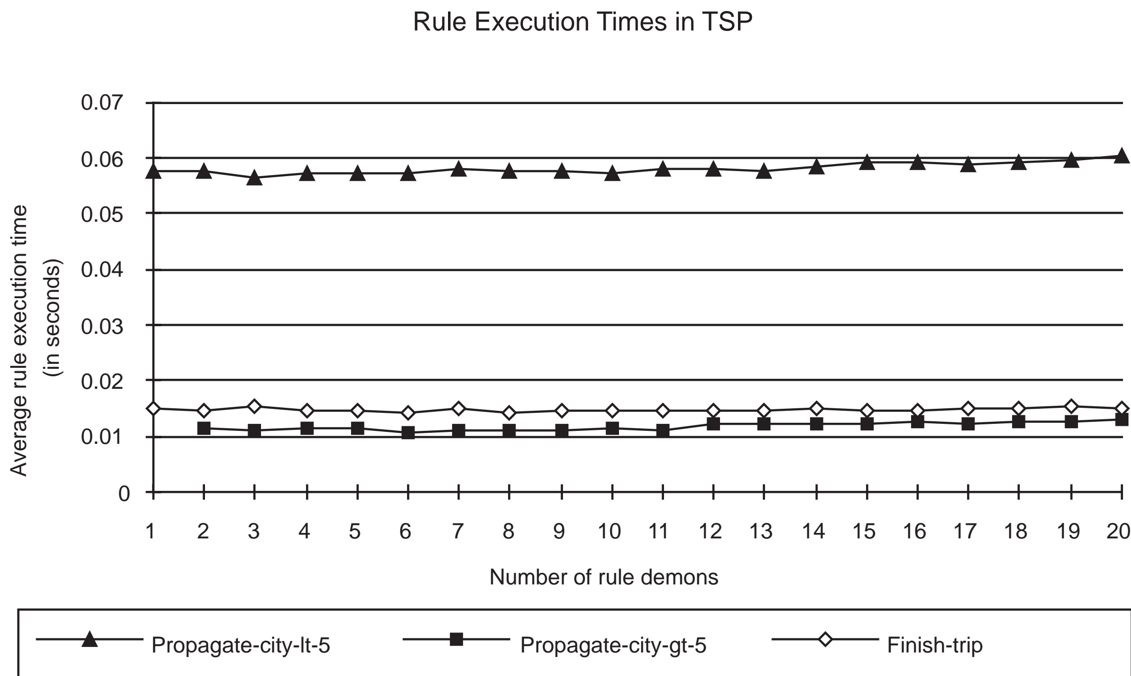


Figure 6.15: The average time required to execute certain rules in TSP, plotted against number of processors.

locks be obtained during the actual insertion of tokens into their associated memory nodes. Thus, the potential for contention for locks is much greater for &NOT nodes. Although UMPOPS incorporates hashed memories which reduces the competition for locks substantially, as the throughput of tokens through the network increases, contention for memory locks also rises. To give an example of the magnitude of the problem, the rate at which tokens must be processed by nodes of the Rete net in Toru-Waltz rises from approximately 40 per second for the single processor case to approximately 500 per second with greater than 12 processors. Because it requires an average of 2/1000's of a second to process a single node of the Rete net (according to data gathered by monitoring the match demons), saturation occurs at this point and the addition of further processors yields only slight gains in performance.

Contention for resources can rise considerably if the number of processors being employed approaches the number available on the machine. At this point, background or system tasks will begin to compete for processing resources because the Sequent performs symmetric multiprocessing and assigns processors equally to all processes, Under these load conditions, it becomes increasingly probable that a processor will be swapped out while it has acquired control over a critical region; the swapping delay is then multiplied by the number of processes waiting to get access to that critical region. Observe that in Figure 6.14, the rule execution times increase significantly as the number of processors approaches 20, the maximum for the configuration of the Sequent on which this experiment was run.

Given the results obtained here, particularly those of the Toru-Waltz benchmark, contention for memory locks within the Rete net would seem to limit the potential performance of parallel rule-firing systems to only a single order of magnitude. But the performance of the Toru-Waltz benchmark is probably due to a number of factors which are not typical of larger rule-based systems. Toru-Waltz is a very homogeneous system with a small number of rules which employs instance parallelism to obtain much of its concurrency. This results in a concentration of matching in a very small area of the Rete net. The rules are also quite small, frequently performing only a single RHS operation, thus, the possible degree of rule parallelism is contingent upon the speed at which that element can be processed. With larger rulebases performing more complex tasks, the match operations can be expected to be spread more diffusely throughout the Rete net, resulting in fewer collisions.

It may also prove possible to increase the rate at which nodes can process data either through optimization of the beta node code, or by applying low-level parallelism to speed the matching process. Contention for memory nodes can be decreased through the use of compilation techniques, for example, the "copy and constrain" algorithm devised by Pasik [Pasik and Stolfo, 1987] could be used to split bottleneck nodes into two or more equivalent nodes. Further research is required both to identify the effects of contention on larger and more sophisticated rule bases, and to study methods of reducing or eliminating the sources of that contention.

6.5 Summary

This chapter has examined three programs modified to take advantage of parallel rule execution: Toru-Waltz, Travelling Salesperson, and Alexsys. These programs illustrate three situations in which parallelism is effective: data parallelism, in which operations can be applied to many different objects in parallel; parallel search, in which many search paths can be explored concurrently; and task parallelism, in which multiple independent tasks can be pursued. Some of the issues underlying parallel rule-based programming have been discussed, including effective use of lock mechanisms, avoiding transient instantiations, merging solutions, and making effective use of action- and match-level parallelism during initialization and mode-changing rules. In Toru-Waltz and TSP, an asynchronous rule-firing control policy was used and, in TSP, it was shown how heuristic control mechanisms could be incorporated into a rule-based program without resorting to synchronizing conflict resolution schemes. Alexsys demonstrated the use of *control tasks* to allow multiple tasks, each consisting of a series of synchronous, sequential rule firings, to execute asynchronously in parallel. The performance characteristics of each program were analyzed and it was shown how multiple levels of parallelism could be used to optimize performance. The experimental results of this chapter, particularly the near-linear performance of the TSP benchmark, indicate that the performance of

rule-firing production systems need not be limited to a single order of magnitude; although the limited number of processors available makes it impossible to empirically identify the upper limits of performance. An equally valuable result has been the identification of those aspects of programs which limit the speedup due to parallel rule-firing. Further research would be profitable to identify methods for reducing contention within the Rete net, for subdividing tasks to allow load-balancing when co-executing synchronous tasks must be developed, and for reducing the serial overhead associated with mode-changing rules. One final observation resulting from these experiments is that the development of parallel rule-firing programs is still far from an exact science; each of these programs underwent a cycle of development, testing to identify bottlenecks and unexpected rule interactions, followed by redesign to eliminate inefficient constructs.

CHAPTER 7

Conclusion

The chapter presents an overview of the main points of this dissertation, summarizes the conclusions, and describes the future work which remains in the area of designing and controlling parallel rule-firing production systems.

7.1 Motivation

The motivation behind any research in parallelism is to increase the performance of the system being parallelized. Because of the disappointing speedups observed by Gupta [Gupta, 1987] in parallelizing expert systems at the match level, it is of considerable practical interest to determine whether there are also intrinsic limits on the degree of parallelism which can be exploited at the rule level in production systems. More generally, by demonstrating high degrees of potential parallelism in rule-based systems, it may be possible to generalize the results to other A.I. systems which employ the same general paradigms.

The primary conclusion to report from this research, then, is that there do *not* appear to be any intrinsic limits on the amount of parallelism available in parallel rule-firing systems due to scheduling or control issues; what limits there may be can be found in the applications for which, and the architectures upon which, the system is implemented. This conclusion is based on a number of assumptions, first, that serial conflict resolution or rule-interaction detection algorithms are avoided, that an asynchronous rule-firing policy is employed, and that, for large applications, multiple lock managers will be employed to evade the 1% overhead due working memory locking. As reported in the previous chapter, a potential bottleneck exists due to contention for resources within the pattern matcher, but can be largely avoided through use of low-level parallelism, redesign of the pattern-matcher, and restructuring of the rulebase.

7.2 Contributions

The contributions of this research stem from a simple observation; any serial delay between the instantiation of rules and their execution would seriously reduce the number of rules which can be executed concurrently. This hypothesis was

supported by the experiments described in Chapters 3 and 4. These experiments verified that serial conflict resolution or rule-interaction detection could limit parallel activity to a single order of magnitude; the limit was based on an analysis of the ratio rule-firing times to the necessary serial overhead of control and correctness activities. The level of granularity of rule executions simply makes any activity which relies on pairwise comparison between rules (and the attendant synchronization) an unacceptable bottleneck. The major problem attacked in this thesis is how to reduce or eliminate the serial overhead of control and rule-interaction detection: the two main sequential activities which are carried out in a processing phase distinct from rule execution and matching activity. Because the emphasis of this research is on increasing the parallel activity available in a rule-firing system, consideration has also been given to reducing the serial bottlenecks associated with imposing sequential control activities on rules: this is achieved by providing enhanced RHS mapping and iteration operators, incorporating a set-oriented syntax into OPS5, and by allowing precise application of match and action-level parallelism to reduce the overhead of single-rule bottlenecks.

In the pursuit of the elimination of serial bottlenecks in parallel rule-firing systems, this research has made contributions in three interrelated areas: the correctness of parallel rule-firing systems, the control of rule firing, and the design of systems to exploit parallelism, both at the rule level and at lower levels of granularity. These areas are interrelated in that one can design rule-based systems to minimize the runtime overhead of correctness checking and the presence of serializing control constructs, while the mechanisms for controlling rule firing and ensuring correctness influence and constrain the design of the system.

7.2.1 Control and Rule-Firing Policies

This thesis proposes two novel rule-firing policies for parallel production systems; an asynchronous rule-firing policy which allows rules to execute as soon as they become enabled, and a task-based control system in which tasks can execute asynchronously with respect to each other while executing rules either asynchronously or synchronously within the context of an individual task. Previous implementations of parallel rule-firing systems have retained the synchronizing conflict resolution phase of processing which requires that all eligible rules be detected before conflict resolution can take place. The synchronization cost of conflict resolution is high because the arrival time of rules in the conflict set cannot be expected to be simultaneous; rule execution times are not necessarily uniform and multiple rules stimulated by the same working memory change necessarily will be instantiated in a staggered fashion. Due to this serialization delay, eligible rules may remain in the conflict set for a considerable length of time while processing resources remain idle. After conflict resolution occurs, multiple rules must be launched simultaneously,

causing potential contention for shared resources in the scheduler and pattern matcher.

7.2.1.1 The Asynchronous Rule-Firing Policy

An asynchronous rule-firing policy was developed which eliminated the synchronizing conflict resolution phase by allowing rules to execute as soon as they became instantiated. To avoid race conditions, the criteria for rule-firing correctness were extended to describe the conditions under which rules may be fired asynchronously. Specifically, the concept of *monotonic* entrance into the conflict set was introduced and local synchronizing constructs were added to the RHS syntax of UMPOPS to avoid transient instantiations caused by parallel assertion of working memory elements.

In an experiment using a highly parallel benchmark (Section 4.2.1), the overhead of synchronous conflict resolution under the most favorable circumstances, rules of roughly equal execution time and no actual processing during the conflict resolution phase, produced a degradation of performance of approximately 40% compared to an asynchronous rule execution policy with an equal number of processors.

The results of this experiment led to the question, "Is synchronous conflict resolution avoidable?". This thesis argues that, for most applications, serial conflict resolution can, in fact, be eliminated. Observation of the use of the conflict resolution in many benchmark programs reveals that conflict resolution is used almost exclusively for program sequencing purposes.When these program sequencing constructs are replaced by imperative RHS code and/or annotated control rules which are coordinated by the scheduler rather than the matching process, the resulting rules sets can, in most cases, be executed non-deterministically, that is, without requiring any decisions about rule firing order.

7.2.1.2 Heuristic Control in Asynchronous Rule-Firing Systems

Not all rule-based programs can support completely non-deterministic rule-firing. The canonical example of an application which requires order and pruning decisions to be made is heuristic search. Although conventional rule-based systems cannot represent search easily, performing search by mapping a computation into an AND/OR tree is one way to exploit concurrency in a parallel rule-firing system. The fact that rules coexisting in the conflict set represent alternate search paths does not automatically imply that all rules can or should be executed simultaneously in a search for the best possible solution. Although this may be true in cases in which there are only a few alternate solution paths and the paths are fairly short, in many cases, there may be many alternate solutions, each representing a sizable investment in resources to investigate. It is inevitable in such cases that examining all possible alternatives would saturate the available processing resources. Thus,

rules representing alternative operators must be sorted according to importance and pruned as they become redundant or unnecessary.

For those applications which require distinctions to be made between eligible rules, it was necessary to demonstrate that the heuristic control could take place incrementally with little or no degradation of solution quality. The Travelling Salesperson problem was chosen as a heuristic search task which is characteristic of a wide class of A.I. problem-solving activities. A control architecture which allows heuristic pruning and ordering to be applied at several points in the rule-firing cycle: match, instantiation, scheduling and execution time was described. Using this architecture and the Travelling Salesperson benchmark, experiments demonstrated that the number of rule executions required during asynchronous rule-firing with incremental control activities was virtually identical to the number of rules fired using a serial best-first rule-firing policy. The explanation for this result, which in turn justifies asynchronous incremental control, is that in a system which attempts to derive an optimal solution, the use of admissible (underestimating) heuristics requires that even after the solution has been found, considerable additional search must be performed to ensure that no better solution exists. Informally, we can divide the set of rules which become eligible during the course of the search process into two groups; rules which, once the solution has been identified, can be pruned using the heuristic, and rules which must be executed regardless. Parallel asynchronous execution becomes inefficient only if significant numbers of the former group get executed. This happens only if a reasonably good approximation of the solution is not developed quickly, or if the available number of processors is so large that even low priority rules get executed. Thus, we can conclude that minor deviations in rule-ordering from a strictly best-first search due to asynchronous rule-firing do not result in extraneous or redundant rule firings. Because the heuristic used is admissible and rules are pruned only if they exceed cost of the lowest cost solution developed, we can also be sure that the correct answer will be developed.

7.2.1.3 Task-based Rule-Firing Policies

Occasional applications will require strict adherence to best-first execution, for example, the Alexsys program described in Section 6.3 requires that rules be evaluated in a specific order in order to reduce database fragmentation. A *task-based* architecture was described in Section 4.3 which allows multiple tasks to execute asynchronously with respect to each other while executing rules either asynchronously or synchronously and with user-defined conflict resolution schemes within the scope of a single task. When conflict resolution schemes are defined within the context of a task, all rules which execute in the context of that task must be synchronized and all match operations in the context of that task must become quiescent before the task's next rule (or set of rules) can be executed. Tasks are a control mechanism and do not guarantee data independence; when tasks which

access the same pool of working memory are allowed to execute asynchronously with respect to each other, the question of determining quiescence becomes quite difficult. This thesis suggests partitioning the rule set into multiple sets of working memory, those which are required to be quiescent before a task can execute and those elements for which quiescence is desirable, but not necessary. Currently, determining local quiescence in a multiple task system remains a research problem.

7.2.2 Explicit Control of Programming Sequencing

The control of program sequencing in production systems takes two forms: phase transitions and trivial iterative operations.

The difficulty of phase transitions lies in ensuring that sets of rules which share a producer-consumer relationship execute in the correct order. A method was described in Section 5.2.1 for annotating rules whose purpose is to sequence or control a computation. By allowing the scheduler to fire these rules only when quiescence has been achieved among domain rules, interactions which could result in the non-firing of domain rules are avoided. An additional benefit of explicit annotation of control rules is that the use of the specificity condition of conflict resolution for control purposes is no longer necessary; this language feature alone made the standard `MEA` and `LEX` conflict resolution routines virtually unnecessary in applications written in UMPOPS.

Iterative operations are composed of repetitive operations performed over sets of working memory elements. In OPS5, such operations are carried out by multiple sequential rule invocations. This is inefficient because the overhead of conflict resolution and matching is encountered during each iteration. UMPOPS provides two mechanisms for composing multiple rule executions into one: mapping operations and a set-oriented syntax. The set-oriented syntax allows a single rule-instantiation to match a set of patterns; the righthand side of the rule is then mapped over members of that set. A similar mapping operation is available to reduce the overhead in manipulating working memory elements containing vectors.

7.3 Correctness

Based on the cost analysis and experiments described in Chapter 3, this research argues in favor of replacing techniques for guaranteeing full serializability with a working memory locking scheme which does not guarantee correctness but which provides sufficient expressive power to allow correct programs to be designed, based on a semantic analysis of the role of each rule in a computation. Comparison of the overhead imposed by the working memory locks with a full run-time analysis of rule interactions indicates that at least a full order of magnitude improvement in performance can be expected by designing programs so that run-time rule interaction detection is not required.

The design viewpoint espoused in this thesis is a departure from the classical view of production systems in which rules are assumed to be completely data-driven and fire independently of each other; instead, it accepts (and exploits) the *de facto* state of affairs which is that well-understood AI paradigms such as heuristic search, planning, and goal-directed problem-solving are routinely imposed upon rule-based computations. The view that the rule-based computation has an algorithmic structure allows us to attach a semantics to rule firing. By examining the role of each rule in the overall computation, we can understand and begin to find a solution to the problems of controlling rule firing and ensuring correctness while maximizing effective use of parallel processing resources. Chapter 3 described a partial taxonomy of the semantic roles of productions in a computation and discussed methods and language idioms for implementing each usage using only a positive locking scheme with low overhead.

Common idioms explored in this chapter can be summarized as follows:

- Search: Conflicting rules can frequently be interpreted as competing operators in a search space. By mapping the execution of each rule into a discrete state space, the execution of such rules can take place without the necessity for interference detection mechanisms or even locking schemes. These savings may be offset by the need for copying of working memory states. Two methods of implementing discrete state spaces for search are described, one which simply annotates working memory (and thus imposes a partitioning cost on the pattern-matcher) and one which actually partitions the memory of the Rete net into discrete pools, thus allowing more complex states to be represented more easily. A programming idiom for merging the results of parallel search was developed; although such idioms must test for the prior non-existence of a working memory element, the use of the **make-unique** operator allows the merge idiom to be implemented without recourse to either region locks or full run-time interaction detection mechanisms.

- Resource Allocation: The use of working memory elements to represent resources in configuration problems was discussed and a transformation was described for converting rules which contained tests for the non-existence of a condition to positive tests for specific enumerable resources which could be guaranteed to be correct using standard database locking techniques.

- Mode-changing or sequencing rules: Mode-changing rules are those which modify working memory elements in order to activate or deactivate sets of rules corresponding to computational phases. These rules will always disable rules within the phase in which they terminate. Because these rules are typically coexistent in the conflict set with the domain rules, any rule interaction detection algorithm must be avoid allowing the mode-changing rule to fire in lieu of eligible domain rules, thus prematurely terminating a computational

phase. The annotation scheme described previously avoids this problem by allowing the scheduler to perform the arbitration, rather than the pattern matcher.

- Data Parallelism: When a direct one-to-one correspondence can be made between "real-world" objects and working memory elements, then operations can usually be applied independently to each working memory object. If objects are interconnected (e.g., semantic nets, graphs, or circuit representations), then only interactions propagated through the interconnections represent potential sources of conflict. Because these interactions are localized and purposeful, the propagation mechanisms can usually be modified so that positive locking mechanisms are sufficient to maintain correctness.

7.4 UMPOPS: A System for Benchmarking Parallel Rule-based Programs

The assertions made about the overhead of control activities and the detection of rule interactions have been empirically verified using UMass Parallel OPS5 (UMPOPS), a Lisp-based version of OPS5 which supports parallelism at the rule (instance), action, and match levels and is extensively instrumented to record critical timing data about the execution of rules in parallel. The ability to experiment with parallel rule-firing schemes using a working system distinguishes this research from much previous research in this area. Although systems which provided match parallelism have been available for some time [Kalp *et al.*, 1988, Tambe *et al.*, 1988], most results on rule parallelism have been simulated (there are recent exceptions, for example [Kuo *et al.*, 1991, Boulanger, 1988, Harvey *et al.*, 1991]).

Early use of the UMPOPS system to explore rule parallelism led to the first empirical observations of the serializing nature of conflict resolution when executing rules in parallel and motivated the development of the the parallel asynchronous rule-firing model. The experience with conflict resolution led to the hypothesis that run-time rule-checking would also exhibit the tendency to limit parallelism by imposing a synchronizing delay upon the system. These observations combined led to the twin threads which run through this research: first, the design paradigm underlying rule-based systems must be modified to incorporate parallelism into the design phase; and second, control activities must occur contemporaneously with rule execution in order to provide adequate performance. In the process of developing systems to test using UMPOPS, many modifications were made to the syntax and capabilities of the language, including:

- A scheduler which supports synchronous, asynchronous, serial, and task-based rule-firing policies.

- A "meta-syntax" which allows instructions to the scheduler to be attached to rules which determine rule priority, locking requirements, and associated heuristic functions. The meta-syntax also allows the programmer to explicitly annotate control rules which are not to be executed until quiescence has been reached for all domain rules.

- RHS constructs for the support of parallel activity, including explicit invocation of match and action-level parallelism, local synchronization mechanisms, and the **make-unique** operator which allows working memory elements to be created uniquely.

- An optional set-oriented syntax and RHS mapping functions for eliminating unnecessary sequential invocation of rules.

Largely because of its Lisp implementation, UMPOPS is highly flexible, allowing experimentation with several variations of the scheduler and rule-demon processes. Multiple experimental versions of UMPOPS have been devised including one which employs a partitioned Rete net to support parallel search using a multiple-worlds paradigm (see Section 5.4) and a version which guarantees serializable results using a full locking scheme including checks for disabling conditions due to negated condition elements (see Section 3.2.4). Although not as fast as versions of OPS5 employing compiled versions of the Rete net, UMPOPS running serially is significantly faster than the public domain version of OPS5 due to its use of hashed memories and multiple optimizations. When executing rules in parallel, execution speeds of greater than 700 rules/second have been measured in UMPOPS for very simple programs, and rule execution rates of greater than 400 rules/second are not uncommon.

One surprising observation about UMPOPS is the parsimony of the language constructs required to support the design of correct parallel programs which require only positive working memory locks. The only novel constructs required to support the design of the benchmarks described in this thesis were the *make-unique*, task, and group synchronization operators. It remains to be seen whether additional language constructs will be required to support more complex real-world applications.

7.5 Future Work

This thesis has laid the foundation for the design and control of parallel rule-firing systems, but much research remains to be done. A formal theory needs to be developed which describes the range of problems which can be solved using only a simple locking scheme. While I claim that positive locking is sufficient for resolving *most* of the rule interactions encountered in common algorithms superimposed upon the rule-based paradigm, the limits of this approach are not yet known. The discussion of design techniques in Chapter 3 is largely anecdotal. It would be desirable to characterized the completeness and representational adequacy of the

UMPOPS language, specifically enumerating the kinds of algorithms that can be handled using the language constructs provided. Since a complete enumeration of rule uses is lacking, many more parallel rule-firing programs will have to be developed before any kind of theory of software engineering for parallel rule-firing systems can be developed.

This dissertation has not discussed the limitations which shared memory architectures might impose upon parallel rule-firing. If bus contention or cache failures become a problem, then very low-level programming techniques may be required to provide adequate performance. For example, the memories of nodes in the Rete net could be designed so that hash buckets fall on distinct pages of memory, so multiple processes could access the same node without causing memory cache failures and excessive bus traffic.

The questions of scaling up the size of applications, working memory, and rules have not been completely addressed; Alexsys is the one benchmark program discussed which supports both complex rules and large databases. The argument for avoiding full run-time rule interaction detection and serial conflict resolution has been based on measurements of the ratio of these activities to rule execution times. Whether these observations would remain valid for applications with large and complex lefthand sides or very large working memories remains to be seen.

The control idioms devised for UMPOPS are limited; if one assumes that the tradition of imposing complex algorithms on the rule-based paradigm will continue, then the language must be extended to support these algorithms. Although there is support for search and single level task activations, more complex problems will require language constructs for managing hierarchies of goals and subgoals so that redundant goals can be eliminated while important subgoals are prioritized. UMPOPS does not provide any mechanisms for supporting truth maintenance; thus, some loss of concurrency is inevitable due to local and global synchronization. Constructs for supporting optimistic concurrency and asynchronous retraction of rule-firings such as those proposed by Wolfson[Wolfson *et al.*, 1990] should be studied to determine whether they can be implemented in a cost-effective manner.

At the implementation level, there are a number of enhancements which could be made to the UMPOPS language. Compilation of the Rete net would increase the speed of the system considerably. A number of modifications would allow more efficient match-level parallelism. Currently, match-level parallelism is inefficient because nodes which require very little computation are parallelized, saturating of the processing demons. "Gating" nodes which match single modal elements against large numbers of working memory elements in the opposing memory cannot be parallelized efficiently because UMPOPS does not support fine-grained intra-node parallelism. This research has revealed a number of deficiencies within the Rete matching algorithm itself; including the inability to efficiently support a number of common idioms, including counting and sorting elements. A frustrating deficiency of UMPOPS from an AI point of view is the inability of the Rete pattern matcher

to improve its performance for a given rule set over a number of trials. A promising research area is the devising of mechanisms to allow the Rete net to learn characteristics of a rule set and modify itself to more efficiently match incoming working memory elements.

APPENDIX A

The Toru-Waltz Benchmark

```
;
;  INITIAL OPS5 VERSION OF WALTZ'S ALGORITHM        by Toru Ishida
;

;  Modified by Dan Neiman
;              COINS Dept.
;              University of Massachusetts

;  11/16/90: Added possible-line-label element.  One element is added
;  for each possible labelling of each end of each line.   This allows
;  easy testing for consistent line labelling without proliferation of rules.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                             ;
;       Data & Knowledge Structure for Waltz's Algorithm                      ;
;                                                                             ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (literalize possible-junction-label junction-type line-1 line-2 line-3)

 (literalize junction junction-type junction-ID line-ID-1 line-ID-2 line-ID-3)

 (literalize labelling-candidate junction-ID line-1 line-2 line-3 l-c-ID)

 (literalize possible-line-label line candidate junction label)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Knowledge of Possible Junction Labeling   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(literalize possible-junction-label junction-type line-1 line-2 line-3)

;                            \   /
; Junction type : L        1 \ / 2
;                             V
```

```
(p initialize
   (meta  (no-lock-required t))
   (stage initialize) --> ;(remove 1) (make stage make-data)
 (in-parallel-sync
 (make possible-junction-label ^junction-type L
                              ^line-1 out ^line-2 in ^line-3 nil)

 (make possible-junction-label ^junction-type L
                              ^line-1 in  ^line-2 out ^line-3 nil)

 (make possible-junction-label ^junction-type L
                              ^line-1 +   ^line-2 out ^line-3 nil)

 (make possible-junction-label ^junction-type L
                              ^line-1 in  ^line-2 +   ^line-3 nil)

 (make possible-junction-label ^junction-type L
                              ^line-1 -   ^line-2 in  ^line-3 nil)

 (make possible-junction-label ^junction-type L
                              ^line-1 out ^line-2 -   ^line-3 nil)

;                            1 \ / 3
; Junction type: FORK           V
;                             2 l

 (make possible-junction-label ^junction-type FORK
                              ^line-1 +   ^line-2 +   ^line-3 +  )

 (make possible-junction-label ^junction-type FORK
                              ^line-1 -   ^line-2 -   ^line-3 -  )

 (make possible-junction-label ^junction-type FORK
                              ^line-1 in  ^line-2 -   ^line-3 out)

 (make possible-junction-label ^junction-type FORK
                              ^line-1 -   ^line-2 out ^line-3 in )

 (make possible-junction-label ^junction-type FORK
                              ^line-1 out ^line-2 in  ^line-3 -  )

;                           1 _____ 3
; Junction type: T              l
;                               l2

 (make possible-junction-label ^junction-type T
                              ^line-1 out ^line-2 +   ^line-3 in)

 (make possible-junction-label ^junction-type T
                              ^line-1 out ^line-2 -   ^line-3 in)
```

```
 (make possible-junction-label ^junction-type T
                              ^line-1 out ^line-2 in  ^line-3 in)

 (make possible-junction-label ^junction-type T
                              ^line-1 out ^line-2 out ^line-3 in)

;                                /l\
; Junction type: ARROW        1 / l \ 3
;                              /  l2 \

 (make possible-junction-label ^junction-type ARROW
                              ^line-1 in ^line-2 +  ^line-3 out)

 (make possible-junction-label ^junction-type ARROW
                              ^line-1 -  ^line-2 +  ^line-3 -  )

 (make possible-junction-label ^junction-type ARROW
                              ^line-1 +  ^line-2 -  ^line-3 +  )))


;;;;;;;;;;;;;;;;;;;;;;;;;;
; Scene to be Analyzed   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;

;(literalize junction junction-type junction-ID line-ID-1 line-ID-2 line-ID-3)

;
;                           ------------>
;
;
;                          A               B
;                         / \             / \
;                      1 /   \ 2        3/   \ 4
;                       /  C  \         /  D  \
;                      / 5/l\6 \       /  /l\  \
;                   E /10/+l \  \     /  / l \  \
;                    l\+/  l  \  \   / 7/ 8l  \9 \
;                    l l F l-  \  \ /  /   l   \  \
;                    l l12 l41 +\  G  /+   l-   \  \
;                  11l l+  l     \   /     l     \  \
;                    l l   l     L\ /    M l     +\  \
;                    l l14/K\      l      / \      \39\ O
;                 H  l l-/   \15   l16 17/   \18   N\+/l
;      ^            /-\l/  P  \-   l+  -/  Q  \-     l l       l
;      l           / 13J  /l\  \   l   /  /l\  \   19l l20     l
;      l        21/      / l \  \  l  /  / l \  \   +l l       l
;      l        /    22/  l  \  \ l /  /  l  \  \   l l       l
;      l       R/      /   l 24\  \l/  /   l 27\  \  l l       V
;              l\30   /+ 23l   +\  T  /+ 26l   +\  \ l l
;              l \+  /    -l     \   /25  -l     \ W\l l
;              l  \S/      l    U \ /      l    V \ /+ l
```

```
;            29l   l       l       l       l       128 l
;              l 31l      /X\    32l+     /Y\     +l   l
;              l   l+    /   \     l     /   \   33l   l
;              l   l    /     \    l    /     \    l   l
;            Z \   l   /40     \   l   /       \   l   / DD
;               \  l  /       35\  l  /36     37\  l  /38
;              34\ l /           \ l /           \ l /
;                 \l/             \l/             \l/
;                  AA              BB              CC
;
;                              <---------
```

```
(p make-data
   (meta  (no-lock-required t))
   (stage initialize)
  -->

  ;(remove 1)
  ;(make stage enumerate-possible-candidates)
(in-parallel-sync
(make junction ^junction-type  L ^junction-ID  A
              ^line-ID-1       2  ^line-ID-2    1   ^line-ID-3    NIL)

(make junction ^junction-type  L ^junction-ID  B
              ^line-ID-1 4 ^line-ID-2    3   ^line-ID-3    NIL)

(make junction ^junction-type  ARROW ^junction-ID  C
              ^line-ID-1 5 ^line-ID-2    41  ^line-ID-3    6)

(make junction ^junction-type  ARROW   ^junction-ID  D
              ^line-ID-1 7 ^line-ID-2    8   ^line-ID-3    9)

(make junction ^junction-type  ARROW ^junction-ID  E
              ^line-ID-1 11 ^line-ID-2    10  ^line-ID-3  1)

(make junction ^junction-type  FORK ^junction-ID  F
              ^line-ID-1 10 ^line-ID-2    12  ^line-ID-3    5)

(make junction ^junction-type  L ^junction-ID  G
              ^line-ID-1 2 ^line-ID-2    3   ^line-ID-3    NIL)

(make junction ^junction-type  FORK    ^junction-ID  H
              ^line-ID-1 11 ^line-ID-2    21  ^line-ID-3    13)

(make junction ^junction-type  ARROW ^junction-ID  J
              ^line-ID-1 14 ^line-ID-2    12  ^line-ID-3    13)

(make junction ^junction-type  FORK ^junction-ID  K
              ^line-ID-1 41 ^line-ID-2    14  ^line-ID-3    15)
```

```
(make junction ^junction-type  FORK ^junction-ID  L
              ^line-ID-1 6 ^line-ID-2    16  ^line-ID-3    7)

(make junction ^junction-type FORK ^junction-ID  M
              ^line-ID-1 8 ^line-ID-2    17  ^line-ID-3    18)

(make junction ^junction-type  FORK ^junction-ID  N
              ^line-ID-1 9 ^line-ID-2    19  ^line-ID-3    39)

(make junction ^junction-type  ARROW ^junction-ID  O
              ^line-ID-1 4 ^line-ID-2    39  ^line-ID-3    20)

(make junction ^junction-type  ARROW ^junction-ID  P
              ^line-ID-1       22 ^line-ID-2    23  ^line-ID-3    24)

(make junction ^junction-type  ARROW ^junction-ID  Q
              ^line-ID-1 25 ^line-ID-2    26  ^line-ID-3    27)

(make junction ^junction-type  ARROW ^junction-ID  R
              ^line-ID-1 29 ^line-ID-2    30  ^line-ID-3    21)

(make junction ^junction-type  FORK ^junction-ID  S
              ^line-ID-1 30 ^line-ID-2    31  ^line-ID-3    22)

(make junction ^junction-type  ARROW   ^junction-ID  T
              ^line-ID-1 17 ^line-ID-2    16  ^line-ID-3    15)

(make junction ^junction-type  FORK ^junction-ID  U
              ^line-ID-1 24 ^line-ID-2    32  ^line-ID-3    25)

(make junction ^junction-type  FORK ^junction-ID  V
              ^line-ID-1 27 ^line-ID-2    33  ^line-ID-3    28)

(make junction ^junction-type  ARROW ^junction-ID  W
              ^line-ID-1 19 ^line-ID-2    18  ^line-ID-3    28)

(make junction ^junction-type  FORK ^junction-ID  X
              ^line-ID-1 23 ^line-ID-2    40  ^line-ID-3    35)

(make junction ^junction-type  FORK ^junction-ID  Y
              ^line-ID-1 26 ^line-ID-2    36  ^line-ID-3    37)

(make junction ^junction-type  L ^junction-ID  Z
              ^line-ID-1 29 ^line-ID-2    34  ^line-ID-3    NIL)

(make junction ^junction-type  ARROW ^junction-ID  AA
              ^line-ID-1 40 ^line-ID-2    31  ^line-ID-3    34)

(make junction ^junction-type  ARROW ^junction-ID  BB
              ^line-ID-1 36 ^line-ID-2    32  ^line-ID-3    35)
```

```
 (make junction ^junction-type  ARROW ^junction-ID  CC
              ^line-ID-1 38 ^line-ID-2    33  ^line-ID-3    37)

 (make junction ^junction-type  L ^junction-ID  DD
              ^line-ID-1 38 ^line-ID-2    20  ^line-ID-3    NIL)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Temporal Labelling Candidates ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(literalize labelling-candidate junction-ID line-1 line-2 line-3)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                            ;
;           Production Rules for Waltz's Algorithm                           ;
;                                                                            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;
; Start ;
;;;;;;;;;

 (p start-Waltz
    (meta  (no-lock-required t))
(start)
-->
(remove 1)
(make stage initialize))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Enumerate Possible Candidates ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (p enumerate-possible-candidates
    (meta  (no-lock-required t))
(stage initialize)
(junction ^junction-type <j-type> ^junction-ID <j-ID>
                 ^line-ID-1 <l1> ^line-ID-2 <l2> ^line-ID-3 <l3>)
(possible-junction-label ^junction-type <j-type>
                       ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
-(labelling-candidate ^junction-ID <j-ID>
                       ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
-->
        (bind <l-c-ID> (ngenatom))
(make labelling-candidate ^junction-ID <j-ID>  ^l-c-ID  <l-c-ID>
                       ^line-1 <line-1> ^line-2 <line-2> ^line-3 <line-3>)
```

```
        (make possible-line-label ^line <l1> ^label <line-1> ^candidate <l-c-ID>
              ^junction <j-ID>)
        (make possible-line-label ^line <l2> ^label <line-2> ^candidate <l-c-ID>
              ^junction <j-ID>)
        (make possible-line-label ^line <l3> ^label <line-3> ^candidate <l-c-ID>
              ^junction <j-ID>))


 (p go-to-reduce-candidates
        (meta (rtype mode-changer)
              (no-lock-required t))
(stage initialize)
-->
(remove-match-parallel 1)
(make-match-parallel stage reduce-candidates))


;;;;;;;;;;;;;;;;;;;;;
; Reduce Candidates ;
;;;;;;;;;;;;;;;;;;;;;

;If a line is labelled '+' on one end, than it must be labelled '+' on the
;other end.
 (P  consistent-plus
(stage reduce-candidates)
 {<line>(possible-line-label ^line <line> ^junction <junction>
                             ^label + ^candidate <c>) }
 {<l-c> (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
       -(possible-line-label ^line <line> ^junction <> <junction> ^label +)
-->
        (remove <line>)
(remove <l-c>))


;If a line is labelled '-' on one end, than it must be labelled '-' on the
;other end.
 (P  consistent-minus
(stage reduce-candidates)
 {<line>(possible-line-label ^line <line> ^junction <junction>
                             ^label - ^candidate <c>) }
 {<l-c> (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
       -(possible-line-label ^line <line> ^junction <> <junction> ^label -)
-->
        (remove <line>)
(remove <l-c>))

;If a line is labelled 'in' on one end, than it must be labelled 'out' on the
;other end.
 (P  consistent-in-out
 (stage reduce-candidates)
 {<line> (possible-line-label ^line <line> ^junction <junction>
```

```
                              ^label in ^candidate <c>) }
 {<l-c>  (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
        -(possible-line-label ^line <line> ^junction <> <junction> ^label out)
-->
        (remove <line>)
(remove <l-c>))


;If a line is labelled 'out' on one end, than it must be labelled 'in' on the
;other end.
 (P  consistent-out-in
 (stage reduce-candidates)
 {<line> (possible-line-label ^line <line> ^junction <junction>
                              ^label out ^candidate <c>) }
 {<l-c>  (labelling-candidate ^l-c-ID <c>) } ;find candidate which label belongs to.
        -(possible-line-label ^line <line> ^junction <> <junction> ^label in)
-->
        (remove <line>)
(remove <l-c>))

;When a labelling-candidate is deleted, we want to also delete all possible line
;labels associated with that labelling-candidate.

 (P eliminate-line-labels
           (stage reduce-candidates)
    {<old>  (possible-line-label ^candidate <c>) }
          -(labelling-candidate ^l-c-ID <c>)
          -->
           (remove <old>))
```

APPENDIX B

The Travelling Salesperson Problem

```
:TSP
;Travelling salesperson problem modified to incorporate the minimum
;spanning tree heuristic.
(literalize go )
(literalize home-city name)
(literalize start tag start-city  length city-list)
(vector-attribute city-list)
(literalize connect-goal tag est-cost city1 city2 length city-list)
(literalize so-far tag distance cities-seen)
(vector-attribute cities-seen)
(literalize solution distance tag cities-seen)
;Note the use of the UNIQUE-ATTRIBUTE for the solution element.
(unique-attribute solution)
(literalize solution-goal distance tag cities-seen)
(literalize distance city1 city2 distance)

;Working memory element declarations for minimum spanning tree calculation
(literalize mst-city tag name flag)
(literalize goal-compute-mst cost-so-far seed-city tag unseen-cities)
(vector-attribute unseen-cities)
(literalize mst-data cost-so-far tag)

(vector-attribute city-list)


(literalize initialized value)
;8/27/91 Modified start-city to use new map-vector rhs-macro.  This will
;cut down on instantiations of start-city and create new nodes to "open"
;in quick succession.
(p start-city
  (meta (priority 0))
  {<start> (start ^start-city <sc> ^length <length> )   }
           (initialized ^value t)
   -->
  (oremove 2)
  (map-vector <start> city-list (item <city> vector-less-item <vli>)
       (bind <tag>  (ngenatom))
```

```
     (in-parallel
       (make connect-goal ^tag <tag> ^city1 <sc> ^city2 <city>
                          ^length (compute <length> - 1)
                          ^city-list  <vli>)
       (make so-far ^tag <tag> ^distance 0 ^cities-seen <sc>)))
   )

(p finish-trip
   (meta (priority 0) (control-fn compare-new-solution-with-solution)
         (control-generator
           ((gen-control-data *tsp-distance*
                              (ari (<d-so-far> + <d> + <d2>))))))
   {<sofar> (so-far ^tag <tag> ^distance <d-so-far> ^cities-seen <start-city> ) }
   {<cg> (connect-goal ^tag <tag> ^city1 <city1> ^city2 <city2> ^length 0 ) }
   (distance ^city1 <city1> ^city2 <city2> ^distance <d>)
   (distance ^city1 <city2> ^city2 <start-city> ^distance <d2> )
  -->
   (bind <cities-seen> (litval cities-seen))
   (make solution-goal  ^distance (compute <d-so-far> + <d> + <d2> )
           ^cities-seen (substr <sofar> <cities-seen> inf) <city2> <start-city>))

;Use action parallelism to reduce the run time of rules which do a
;lot of processing in their righthand sides.
(p propagate-city-5
   (meta (priority 1) (control-fn compare-with-solution)
         (priority-queue t)
         (lock-not-required t)
         (priority-fn propagate-city-priority-fn)
         )
   {<sofar> (so-far ^tag <tag> ^distance <d-so-far> ) }
   {<cg> (connect-goal ^tag <tag> ;^est-cost {<> nil <e-cost>}
                       ^city1 <city1> ^city2 <city2> ^length { = 5 <l> } ) }
   (home-city ^name <home>)
   (distance ^city1 <city1> ^city2 <city2> ^distance <d>)
-->
   (bind <cities-seen> (litval cities-seen))
   (map-vector <cg> city-list ( item <new-city> vector-less-item <vli>)
      (bind <newtag> (ngenatom))
     (in-parallel
      (make connect-goal ^tag <newtag> ^city1 <city2> ^city2 <new-city>
           ^length (compute <l> - 1)
           ^city-list  <vli> )
      (make so-far ^tag <newtag> ^distance (compute <d-so-far> +  <d>)
             ^cities-seen (substr <sofar> <cities-seen> inf) ;previously visited
                      <city2> ) ;and the new city
   )))

;No action parallelism in righthand sides as by the time these rules are
;invoked, full rule parallelism will be in use.
(p propagate-city-lt-5
   (meta (priority 1) (control-fn compare-with-solution)
```

```
        (priority-queue t)
        (lock-not-required t)
        (priority-fn propagate-city-priority-fn)
        ;make control generator an externally compiled function...
        )
  {<sofar> (so-far ^tag <tag> ^distance <d-so-far> ) }
  {<cg> (connect-goal ^tag <tag> ;^est-cost {<> nil <e-cost>}
                     ^city1 <city1> ^city2 <city2> ^length { > 0 < 5 <l> } ) }
  (home-city ^name <home>)
  (distance ^city1 <city1> ^city2 <city2> ^distance <d>)
-->
  (bind <cities-seen> (litval cities-seen))
  (map-vector <cg> city-list ( item <new-city> vector-less-item <vli>)
     (bind <newtag> (ngenatom))
     (make connect-goal ^tag <newtag> ^city1 <city2> ^city2 <new-city>
          ^length (compute <l> - 1)
          ^city-list  <vli> )
     (make so-far ^tag <newtag> ^distance (compute <d-so-far> +  <d>)
           ^cities-seen (substr <sofar> <cities-seen> inf) ;previously visited
                    <city2> ) ;and the new city
  ))



(p init-distance-table
  (start)
 [(distance ^city1 <c1> ^city2 <c2> ^distance <d>)]
 -->
  (map-set
    (add-to-distance-table ($varbind '<c1>) ($varbind '<c2>)
                          ($varbind '<d>)))
  (make initialized ^value t))


 (p start
  (go)
 -->
 (in-parallel-sync
  (make start ^start-city NY ^length 6
        ^city-list SEATTLE HTFD SF  CHI PHOENIX BOSTON)
  (make home-city ^name NY)
  (make distance ^city1 NY  ^city2 SF ^distance 3000)
  (make distance ^city1 NY  ^city2 HTFD ^distance 80 )
  (make distance ^city1 NY  ^city2 SEATTLE  ^distance 3500)
  (make distance ^city1 NY  ^city2 CHI ^distance 1500)
  (make distance ^city1 NY  ^city2  PHOENIX ^distance 2300)
  (make distance ^city1 NY  ^city2  BOSTON  ^distance 190)

  (make distance ^city1 SF  ^city2 HTFD ^distance 3040)
  (make distance ^city1 SF  ^city2 SEATTLE ^distance 450)
  (make distance ^city1 SF  ^city2 CHI ^distance 2000)
```

```
  (make distance ^city1 SF  ^city2 NY ^distance 3000)
  (make distance ^city1 SF  ^city2  PHOENIX ^distance 1850)
  (make distance ^city1 SF  ^city2  BOSTON  ^distance 2900)


  (make distance ^city1 SEATTLE ^city2  SF ^distance 450)
  (make distance ^city1 SEATTLE ^city2  HTFD ^distance 3600)
  (make distance ^city1 SEATTLE ^city2  NY ^distance 3500)
  (make distance ^city1 SEATTLE ^city2  CHI ^distance 2200)
  (make distance ^city1 SEATTLE ^city2  PHOENIX ^distance 2300)
  (make distance ^city1 SEATTLE  ^city2  BOSTON  ^distance 3400)

  (make distance ^city1 CHI  ^city2  NY ^distance 1500)
  (make distance ^city1 CHI  ^city2  SF ^distance 2000)
  (make distance ^city1 CHI  ^city2  HTFD ^distance 1450)
  (make distance ^city1 CHI  ^city2  SEATTLE  ^distance 2200)
  (make distance ^city1 CHI  ^city2  PHOENIX ^distance 1400)
  (make distance ^city1 CHI  ^city2  BOSTON  ^distance 1610)


  (make distance ^city1 HTFD  ^city2  NY ^distance 80)
  (make distance ^city1 HTFD  ^city2  SF ^distance 3040)
  (make distance ^city1 HTFD  ^city2  SEATTLE ^distance 3600)
  (make distance ^city1 HTFD  ^city2  CHI ^distance 1450)
  (make distance ^city1 HTFD  ^city2  PHOENIX ^distance 2350)
  (make distance ^city1 HTFD  ^city2  BOSTON  ^distance 110)


  (make distance ^city1 PHOENIX  ^city2  NY ^distance 2750)
  (make distance ^city1 PHOENIX  ^city2  HTFD ^distance 2700)
  (make distance ^city1 PHOENIX  ^city2  SF ^distance 1000)
  (make distance ^city1 PHOENIX  ^city2  SEATTLE ^distance 2300)
  (make distance ^city1 PHOENIX  ^city2  CHI ^distance 1800)
  (make distance ^city1 PHOENIX  ^city2  BOSTON  ^distance 2600)


  (make distance ^city1 BOSTON ^city2 NY ^distance 190)
  (make distance ^city1 BOSTON ^city2 SF ^distance 2900)
  (make distance ^city1 BOSTON ^city2 SEATTLE ^distance 3400)
  (make distance ^city1 BOSTON ^city2 CHI ^distance 1610)
  (make distance ^city1 BOSTON ^city2 HTFD ^distance 110)
  (make distance ^city1 BOSTON ^city2 PHOENIX ^distance 2350)

 )
  (init-distance-table 5)
 )

;If better solution found, propagate it.
;Note:  Without locking mechanism, this rule has two error modes,
;depending on when remove is performed.  If the remove comes
;second, then there will be two solutions for a brief period.
```

```
;Another rule might fire on the old solution value, and create
;a superfluous solution (or, in some systems, might overwrite the
;new, better solution).  If the remove comes first, then there
;will be a brief period in which no solution exists, in which
;case an init production referencing -(solution) might fire.

(p init-solution
   (meta (priority 0))
   {<new> (solution-goal ^distance <dist> ^tag <tag> ) }
         -(solution-goal ^distance < <dist>)
         -(solution)
  -->
 (bind  <cities-seen> (litval cities-seen))
  (make-unique solution ^tag <tag> ^distance <dist>
        ^cities-seen (substr <new> <cities-seen> inf)))

;Now here's an application for functionally accurate programming.
;Note that this rule may fire, even though, while it's firing,
;a solution-goal whose distance is < <dist> may appear, in
;effect disabling this rule.   But, because solution is locked,
;competing rule won't fire until new solution is postulated, so
;no harm is done, and correct solution eventually becomes asserted.

(p new-and-improved
  (meta (priority 0))
  {<new> (solution-goal ^distance <dist> ^tag <tag> ) }
        -(solution-goal ^distance < <dist>)
  {<old> (solution ^distance > <dist> ^tag <oldtag>)  }
 -->
  (bind  <cities-seen> (litval cities-seen))
  (make solution ^tag <tag> ^distance <dist>
        ^cities-seen (substr <new> <cities-seen> inf))
  (remove <old>)
  )
```

```
;SALES-CONTROL.LISP
;These are the lisp functions used to implement heuristic control
;in the travelling salesperson program.


(defvar *solution-so-far* nil "The solution generated so far")
;A list of variables to be reset before each run.
(defvar *control-variables* '(*solution-so-far*))

;A pruning function which determines if a new solution is better than
;the current solution.  If so, records the new value and returns t.

(defun  compare-new-solution-with-solution(instance)
  (declare (optimize (speed 3) (safety 0) (space 0)))
  "Compare-with-solution(control-data): Control-data is an a-list derived from
   the rule-instance-control-data slot.   If the control function returns a
   non-nil value, the rule should be executed, otherwise it should be pruned."
  (let ((new-solution (cdr (assoc '*tsp-distance*
                                  (rule-instance-control-data instance)))))
     (cond ((not *solution-so-far*)
             (setf *solution-so-far* new-solution))
            ((< new-solution *solution-so-far*)
             (setf *solution-so-far* new-solution))
            (t
             nil ;indicating bad rule -- don't execute
             ))))

;This function is used to compare a developing solution with a complete solution.
;If the developing solution ever exceeds the current best, then return nil.
;If no current best solution, return t.
(defun compare-with-solution(instance)
  (declare (optimize (speed 3) (safety 0) (space 0)))
  "Compare-with-solution(control-data): Control-data is an a-list derived from
   the rule-instance-control-data slot.   If the control function returns a
   non-nil value, the rule should be executed, otherwise it should be pruned."
  (if *solution-so-far*
       (< (rule-instance-rating instance) ;distance travelled < best-solution
           *solution-so-far*)
      t))


;Priority computation for propagate city.  Extracts relevant data from
;vector representation.

(defun propagate-city-priority-fn()
  (+ ($varbind '<d-so-far>)
     (compute-mst (list ($varbind '<city2>))
                  (cons ($varbind '<home>)
                        (substr-to-list '<cg> 'city-list 'inf)))))

;Minimum Spanning Tree Computation:
```

```
(defvar *distance-table* nil)
(defun init-distance-table(n-cities)
     (setf *distance-table*
           (new-dhash-table n-cities)))

(defun add-to-distance-table(city1 city2 distance)
    (let ((tmp (get-dhash city1 *distance-table*)))
      (set-dhash city1
                 (push (cons city2 distance)
                       tmp)
                 *distance-table*)))

(defun do-mst-fn(city-list)
    (compute-mst (list (car city-list))
                 (cdr city-list)))

(defun compute-mst(seen-cities not-seen-cities)
    (let ((min-so-far 0)
          (min-city nil)
          (dist 0))
      (while not-seen-cities
        (setf min-city (car not-seen-cities))
        (setf min-so-far (apply #'min
                                (mapcar #'(lambda(city1)
                                            (city-distance city1 min-city))
                                        seen-cities)))
        (mapc #'(lambda(seen)
                   (mapc #'(lambda(not-seen)
                             (setf dist (city-distance seen not-seen))
                             (cond ((< dist min-so-far)
                                    (setf min-so-far dist)
                                    (setf min-city not-seen))))
                         (cdr not-seen-cities)))
              seen-cities)
        (push min-city seen-cities)
        (setf not-seen-cities (delete min-city not-seen-cities)))
      min-so-far))

(defun city-distance(city1 city2)
     (cdr (assoc city2
                 (get-dhash city1 *distance-table*)
                 :test #'eq)))
```

BIBLIOGRAPHY

[Acharya and Tambe, 1989] Acharya, Anurag and Tambe, Milind. Production systems on message passing computers: Simulation results and analysis. In *1989 International Conference on Parallel Processing*, pages II 246–256, August 1989.

[Acharya *et al.*, 1991] Acharya, Anurag, Tambe, Milind, and Gupta, Anoop. Implementations of production systems on message passing computers, 1991. To appear in IEEE Transactions on Parallel and Distributed Computing.

[Blelloch, 1986] Blelloch, G. E. CIS: A massively concurrent rule-based system. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 735–741, August 1986.

[Boulanger, 1988] Boulanger, Albert. The modification of a rule-based diagnostic system for routinized parallelism on the butterfly Technical Report 6713, BBN Laboratories, Inc., February 1988.

[Brownston *et al.*, 1985] Brownston, Lee, Farrell, Robert, Kant, Elaine, and Martin, Nancy. *Programming Expert Systems in OPS5 – An Introduction to Rule-based Programming.* Addison-Wesley Publishing Company, 1985.

[Clocksin and Mellish, 1981] Clocksin, W. F. and Mellish, C. S. *Programming in Prolog.* Springer-Verlag, Berlin; New York, 1981.

[Conery, 1987] Conery, John S. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell, MA, 1987.

[Corkill, 1989] Corkill, Daniel D. Design alternatives for parallel and distributed blackboard systems. In Jagannathan, V., Dodhiawala, Rajendra, and Baum, Lawrence S., editors, *Blackboard Architectures and Applications*, pages 99–136. Academic Press, 1989.

[Davis, 1980] Davis, Randall. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.

[Decker *et al.*, 1991] Decker, Keith S., Garvey, Alan J., Humphrey, Marty A., and Lesser, Victor R. Effects of parallelism on blackboard system scheduling. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia, August 1991.

[Delcambre and Etheredge, 1988] Delcambre, Lois M. L. and Etheredge, James N. A self-controlling interpreter for the relational production language. In *SIGMOD: International Conference on Management of Data*, pages 396–403, June 1988.

[Delcher and Kasif, 1989] Delcher, Arthur and Kasif, Simon. Some results on the complexity of exploiting data dependency in parallel logic programs. *The Journal of Logic Programming*, 6(3), May 1989.

[Feigenbaum and McCorduck, 1983] Feigenbaum, Edward A. and McCorduck, Pamela. *The Fifth Generation : artificial intelligence and Japan's computer challenge to the world.* Addison-Wesley, Reading, Mass, 1983.

[Fennell and Lesser, 1977] Fennell, R. D. and Lesser, V. R. Parallelism in AI problem solving: A case study of Hearsay-II. *IEEE Transactions on Computers*, C-26(2):198–111, February 1977.

[Forgy, 1979] Forgy, C. L. *On the Efficient Implementation of Production Systems.* PhD thesis, Carnegie-Mellon University, 1979.

[Forgy, 1980] Forgy, C. L. Note on production systems and Illiac-IV. Technical Report CMU-CS-80-130, CMU Computer Science Department, July 1980.

[Forgy, 1981] Forgy, C. L. OPS5 user's manual. Technical Report CMU-CS-81-135, CMU Computer Science Department, July 1981.

[Forgy, 1982] Forgy, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Forgy, 1984] Forgy, C. L. The OPS83 report. Technical Report CMU-CS-84-133, CMU Computer Science Department, May 1984.

[Friedman, 1985] Friedman, Leonard. Controlling production firing: The FCL language. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 359–366, 1985.

[Fujimoto, 1990] Fujimoto, Richard. Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53, 1990.

[Gamble, 1990] Gamble, Roseanne Fulcomer. A methodology for developing correct rule-based programs for parallel implementation. Thesis proposal, Washington University, Sever Institute of Technology, 1990.

[Georgeff, 1982] Georgeff, M. P. Procedural control in production systems. *Artificial Intelligence*, 18:175–201, 1982.

[Gordin and Pasik, 1991] Gordin, Douglas N. and Pasik, Alexander J. Set-oriented constructs: From Rete rule bases to database systems. In *Proceedings 10th ACM Symposium on PODS*, pages 60–67, 1991.

[Gupta *et al.*, 1988] Gupta, A., Tambe, M., Kalp, D., Forgy, C., and Newell, A. Parallel implementation of OPS5 on the encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2), 1988.

[Gupta, 1984] Gupta, Anoop. Implementing OPS5 production systems on DADO. Technical Report CMU-CS-84-115, CMU Computer Science Department, 1984.

[Gupta, 1987] Gupta, Anoop. *Parallelism in Production Systems.* Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Harvey *et al.*, 1991] Harvey, Wilson, Kalp, Dirk, Tambe, Milind, McKeown, David, and Newell, Allen. The effectiveness of task-level parallelism for production systems. *Journal of Parallel and Distributed Computing*, 13(4):395–411, December 1991.

[Hayes-Roth, 1985] Hayes-Roth, Barbara. A blackboard architecture for control. *Artificial Intelligence*, 26:251–321, 1985.

[Hillyer and Shaw, 1988] Hillyer, B. K. and Shaw, D. E. Execution of OPS5 production systems on a massively parallel machine. *Journal of Parallel and Distributed Processing*, August 1988.

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

[Ishida *et al.*, 1990] Ishida, Toru, Yokoo, Makoto, and Gasser, Les. An organizational approach to adaptive production systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 52–57, 1990.

[Ishida, 1990] Ishida, Toru. Methods and effectiveness of parallel rule firing. In *Sixth IEEE Conference on Artificial Intelligence Applications*, March 1990.

[Ishida, 1991] Ishida, Toru. Parallel rule firing in production systems. *IEEE Transactions on Knowledge and Data Engineering*, March 1991.

[Kalp *et al.*, 1988] Kalp, Dirk, Tambe, Milind, Gupta, Anoop, Forgy, Charles, Newell, Allen, Acharya, Anurag, Milnes, Brian, and Swedlow, Kathy. Parallel OPS5 user's manual. Technical Report CMU-CS-88-187, CMU Computer Science Department, November 1988.

[Kelly and Seviora, 1989] Kelly, Michael and Seviora, Rudolph. An evaluation of DRete on CUPID for OPS5 matching. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 84–90, 1989.

[Kleinrock, 1975] Kleinrock, Leonard. *Queueing Systems, Volume I: Thoery.* John Wiley and Sons, 1975.

[Krall and McGehearty, 1986] Krall, Edward J. and McGehearty, Patrick F. A case study of parallel execution of a rule-based system. *International Journal of Parallel Processing*, 15(1):5–32, 1986.

[Kumar *et al.*, 1988] Kumar, Vipin, Ramesh, K., and Rao, V. Nageshwara. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 122–127, 1988.

[Kuo and Moldovan, 1991] Kuo, Steve and Moldovan, Dan. Implementation of multiple rule firing production systems on hypercube. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 305–309, 1991.

[Kuo *et al.*, 1991] Kuo, Chin-Ming, Miranker, Daniel, and Browne, James C. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13(4):424–441, December 1991.

[Lesser and Corkill, 1981] Lesser, Victor R. and Corkill, Daniel D. Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):81–96, January 1981.

[Lin and Kumar, 1991] Lin, Yow-Jian and Kumar, Vipin. And-parallel execution of logic programs on a shared-memory multiprocessor. *The Journal of Logic Programming*, 1991.

[Maruyama *et al.*, 1985] Maruyama, T., Hirata, K., Tanaka, H., and Moto-oka, T. A note on the elementary execution unit in a parallel inference machine. In *Logic Programming '85 – Proceedings of the 4th Conference*, pages 25–33, July 1985.

[McCracken, 1981] McCracken, Donald L. *A Production System Version of the Hearsay-II Speech Understanding System.* UMI Research Press, Ann Arbor, Michigan, 1981.

[McDermott and Forgy, 1978] McDermott, J. and Forgy, C. Production system conflict resolution strategies. In Waterman, D. A. and Hayes-Roth, Frederick, editors, *Pattern-Directed Inference Systems*, pages 177–199. Academic Press, New York, New York, 1978.

[McDermott, 1980] McDermott, J. R1: A rule-based configurer of computer systems. Technical Report CMU-CS-80-119, CMU Computer Science Department, 1980.

[Miranker *et al.*, 1989] Miranker, Daniel, Kuo, Chin-Ming, and Browne, James C. Parallelizing transformations for a concurrent rule execution language. Technical Report TR-89-30, Department of Computer Science, University of Texas at Austin, October 1989.

[Miranker, 1990a] Miranker, Daniel P. An algorithmic basis for integrating production systems and large databases. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, CA, February 1990.

[Miranker, 1990b] Miranker, Daniel P. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[Moldovan, 1986] Moldovan, Dan I. A model for parallel processing of production systems. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 568–573, 1986.

[Moldovan, 1989] Moldovan, D. I. Rubic: a multiprocessor for rule-based systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4):699–706, July/August 1989.

[Morgan, 1988] Morgan, K. BLITZ: A rule-based system for massively parallel architectures. In *Proceedings 1988 ACM Conference for Lisp and Functional Programming, Snowbird, Utah*, 1988.

[Neiman, 1991] Neiman, Daniel. Control issues in parallel rule-firing production systems. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 310–316, 1991.

[Neiman, 1992a] Neiman, Daniel. Parallel OPS5 user's manual and technical report. COINS Technical Report 92–28 (Supersedes TR 91-1), Computer and Information Sciences Dept., University of Massachusetts, April 1992.

[Neiman, 1992b] Neiman, Daniel E. A multiple worlds implementation for parallel rule-firing production systems, August 1992. Technical Report in preparation.

[Nii *et al.*, 1989] Nii, H. Penny, Nelleke, Aiello, and Rice, James. Experiments on cage and poligon: Measuring the performance of parallel blackboard systems. In Gasser, Les and Huhns, Michael N., editors, *Distributed Artificial Intelligence, Vol. II*, pages 319–384. Morgan Kaufmann Publishers, Inc., 1989.

[Oflazer, 1984] Oflazer, Kemal. Partitioning in parallel processing of production systems. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1984.

[Pasik and Stolfo, 1987] Pasik, A. and Stolfo, S. Improving production system performance on parallel architectures by creating constrained copies of rules. Technical report, Computer Science Dept., Columbia University, 1987.

[Pearl, 1984] Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison–Wesley, Reading, Massachusetts, 1984.

[Perlin, 1989] Perlin, Mark W. The match box algorithm for parallel production system match. Technical Report CMU-CS-89-163, Computer Science Dept., Carnegie-Mellon University, May 1989.

[Schmolze and Goel, 1990] Schmolze, James G. and Goel, S. A parallel asynchronous distributed production system. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 65–71, 1990.

[Schmolze and Neiman, 1992] Schmolze, James G. and Neiman, Daniel E. Comparison of three algorithms for ensuring serializability in parallel production systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-92)*, July 1992.

[Schmolze, 1989] Schmolze, James G. Guaranteeing serializable results in synchronous parallel production systems. Technical Report 89-5, Department of Computer Science, Tufts University, October 1989.

[Schmolze, 1991] Schmolze, James G. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13(4), December 1991.

[Schor *et al.*, 1986] Schor, Marshall I., Daly, Timothy P., Lee, Ho Soo, and Tibbits, Beth R. Advances in Rete pattern matching. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 226–232, 1986.

[Sellis *et al.*, 1987] Sellis, Timos, Lin, Chih-Chen, and Raschid, Louiqa. Implementing large production systems in a dbms environment: Concepts and algorithms. Technical Report CS-TR-1960, Dept. of Computer Science, University of Maryland at College Park, 1987.

[Sherman and Martin, 1990] Sherman, Porter D. and Martin, John C. *An OPS5 Primer: Introduction to Rule-based Expert Systems.* Prentice Hall, Eaglewood Cliffs, New Jersey, 1990.

[Siler *et al.*, 1987] Siler, William, Tucker, Douglas, and Buckley, James. A parallel rule firing fuzzy production system with resolution of memory conflicts by weak fuzzy monotonicity, applied to the classification of multiple objects characterized by multiple uncertain features. *International Journal of Man-Machine Studies*, 26:321–332, 1987.

[Srivastava and Wang, 1991] Srivastava, J. and Wang, J.-H. A transaction model for parallel production systems. Technical Report AHPCRC TR 91–17, University of Minnesota, 1991.

[Stolfo and Miranker, 1984] Stolfo, Salvatore and Miranker, Daniel. Dado: A parallel processor for expert systems. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 74–82, 1984.

[Stolfo *et al.*, 1990] Stolfo, Salvatore J., Woodbury, Leland, Glazier, Jason, and Chan, Philip. The ALEXSYS mortgage pool allocation expert system: A case study of speeding up rule-based programs. In *AI and Business Workshop, AAAI-90*, 1990.

[Stolfo *et al.*, 1991a] Stolfo, Salvatore J., Dewan, Hasanat M., and Wolfson, Ouri. The PARULEL parallel rule language. In *1991 International Conference on Parallel Processing*, pages II–36–45, 1991.

[Stolfo *et al.*, 1991b] Stolfo, Salvatore J., Wolfson, Ouri, Chan, Philip K., Dewan, Hasanat M., Woodbury, Leland, Glazier, Jason S., and Ohsie, David A. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4):366–382, December 1991.

[Stolfo, 1984] Stolfo, Salvatore. Five parallel algorithms for production system execution on the dado machine. In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 300–307, 1984.

[Stolfo, 1987] Stolfo, Salvatore J. Initial performance of the DADO2 prototype. *Computer*, pages 75–82, January 1987.

[Stonebraker *et al.*, 1986] Stonebraker, M., Sellis, T., and Hanson, E. An analysis of rule indexing implementations in data base systems. In Kershberg, L., editor, *Expert Database Systems: Proceedings of the First International Workshop*, pages 353–363. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1986.

[Tambe *et al.*, 1988] Tambe, M., Kalp, D., Gupta, A., Forgy, C., Milnes, B., and Newell, A. Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of Parallel Programming Environments, Application Languages, and Systems(PPEALS)*, July 1988.

[Tambe *et al.*, 1989] Tambe, Milind, Acharya, Anurag, and Gupta, Anoop. Implementation of production systems on message passing computers: Techniques, simulation results, and analysis. Technical Report CMU-CS-891-29, School of Computer Science, Carnegie-Mellon University, 1989.

[Tenorio and Moldovan, 1985] Tenorio, M. and Moldovan, D. Mapping production systems into multiprocessors. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 56–62, 1985.

[Uhr, 1979] Uhr, Leonard M. Parallel-serial production systems. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 911–916, 1979.

[van Biema *et al.*, 1986] Biema, M.van, Miranker, D. P., and Stolfo, S. J. The do-loop considered harmful in production system programming. In *First International Conference on Expert Database Systems*, pages 88–97, 1986.

[Waltz, 1975] Waltz, David. Understanding line drawings of scenes with shadows. In Winston, Patrick H., editor, *The Psychology of Computer Vision.* McGraw-Hill Book Company, New York, 1975.

[Widom and Finkelstein, 1990] Widom, J. and Finkelstein, S.J. Set-oriented production rules in relational database systems. In *ACM-SIGMOD International Conference on the Management of Data*, pages 259–270, 1990.

[Wogrin and Cooper, 1988] Wogrin, Nancy and Cooper, Thomas. *Rule-based Programming in OPS5*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[Wolfson and Ozeri, 1990] Wolfson, Ouri and Ozeri, Aya. A new paradigm for parallel and distributed rule-processing. In *Proceedings of the ACM-SIGMOD 1990 International Conference on Management of Data*, pages 133–142, May 1990.

[Wolfson *et al.*, 1990] Wolfson, Ouri, Dewan, Hasanat, Stolfo, Salvatore, and Yemini, Yechiam. Incremental evaluation of rules and its relationship to parallelism. Technical Report CUCS-058-90, Department of Computer Science, Columbia University, New York, New York, 1990.

[Xu and Hwang, 1991] Xu, Jian and Hwang, Kai. Mapping rule-based systems onto multicomputers using simulated annealing. *Journal of Parallel and Distributed Computing*, 13(4):442–455, December 1991.