DECOMPOSITION ABSTRACTION IN PARALLEL RULE LANGUAGES

SHIOW-YANG WU

Department of Computer Sciences

The University of Texas at Austin

Austin, TX 78712-1188

TR-95-33 August 1995

DECOMPOSITION ABSTRACTION IN PARALLEL RULE LANGUAGES

APPROVED BY
DISSERTATION COMMITTEE:

_____

_____

_____

_____

_____

DECOMPOSITION ABSTRACTION IN PARALLEL RULE LANGUAGES

by

SHIOW-YANG WU, M.S., B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
August, 1995

Acknowledgments

Even though this work is about the specification of semantics, I have to admit that some things can not be described. There is no words to express my gratitude to my advisors Professors James C. Browne and Daniel P. Miranker. Their continuous encouragement and guidance have been the greatest support that helps me get through all these years. This dissertation could not have been completed without their insight and invaluable comments. I would also like to thank all the other members of my committee, Professors Al Mok, Martin D. F. Wong, and Vijay Garg, for their suggestions and helpful comments.

Special thanks to Professor Salvatore J. Stolfo at the Columbia University, for his in-depth discussion and exchange of ideas related to this work, to Lance Obermeyer at Applied Research Lab, for his help during my implementation of the Venus/DA language system, and to many colleague for their friendships and encouragement during my stay at the University of Texas.

My deepest appreciation goes to my parents, for their devotion and everlasting support, to my sons Alexander and Douglas, for the surprises and joyfulness they bring me, and most importantly, to my wife, Hsin-yi, for her immeasurable patience and endless love.

SHIOW-YANG WU

The University of Texas at Austin
August, 1995

DECOMPOSITION ABSTRACTION IN PARALLEL RULE LANGUAGES

Publication No. _____

SHIOW-YANG WU, Ph.D.
The University of Texas at Austin, 1995

Supervisors: James C. Browne and Daniel P. Miranker

As the applications of production systems expand from traditional artificial intelligence domains into the data intensive and real-time arenas, program complexity and the volume of data also increase dramatically. Over a decade of efforts to exploit this opportunity, the previous approaches of employing *parallel match* and/or syntactic based *multiple-rule-firing* have failed to raise the performance to a satisfactory level. Based on the observations made in a pilot study, we found that by incorporating *application semantics*, it is possible to achieve a much higher level of concurrency than what can be achieved by traditional techniques. This dissertation presents a new approach called *decomposition abstraction* that aims at the exploration of *application parallelism* in production systems.

Decomposition abstraction is the process of organizing and specifying parallel decomposition strategies. We propose a general object-based framework and present the formal semantics of a set of *decomposition abstraction mechanisms* that are applicable to any rule language. A semantic-based dependency analysis technique that uncovers hidden concurrency based on a new notion of *functional dependency* successfully derives parallelism that is very difficult, if not impossible, to discover by traditional syntactic analysis techniques.

The effectiveness of our approach is validated both by simulation and implementation on Sequent Symmetry multiprocessor. The performance results demonstrate the potential of the decomposition abstraction approach to achieve linear and scalable speedup.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Production systems, also known as rule-based systems or simply rule systems, have been shown to be a powerful architecture for intelligent systems, especially expert systems such as Prospector [37], R1 [94], and MYCIN [20]. Initial implementations of production systems suffered from poor performance which prohibited their use in large scale applications [45]. Nevertheless, applications of rule-based programming have continued to expand. Recent interest in data intensive rule-based applications [121] has further fueled the need for high performance execution environments for production systems.

Intuition suggests that languages based on the production system model admit a high degree of parallelism [55]. Efforts to exploit parallel processing to increase production system performance have been ongoing for over a decade [85, 98, 136]. However, the maximum speedup achieved by actual implementation rarely exceeds tenfold and has never done so over a general suite of applications no matter how many processors are used.

Most of the existing techniques for parallel production systems are, from a methodology point of view, similar to the techniques used in the parallelization of sequential imperative languages (mostly FORTRAN) [9, 115, 116, 162, 167]. Critical part(s) of the sequential execution is(are) parallelized, or optimizing compilation and transformations are applied to automatically transform a sequential program into a parallel program. This approach has the obvious benefit of its general applicability to existing sequential programs. However, the experiences show that these techniques have met with limited success, both on imperative languages [5] and rule languages [56, 98].

In this research, we took an unusual approach to the problem of parallelizing production systems. We promote the change of direction toward *semantic-based parallelism*. We believe that to significantly improve the performance of rule-based programs, programmers should share part of the responsibility for exposing parallelism. This is achieved by enable the programmers to provide *semantic information*, in the forms of *data* and *function decompositions*, to the language systems. In other words, we suggest the design of

parallel rule languages and the development of techniques for *parallel rule-based programming*. The challenges are:

- to provide proper mechanisms for expressing application semantics without asking programmers to be an experts in parallel programming, and
- to effectively exploit the semantic information supplied by the programmers.

This chapter highlights our approach, contributions, and research results.

1.1 Production System Paradigm

We review the structure and operation of production systems. This serves both as an introduction to the terminology used throughout this thesis and as a characterization of essential features of production system that must be accounted for when we develop our framework. More information about production systems in general and about OPS5, a popular sequential rule language, in particular can be found in [19, 26].

As depicted in Figure 1.1, a typical production system is composed of three components: a data store called *working memory*, a set of *rules*, and an *inference engine*. Working memory is a global database composed of data objects called *working memory elements* (WME's) representing the state of the system. A rule is essentially a conditions-actions pair. The inference engine stands for the three-phase cyclic execution model of matching, conflict-resolution and firing, which is also known as the *recognize-act* cycle. In a cycle, the conditions of each rule are matched against the working memory. A rule with a set of WME's matching the conditions is called an *instantiation*. The set of all instantiations constitutes the *conflict set*. In a sequential environment, conflict-resolution aims to select one instantiation from the conflict set for firing. In a parallel environment, multiple rule instantiations can be selected for firing simultaneously subject to proper correctness constraints such as serializability [123]. Firing an instantiation simply means to execute the actions which may add, delete, or modify WME's in the working memory.[1] The cycle repeats until no rule can be fired, i.e. no instantiations are computed by the match.

[1]The actions may include changes to the rules as indicated by the dash arrow. That constitutes the so called *learning production systems* such as Soar [86]. Learning is not in the scope of this thesis.

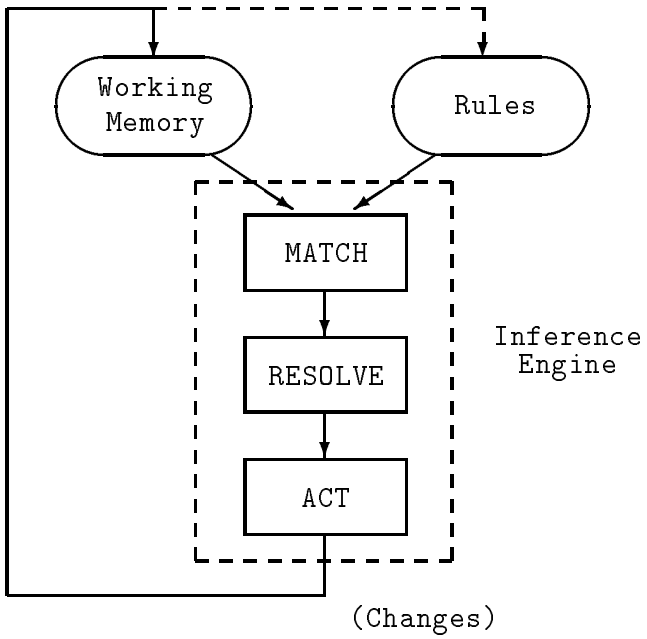

Figure 1.1: Production System Model.

1.2 Motivating Examples

Analysis of the parallelism in production systems [55, 79, 113, 136] has focused on the parallelism in the production system model and syntactic structure of the production system programs. Chapter 2 details the research that explores this type of parallelism, which we will call *syntactic-level parallelism* or *application independent parallelism*. Most of the techniques employed were built upon the parallelization of the production system execution engine. The extraction of parallelism relied only on the syntactic structure of the rule programs. This approach has the advantage that the techniques developed can be applied on any rule programs without further information from the programmer besides the program text. However, limited by similar (perhaps more) obstacles as the automatic parallelization of imperative languages, namely, name ambiguity, non-statically resolvable dependencies, large space of possible transformations, sequential semantics of the languages, highly dynamic and nondeterministic run-time behavior, this approach have met with little success.

We begin this research by analyzing a set of common benchmark problems and the rule-based programs written in OPS5 to realize them for potential parallelism. These programs have been widely used in previous studies

Program	No. Rules	Description
LIFE	16	A simulation program implements Conway's LIFE.
WALTZ	33	A constraint satisfaction problem using Waltz's algorithm for scene labeling [158].
MANNERS	8	A combinatorial search problem for seat assignment.

Table 1.1: Benchmark programs.

to evaluate the effectiveness of language extensions and compilation techniques [80, 84, 103, 123]. While the amount of parallelism found in previous work has typically been modest and not necessarily scalable, we found, contrary to previous expectations, that several of these programs had the potential for massive and scalable parallelism. In this section, we give the results of preliminary simulated parallel execution of three of the benchmark programs and identify the sources of parallelism in the algorithms which leads to our semantic-based approach toward parallel production systems.

The programs are LIFE, WALTZ, and MANNERS as listed in Table 1.1. All results are obtained by going through the following steps:

- First, OPS5 benchmark programs and their sequential execution traces are carefully studied and analyzed to identify the potential parallelism in the problems and the algorithms.
- Then, all programs are reformulated such that inherent parallelism can be effectively exploited.
- Both the results of sequential and parallel executions are collected in terms of number of execution cycles.

The speedup is measured by comparing the number of cycles between sequential and parallel executions. Sequential cycles are obtained by actually running the OPS5 programs using OPS5c [103] on SUN SPARCs and HP 9000 workstations. Parallel cycles are calculated by hand with the assumption of unlimited resources, no overhead and no contention. To see whether our approach scales up, the performance results of increasing problem size are collected for each program.

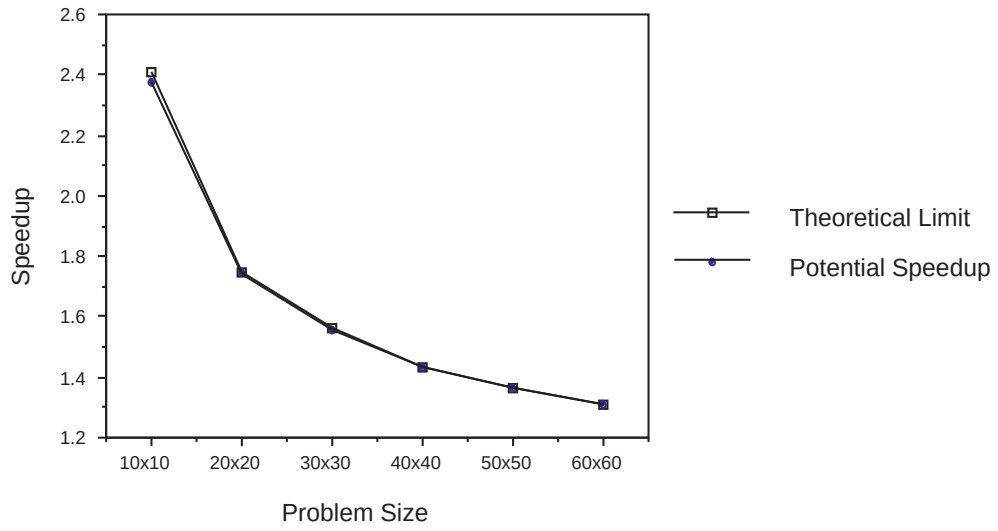

Figure 1.2: LIFE Speedup (with printing).

1.2.1 LIFE

There are three sets of results on the LIFE program. The original program contains a sequential print context to print out intermediate and final results. Since this process is inherently sequential, according to Amdahl's law [2],[2] the speedup is limited by the sequential part. The result on this version of the program is given in Figure 1.2. Because of the limit imposed by the sequential printing, the potential speedup is quite small but close to the theoretical maximum speedup calculated following Amdahl's law. To measure the actual speedup in the computation part, we have obtained the results on two slightly modified versions of the program. The first one is the version without printing intermediate results. This is presented in Figure 1.3 together with the theoretical speedup limits. The second set of results, which is in Figure 1.4, is to measure the computation part alone without any printing.

The key reason for such impressive results resides in the identification of the following sources of inherent parallelism in the LIFE program:

- All live cells can generate neighbors at the same time.

[2] Amdahl's law says that if f is the fraction of a computation that must be performed sequentially, where $0 \leq f \leq 1$, then the maximum speedup S achievable by a parallel computer with p processors is $S \leq 1/(f + (1-f)/p)$.

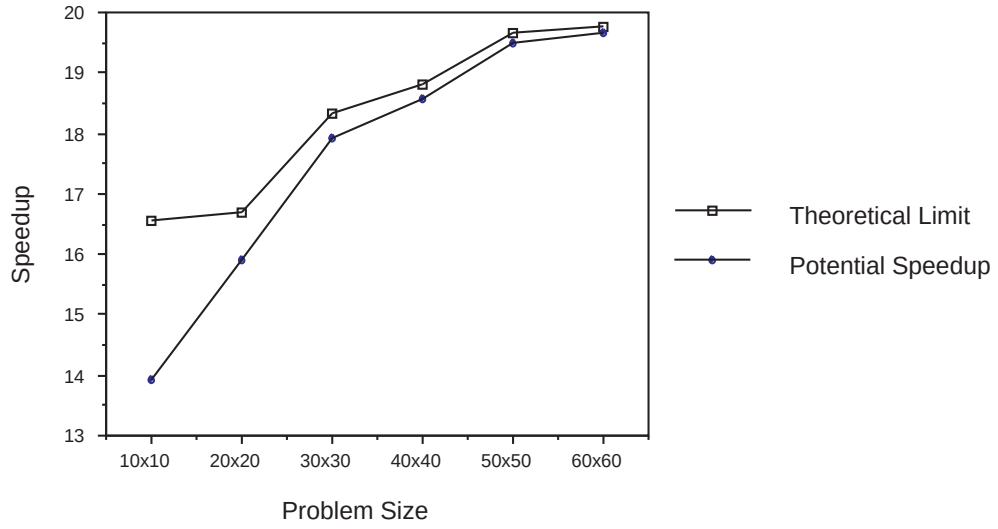

Figure 1.3: LIFE Speedup (without printing intermediate results).

- All cells can compute their neighbors simultaneously.
- The generation update of all cells can be executed in parallel.

The consequence is that it takes a constant number of parallel cycles to do the computation part while sequential cycles increase dramatically with the problem size. These sources of parallelism are quite difficult, sometimes impossible, to detect at compile-time using syntactic-based techniques developed in previous research. This suggests the need for a new approach that could capture the semantic parallelism described above.

1.2.2 WALTZ

Because of the time taken to simulate large data sets, we only performed simulations on small data sets for the WALTZ program. Nevertheless, the results are still quite inspiring as depicted in Figure 1.5.

The potential parallelism in the WALTZ program rests on the important semantic information in the problem and the organization of the data:

- Each line is associated with exactly two edges with opposite end points.
- The type of a junction is unique and each junction is associated with a unique set of edges.

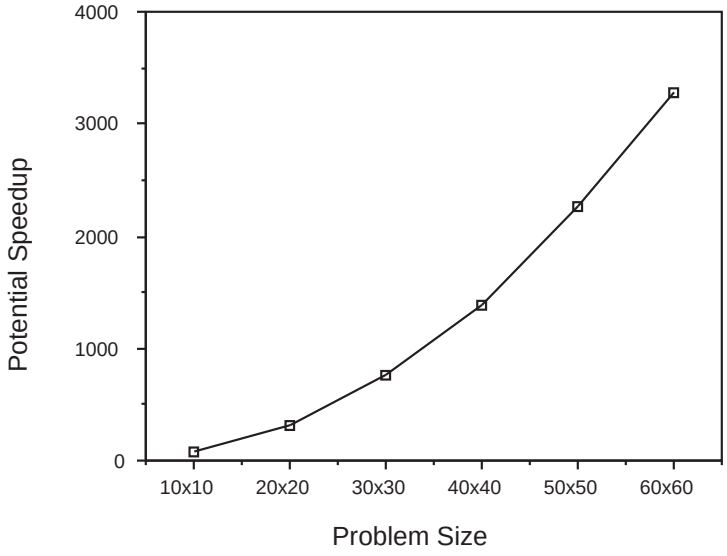

Figure 1.4: LIFE Speedup (without printing).

From this information, we can identify the following additional parallelism in the program:

- All junctions can be made concurrently without interfering with one another.
- Since each labeling rule matches a single junction and its associated edges, all enabled labeling rules with disjoint matching working memory elements can be fired in parallel.

This is clearly scalable parallelism because the larger the problem (in terms of number of line segments in the input drawing), the more junctions and edges a drawing has, which results in more rules being eligible for firing in parallel. Like the case for the LIFE program, the additional parallelism is a characteristic of the problem and is derived from the implicit design decisions of the WALTZ program. Without this level of information available, a general dependency analysis technique can only detect problem independent parallelism which is quite modest and not necessarily scalable.

1.2.3 MANNERS

The MANNERS program contains a hot-spot rule which fires repeatedly during the execution of the program. The number of repetitions increases

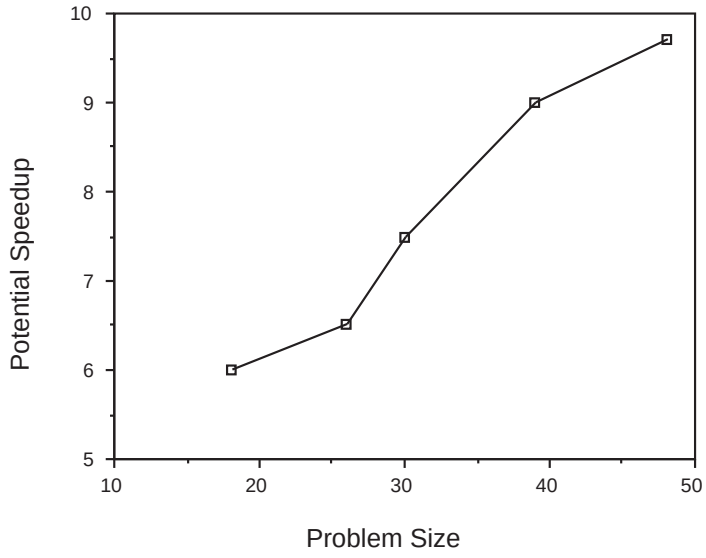

Figure 1.5: WALTZ Speedup.

dramatically with the problem size. After carefully analyzing the program and traces, it turns out that all instantiations of this rule can be fired in parallel. An impressive linear speedup as shown in Figure 1.6 should be obtainable if this important piece of semantic information can be exploited.

1.2.4 Remarks

The three examples above indicate that application specific information is the key to the effective exploitation of inherent parallelism in the problems and the rule-based programs. In this research, we have made an effort to systematically explore the potential and sources of this level of parallelism beyond that of compile-time dependency analysis techniques developed by previous research. Because of the use of application specific knowledge and the semantic nature of this approach, we call it *application parallelism* or *semantic level parallelism* in production systems.

1.3 Decomposition Abstraction: A Semantic Approach

The more experience we gain from programming parallel machines, the more we learn that run-time success more often associated with explicit decomposition. This is why most parallel programming languages, both imperative and declarative, provide constructs for parallel decomposition. The

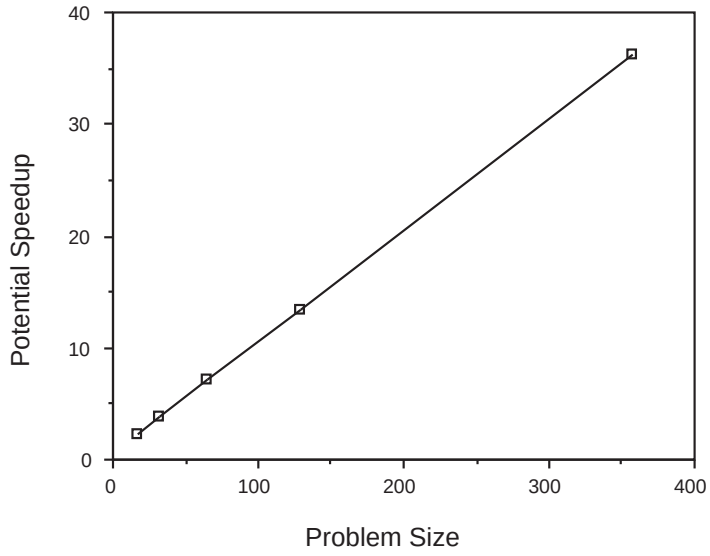

Figure 1.6: MANNERS Speedup.

PARTITION statement in Refined Fortran [77], the DECOMPOSITION and related statements in Fortran D [64], the Ada tasking mechanism [107], the `process` type in Concurrent C [50], the `pcall` and `future` in Multilisp [60], and the notions of *blackboard* and *theory* in Shared Prolog [17], just to name a few, are all examples of language mechanisms for data or function decomposition to facilitate the expression of parallelism. The models of parallel structuring used in these languages (particularly data partitioning [64, 77] and data parallel [63] models of parallel structuring) suggested analogies for rule languages to capture the application parallelism.

We propose a new approach called *decomposition abstraction* (DA) toward the expression and organization of semantic level parallelism in production systems. Decomposition abstraction is the process by which programmers specify decomposition strategies for the exploitation of parallelism embodied by an application. We provide the programmers with abstraction mechanisms, programming constructs, programming methodology, and compiler assistant to expose the application parallelism through data and function decompositions. It is our belief that, just like the roles played by procedure abstraction, control abstraction, and data abstraction in sequential programming, decomposition abstraction is the key to scalable and portable parallelism not only in rule-based languages, but also in other parallel languages as well. The scope of this thesis, however, is focused on rule languages.

Figure 1.7 depicts the comprehensive approach we propose that combine semantic-based and syntactic-based techniques to achieve high speed execution of parallel production system programs. All components except the syntactic-based techniques (i.e. *syntactic-based interference analysis* [70, 123] and *cluster analysis* [80, 101]) will be discussed in later chapters.

1.4 Summary of Results

The main results in this research can be summarized as follows.

- A new general formulation of production systems in an object-based framework together with a rule notation that abstracts away unnecessary details while characterizing all essential features of production systems. This formulation greatly simplifies our discussion. The framework provides a solid basis for specifying the formal semantics of the DA mechanisms.

- A set of DA mechanisms for rule languages, including *set selection conditions* (to match a qualified set of objects), *aggregate operators* (to operate on a selected set of objects as a whole), an **ALL** *combinator* (to combine several conditions into a specification of data decomposition), a **DISJOINT** *combinator* (to specify disjoint partitioning), and *contexts* (for grouping relevant rules and for the specification of causal dependencies between different groups of rules). By providing this set of minimal but semantically rich constructs and their formal semantics, we greatly clarify and formalize the essential elements of the so called *set-oriented constructs* [34, 52, 142, 156] in production systems.

- A semantic-based interference analysis technique for rule systems based on *data relationship specifications*. This technique determines parallel executable rules based on user-supplied semantic information in the form of *functional dependency*. Our notion of functional dependency is analogous to the corresponding notion in database systems but used in a completely different way. Specifically, it is used to characterize data relationship that implies disjoint decompositions. The recognition of this important property leads to useful theorems that forms the basis of our semantic-based interference analysis techniques.

- Methodologies for decomposition abstraction to transform sequential programs into parallel programs and to write parallel programs from scratch.

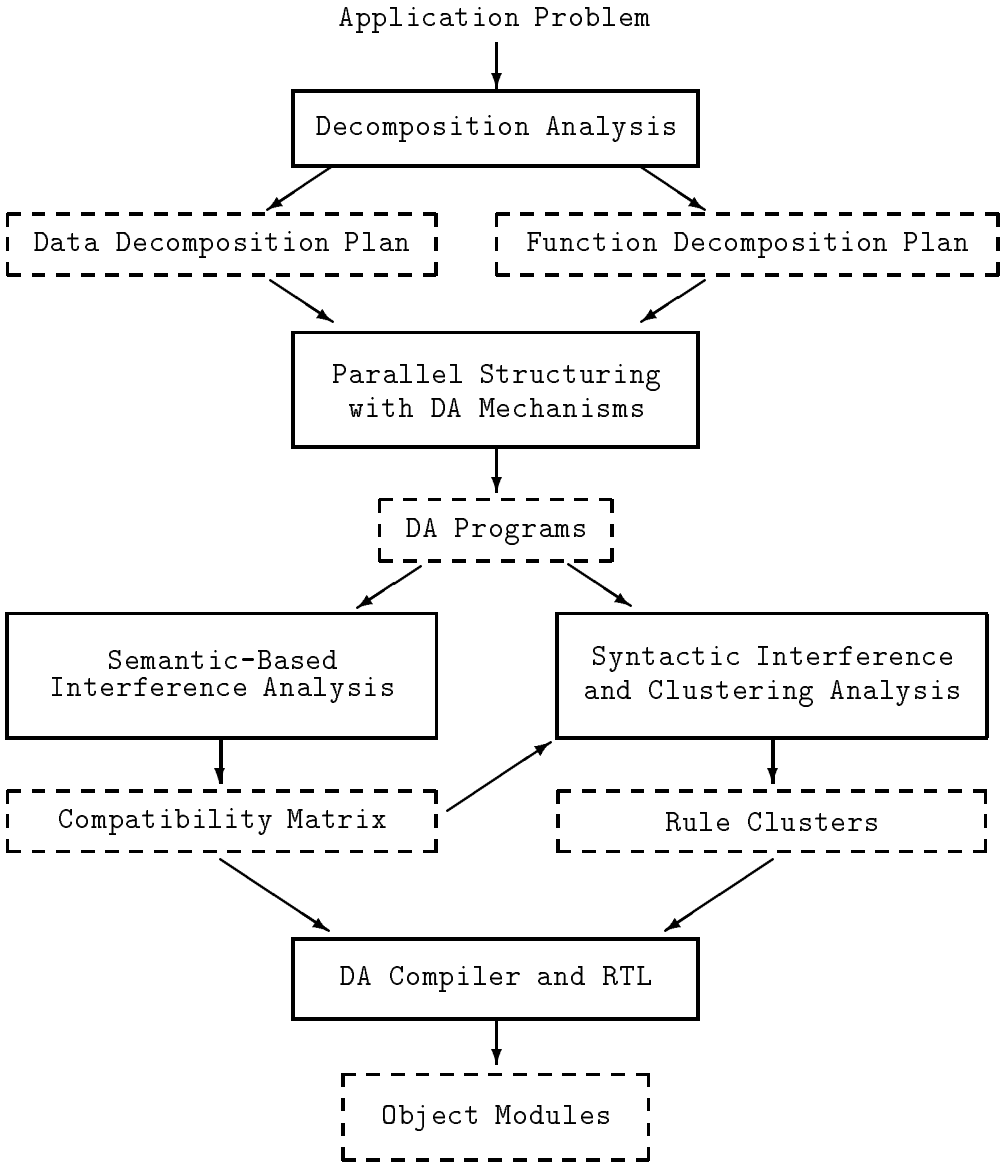

Figure 1.7: Decomposition Abstraction: A Comprehensive Approach toward Parallel Production Systems.

Sequential programs are converted by following a set of heuristic rules that identify and transform the parts that can be parallelized. Parallel programs are developed by following a sequence of steps that facilitate the effective use of DA mechanisms.

- A new technique for rapid system development and evaluation without the high cost of full-fledged system implementation or possible inaccuracy of simulation. This technique includes a parallel rule execution engine that fires multiple rules in parallel on the Sequent Symmetry multiprocessor and a work load generator that generates rule firing sequence from sequential execution trace file to feed into the parallel rule execution engine. The technique accurately reflects the system performance because all rule instantiations are faithfully executed and all scheduling and synchronization operations for correct parallel execution are actually performed. It also has the nice feature that, given the same rule program and the same data set, the simulator terminates with exactly the same results as the sequential execution it is based upon. This makes it trivial to tell the correctness of the parallel execution.

- Simulation results of applying the proposed DA mechanisms on three commonly used benchmark programs. A variety of experiments targeting factors that affects system performance are conducted on the parallel rule execution engine. Near linear speedup observed on all three benchmark programs provides a strong evidence that the DA approach and the proposed mechanisms are effective and scalable. The analysis of the simulation results suggests effective implementation strategies on the target machine.

- Implementation of a DA language called Venus/DA on Sequent Symmetry multiprocessor. The implementation demonstrates both the effectiveness of the DA approach and the process of converting a sequential rule language (Venus [18] in this case) into a parallel rule language supporting decomposition abstraction. The core of the implementation is a LEAPS-based [99, 100] parallel inference algorithm and an asynchronous rule execution engine. User-supplied semantic information are used in the algorithm for pruning the search space and for intelligent backtracking. Concurrently executable instantiations are generated in parallel without the need for conflict resolution or run-time interference analysis. The rule execution engine executes the instantiations asynchronously and enforces barrier synchronization whenever necessary. The integration of these techniques results in linear speedup on the benchmark programs.

1.5 Dissertation Outline

Chapter 2 surveys related works on parallel production systems as well as rule languages in database systems. Chapter 3 lays the groundwork of this research by formalize production systems under a general object model and provide an abstract rule notation for ease of discussion. Chapter 4, Chapter 5, and Chapter 6 constitute the main theorems and core technologies of this research. More specifically, Chapter 4 presents the set of decomposition abstraction mechanisms with illustrative examples and formal semantics. Chapter 5 introduces the notions of functional dependency and the semantic-based interference analysis techniques that determines the *semantic compatibility* between rules. And Chapter 6 describes the methodologies of transforming sequential programs and of writing parallel programs. A performance assessment on the parallel rule execution engine done before the real implementation is included as Chapter 7. The simulation leads to the implementation of Venus/DA, which is detailed in Chapter 8. Chapter 9 includes a good variety of experiments and performance results on the Venus/DA implementation. Finally, we conclude with the experience we gained from this research and point out the future directions of this work in Chapter 10.

Chapter 2

Related Works

The scope of this dissertation intersects with various research areas that are different in technical appearance but contain a nonobvious common in an underlying central issue — decomposition. This chapter presents a brief survey of the research in parallel production systems and relates other work within the theme of decomposition abstraction.

2.1 Parallel Production Systems

Parallelization of production rule systems has been a significant research topic for over a decade [85, 136]. Production systems have been assumed to encompass a high degree of parallelism [55]. Operations within all three phases can potentially be processed in parallel. Pipeline parallelism can be explored between phases or even across cycles. Driven by this expectation, a burst of research started in the early 1980s aiming at applying parallel processing techniques to address the performance issue [98]. Based on the target phase(s) or approach of parallelization, parallel production systems can be roughly classified into:

- systems that parallelize the match phase only,
- systems that fire multiple rules in a cycle or asynchronously (also known as multiple-rule-firing systems), and
- systems that employ other approaches such as specialized hardware architectures and connectionist production systems.

We go through parallel matching and other approaches briefly while discuss multiple-rule-firing systems in more details for their close relationship with our work.

2.1.1 Parallel Matching Sequential Rule Firing Systems

Early focus of research on parallel production systems were almost exclusively on parallel matching. These systems parallelize only the match phase of the recognize-act cycle. Conflict resolution and rule firing are still executed sequentially. The rationale behind this approach is the early report that production systems spent more over 90% of their execution time in the match phase [43, 54]. Just to name a few, Gupta and others [55, 56] explored parallelism in the Rete match algorithm [44], which is the match algorithm used in OPS5. The TREAT match algorithm by Miranker [96, 97] and other algorithms by Gupta [53] and Stolfo [137] were developed for DADO, a tree-structured massively parallel machine [140, 142]. DRete is a distributed version of the Rete algorithm proposed by Kelly and Seviora [73] for a special machine called CUPID [72].

The improvement in sequential match algorithms and advances in compilation techniques [62, 87, 100, 103, 126] drastically reduce the proportion of time spent in the matching to less than 50% as reported in [101]. From Amdahl's law, systems that parallelize only the match phase can not have significant speedup over the optimized sequential version. Parallelism in other phases of the recognize-act cycle must also be exploited.

2.1.2 Multiple Rule Firing Systems

Multiple rule firing systems parallelize not only the match phase, but also the act phase (actually, all phases) of the recognize-act cycle. Some systems even brake the barrier synchronization boundary between cycles by firing rules asynchronously. In this approach, maintaining the correct execution of a program becomes as important as the performance issue. This type of system is of particular interest to us since our decomposition abstraction mechanisms are designed for languages capable of firing multiple rules either synchronously or asynchronously. We described in more details several important work that have significant impact on the research of parallel production systems.

2.1.2.1 Ishida and Stolfo's Work The work done by Ishida and Stolfo [70] is both important and influential. Much work done by other researchers are either inspired by their work or using the same or similar analysis method proposed in their paper.

Two essential problems are discussed to realize parallel rule firings:

- **Synchronization Problem**: Rules may interfere with each others. It is necessary to identify the rules that must be synchronized.
- **Decomposition Problem**: Efficient decomposition algorithms are required to partition or distribute the rules so that multiple rules can be fired as often as possible.

They identified the possible interference between parallel execution of rules in OPS5-like language and proposed an important tool — *data dependency graph* — as the basis for synchronization analysis. By using this tool, they were able to produce a *synchronization set* for each rule, which contains all rules that must be synchronized with the rule in question. Rules that do not need to synchronize can be fired in parallel. For dependencies that can not be resolved at compile-time, run-time analysis is applied to increase the parallelism. This graph based analysis method has been widely used in many other works for similar analysis problems [80, 102, 118, 123, 125].

A less important result is their decomposition algorithm based on the so called *parallel executability* between each pair of rules which measures the number of production cycles that can be reduced by allocating the two rules in the same partition. The algorithm given is quite ad hoc and no method of computing parallel executability is given.

Though influential, I&S's method has been identified as overly restricted and in many cases, may cause unnecessary synchronizations [122, 123]. Two rules can be fired in parallel, under I&S's requirements, if they are *commutative* [112]. It has been demonstrated in other research such as [102, 122] that commutativity is too strong for efficient parallel rule execution.

2.1.2.2 IRIS IRIS [118] is claimed to be a production system programming methodology rather than a language. Motivated by an attempt to solve the problems of parallelizing production systems reported by Gupta [55], Pasik developed several techniques for reducing the software complexity and improving the parallelism in production systems.

Pasik proposed to partition a program into *rulesets* consisting of independent rules that can be fired in parallel. An external control mechanism is employed to invoke rulesets explicitly. A sequence of rules that always fire in serial are rewritten into a *macrorule*. *Table-driven rules* are used to provide knowledge representational and system maintenance advantage.

The most interesting technique, which is probably the one contributes the most to the effectiveness of the IRIS programs, is the technique called *copy-and-constrain* (C&C). Under this technique, a rule is called a *culprit rule* if it takes substantially more computation time to match and generate instantiations. This type of rules cause severe load balance problem in a multiple rule firing environment. The C&C technique is to replace culprit rule with an equivalent set of smaller independent rules which require less computation time. This technique proved to be quite effective not only in the IRIS production systems but also in other languages like CREL [80, 102] as well.

2.1.2.3 Ishida's Work Ishida has provided implementation methods and a parallel programming environment for multiple rule firing production systems [67, 68, 69]. The proposed methods combine compile-time and run-time dependency analysis and form the set of parallel executable instantiations using an incremental algorithm. Both *paired-rule conditions* and *all-rule conditions* (i.e. cyclic conditions) are used whenever appropriate for detecting interference.

The parallel programming environment provides language constructs and a simulation environment which in turn consists of an *analyzer* and a *simulator*. The construct of *ruleset* is introduced to form group of rules such that different conflict resolution strategies can be defined. A new conflict resolution strategy called *DON'T-CARE* is added to declare that rules in a ruleset are to be fired in parallel. A *focusing mechanism* is then provided for ordering the priority between rulesets. The simulation environment is used to obtain the performance results of the proposed methods. The compile-time interference analysis results, generated by the analyzer, are used in the simulator together with run-time analysis to achieve the most effective results.

2.1.2.4 Schmolze's Work Schmolze has conducted a series of research on multiple-rule execution systems in both synchronous and asynchronous environment [122, 123, 124, 125]. Their framework is generally taken from [70] and improve upon I&S's method. The basic approach is based on serializability. The parallel execution of multiple rule instantiations is *serializable* if there exists some serial execution of the same set of rule instantiations that would produce the same result. They called the problem of guaranteeing that each execution in a multiple-rule execution system is serializable the *serialization problem*.

Two causes of non-serializability are identified:

- **Disabling**: A set of instantiations, if executed in parallel, may *disable* each other. In such a case, the execution is not serializable.
- **Clashing**: If the order of the execution of the actions of multiple rules is not carefully controlled, two types of non-serializable effects may occur which is collectively called *clashing*.

  The first type can occur if one rule can add a WME that the other rule can delete and one rule can disable the other. If both rules are executed simultaneously, non-serializable result may occur.

  The second type can occur if one rule can add a WME that the other rule can delete and the actions from two rules are executed in an intermingled order. Again, the result of parallel execution may not be producible by any serial execution.

Examples are provided in the paper for both cases and it is instructive to read through them and try to provide additional examples.

The possible non-serializable effects due to disabling can be avoided by preventing the parallel execution of certain pairs of instantiations. The SELECT phase of the production system cycle is modified to prohibit the co-execution of any pair of instantiations that is critical to a cycle of disabling relations among instantiations.

Clashing is avoided by either prohibiting certain rule instantiations from co-executing as in the approach for disabling, or by imposing a partial order on the execution of actions of selected instantiations in the ACT phase.

The model is extended to asynchronous execution environment [125] where rules and WME's are physically distributed among several processors. The possible causes of non-serializability are still the same and the solutions are based on the same principles as in synchronous environment. A new problem which is unique in distributed environment is the WME inconsistency problem. Since the working memory is distributed, temporary inconsistency may occur which can lead to non-serializable effects. The inconsistency problem is solved by a simple protocol similar to the two-phase locking protocol [11, 40].

2.1.2.5 Kuo and Moldovan's Work Kuo, Moldovan, and their colleague have developed a parallel inference environment under the RUBIC project at USC for the analysis, simulation, and execution of parallel production programs [82, 81, 83, 84, 105, 106].

Two problems that must be solved by a multiple rule firing system are identified:

- **Compatibility Problem** To avoid interference between concurrently executing rule instantiations, a system must determine which rule instantiations are compatible, i.e. they do not interfere with each other.
- **Convergence Problem** Firing only compatible rule instantiations does not guarantee the correctness of the final solution because the system may search down a wrong path. The convergence problem is concerned with the control of multiple rule firing such that correct results are always guaranteed.

The *multiple-contexts-multiple-rules* (MCMR) model is proposed to address the problems at both *context* and *program* levels. At the context level, contexts are divided into *sequential* and *converging* contexts by using a set of UNITY-style [22] proof logics. Rule instantiations in a sequential context must be fired serially while those in a converging context can be executed in parallel without error. At the program level, only *compatible contexts* are activated in parallel such that the control flow of the program is not violated. Simulation results on the RUBIC simulator [106] show that the MCMR model performs better than both the *rule dependence model* and the *single-context-multiple-rules* model.

2.1.2.6 CREL It has been reported that the semantics of OPS5 production system language is not suitable for parallel execution [102, 124]. A natural way to cope with this problem is then to modify the OPS5 language so that the language is suitable for parallel execution. Design a completely new language also suffices. CREL [80, 102] is the result of an effort taking the first approach.

The syntax of CREL is identical to OPS5 but the semantics is different. Rules are executed asynchronously. CREL programs that run correctly in a sequential environment are guaranteed to run correctly in a parallel environment. The correctness of parallel execution is also based on serializability. A bipartite data dependency graph adopted from [70] is used in dependency analysis of rules. Two types of interference between rules are identified as special properties of the dependency graph. An algorithm is provided to find the *mutual exclusion sets* in a program, which are defined to be sets of rules that cannot be statically determined to be executable in parallel and thus require synchronization. Parallel execution is then guaranteed to be serializable if multiple rules selected from the same mutual exclusion set for parallel firing do not form a cycle with conflicting interferences. For running in an asynchronous environment, the *synchronization set* is defined to be a set of rules where global

synchronization is needed to ensure serializability. It is shown that a synchronization set is actually the maximum cycle among mutual exclusion sets where all synchronization is needed. The partition of rules by synchronization sets is then having the desired property that global synchronization is no longer needed among different partitions for correct execution. A program can thus be executed completely asynchronously.

Another contribution of this research is the optimizing transformations performed on programs to further increase the available parallelism. Several transformation techniques are developed and proved to be quite effective.

2.1.2.7 SPAM/PSM SPAM/PSM [61, 95] is not a production system language. Instead, it is a high-level vision system implemented as a production system. The reason why we want to discuss it here is that the concept of *task-level parallelism* promoted by SPAM/PSM is actually a form of semantic level parallelism. Task-level parallelism refers to parallelism inherent in the given task. It is certainly application specific and requires the programmers to provide the necessary knowledge for the exploration of the parallelism.

Three dimensions for task-level parallelism are identified:

- **Implicit vs. Explicit** The parallelism can be *implicit* such that the system or the compiler must extract parallelism out of the program code. On the other hand, *explicit* parallelism refers to providing explicit information for the system to explore task-level parallelism.
- **Synchronous vs. Asynchronous** A rule system can be executed either synchronously following the recognize-act cycle, or asynchronously if no global synchronization in the resolve phase across processors.
- **Distribution of Rules and WME's** Either rules or WME's can be distributed across processors. Or, there can be no distribution at all.

SPAM/PSM is a explicit and asynchronous system with WME distribution. The task-level parallelism is achieved by following a design methodology which systematically decomposes the given task into levels of subtasks for parallel execution.

The SPAM/PSM architecture and methodology proved to be quite effective in the vision domain and was able to achieve a 12-fold speedup on 14 processors [61]. It was also reported that the framework seems most suitable

for parallelizing knowledge-intensive systems that exhibit weak interaction between the individual subtasks of the task for which vision problem is a perfect example. However, it is not clear whether this approach is equally effective on other domains as well.

2.1.2.8 PARULEL Among all the multiple rule firing production systems, PARULEL [139, 143] is probably the only one that makes use of meta-level knowledge in forming parallel executable rule instantiations. Like SPAM/PSM, it is another example of using semantic level knowledge in multiple rule firing production systems. However, the approach taken is completely different from ours.

The most distinctive feature of PARULEL is that it is a two level system. *Domain rules* are for encoding domain knowledge while *meta-rules* (or *redaction rules*), on the other hand, are used to select parallel executable rule instantiations. The way meta-rules are used in PARULEL is quite unique in the literature. Programs are executed through the following cycles until a fixpoint if reached:

Match All domain rules are matched to form the conflict set.

Redact Incompatible rule instantiations are *redacted* by the meta-rules.

Fire All remaining rule instantiations are fired in parallel.

In other words, meta-rules are used to eliminate incompatible rule instantiations from the conflict set so that the resulting set of instantiations can be fired in parallel without error.

Naturally, the use of meta-rules is the most important feature in PARULEL. Programmers provide application specific control knowledge in a similar way as domain knowledge, i.e. by way of using rules. This results in a both uniform and flexible system. However, the responsibility of writing correct meta-rules to guarantee the correctness of the final result is completely on the programmers. The run-time overhead of matching and executing meta-rules can be substantial.

2.1.2.9 Neiman's Work UMass Parallel OPS5 [109, 110] is a Lisp-based OPS5 that support both parallel matching and multiple-rule-firing. Neiman points out the significant effect of scheduling overhead and the cost of guaranteeing serializability on the performance of parallel rule-firing production

systems. It is reported that run-time interference detection can impose an approximately 10% serial overhead on the execution. Synchronous rule firing results in even more serial bottleneck.

To reduce the scheduling and synchronization overhead, as well as the cost of guaranteeing serializability, Neiman combines a task-based scheduler and an asynchronous rule-firing policy with a weaker notion of serializability [111]. Rule instantiations may be associated with high-level tasks which can be executed asynchronously with one another. The asynchronous rule-firing policy executes rule instantiations as soon as they are generated. Correctness is ensured with language mechanisms in the design phase and a locking mechanism at run time.

In comparison with Neiman's work, we completely eliminate the need for run-time interference detection. Locking is minimized by generating independent and parallel executable rule instantiations directly. The decomposition abstraction mechanisms are among the first to provide general constructs for programming in parallel.

2.1.3 Hardware Approaches

When software approaches fail to deliver satisfactory results, hardware approach is an immediate alternative. Many architectures and machines for production systems have been proposed over the past decade. Only a small portion of it have been actually implemented. As early as 1980, Forgy has studied the possibility of implementing production systems on Illiac-IV [42]. The Concurrent Inference System (CIS) developed at MIT [15] is a forward- and backward- chaining system implemented on the Connection Machine. DADO [140] is a tree-structured machine architecture that employs the TREAT algorithm for parallel matching. Hardware prototypes for both DADO and the subsequent DADO2 [138] have been actually constructed. PESA-I, proposed by Schreiner and Zimmermann [128], is a distributed pipelined architecture implementing the Rete algorithm but without the need for any central scheduler or task queue. Simulation results show that 8000 rule-firings per second can be achieved. The Production System Machine (PSM) project at CMU is probably the most extensive research that studies the implementation of production system on both shared-memory [57] and message-passing architectures [1]. Based on the hardware available at that time, a shared-memory architecture is suggested to explore fine-grained parallelism in the Rete algorithm [56]. For executing the DRete algorithm discussed earlier, a special machine

architecture called CUPID [73] is designed to maximize the performance. The machine is designed to parallelize only the match phase while leaving a host computer to perform the conflict resolution and act phases. On the other hand, the RUBIC architecture and environment developed at USC [106] exploits the MCMR model of parallel production system mentioned earlier which is capable of activating multiple contexts and multiple rule instantiations in parallel. Finally, a parallel processing scheme called DYNAMIC-JOIN with associated parallel architecture is proposed by Oflazer [114]. The main idea behind the scheme is to reduce the variance in processing time of different rules. The reduction is made possible by a new state representation of rules and WMEs such that all possible partial matching information is included and be evenly distributed across processors. Again, the proposed architecture has not been actually implemented so far.

As a summary, hardware approaches have not offered a convincing success over software approaches. This may explain the reason why only a small portion of it has been actually implemented.

2.1.4 Other Approaches

In search for new approaches of implementing production systems, some other methods have also been investigated. Gaudiot and Sohn employ the so-called *macro data-flow* approach to implement the Rete algorithm [49, 135]. A data-flow approach is actually quite natural since Rete algorithm is basically a data-flow algorithm. A 17-fold speedup is reported on a macro data-flow multiprocessor simulator with 32 PEs. Another completely different approach is the so-called *connectionist production systems* which employ the connectionist architecture (i.e. neural network) to implement production systems. Galant [48] and Sohn and Gaudiot [134] both adopt *local representation* as the basic system architecture. Touretzky and Hinton introduce a different architecture called distributed connectionist production systems (DCPS) [153]. Sohn and Gaudio later on introduce a scheme, called *hierarchical representation* [133], that combines the local and distributed representation techniques.

2.1.5 Remarks

Even with such an extensive research effort, to effectively exploit the parallelism in production systems has been known to be very difficult. The results have not been quite to the expectation. The speedup achieved by systems with real implementation is quite limited, only about 10-fold, no matter how

many processors are used. The key reason is that the most valuable source of parallelism — the parallelism exhibited in the application domain — has been almost completely overlooked. This level of parallelism, which resides in the semantic characteristics of applications, is exactly the unexploited area of parallel production systems we intend to investigate in this dissertation.

2.2 Rule Languages in Database Systems

Rule languages also appear in database context. It is fair to say that the development of database production system languages is driven by the demand of integrating database and expert system technology. In general, there are two approaches toward a solution to this problem:

- augmenting a database system with rule constructs, or
- extending a production system with interface to databases.

Even though our research is targeted on main memory production systems, this line of research is still interesting to us because many problems that need to be solved are quite similar in these two contexts. In this section, we review some of the systems and languages with emphasis on the semantics of the rule languages and how the problem of concurrency control is addressed.

2.2.1 RPL

RPL (Relational Production Language) [33] is a proposed language for the integration of production system language and relational database. It was motivated by the similarity of the LHS of production rules to relational queries observed by Woods [163]. The goal is to enable a production system to directly access any conventional database that support a relational query language interface. The use of relational data model also provides a formal basis which is not usually seen in other conventional production system languages.

The syntax of RPL is based on SQL. OPS5 is used as a representative production system language for comparison. The data structures for an RPL program are defined using a relational DDL as in most relational database systems. What makes RPL different from other relational database systems is its addition of production rules for the manipulation of tuples. The LHS of a RPL rule is any valid SQL query which is relational complete. The RHS is then a collection of insert, modify, and delete tuple commands which correspond

directly to the make, modify, and remove actions of OPS5. It is the power of the relational complete LHS that makes RPL strictly more expressive than OPS5. The relational basis also makes it quite suitable for integration with relational databases.

Even though, as far as we know, RPL has never been fully implemented, the language proposal is quite influential. It demonstrates the potential benefits and feasibility of integrating database and expert system technology. It is also a good example of showing the advantage of having a formal model underlying a language design.

2.2.2 DIPS

Like RPL, the DIPS system [120, 130, 131] represents another example of using database technology in supporting production rules functionality. Two special data structures are used for the processing of OPS5 rules in a database environment: the Working Memory Relations (WM) and the Condition Relations (COND). Each class of WME's is stored as a WM relation. All condition elements in rules that refer to the same class of WME's, say C, are represented as tuples in a corresponding COND-C relation. In this way, both matching and instantiation generation can be done using database techniques. In particular, for the matching of variable-free condition elements, a simple selection of the corresponding COND-C relation is sufficient. For condition elements with variables, the necessary join of related WM relations is performed incrementally with intermediate results stored as tuples called *matching patterns* in the COND relations. From the parallel processing point of view, this approach is better than the RETE approach [44] since the propagation of changes can be performed in parallel to all the COND relations. More important, the conflict set is updated first in contrast to the RETE approach where conflict set is updated after the propagation is completed. Rules can thus be executed earlier than the RETE approach.

For the processing of applicable rules, since the matching patterns do not include identifiers to corresponding WME's, an additional selection of the corresponding tuples from the WM relations must be performed. The matching patterns provide necessary information for the selection criteria. The execution of applicable rules is then proceeded by treating the RHS actions of each rule as a database transaction. The concurrency control mechanism is then used to manage the execution of multiple transactions (i.e. rules) simultaneously. The correctness of concurrent execution is, as usual, based on serializability. Serializable execution is enforced by specialized locking mechanism. The conditions

under which relations must be locked are specified and a *logical* commit point is defined after which the execution of a rule is no longer affected by other rules.

2.2.3 The HiPAC Project

The HiPAC project [28, 29, 66, 93] is a representative research of the so-called *active database systems*. In here, we will concentrate only on the *Event-Condition-Action* (ECA) rules proposed by the project. In particular, we will discuss the *knowledge model* and the *execution model* of HiPAC designed to support the ECA rules.

The concept of ECA rules is central to the HiPAC knowledge model. A rule in the model is represented as an object with the following attributes: [93]

Event The event that triggers the rule.

Condition A collection of queries to be evaluated when the rule is triggered.

Action A sequence of operations to be executed when the condition is satisfied.

E-C Coupling A coupling mode that specifies when the condition is evaluated relative to the transaction that signals the triggering event.

C-A Coupling A coupling mode that specifies when the action is executed relative to the transaction in which the condition is evaluated.

The semantics is quite straightforward: when the event occurs (is *signalled*), evaluate the condition; and if the condition is satisfied, execute the action.

The event that triggers a rule can be a primitive event such as a database operation, a temporal event, or an external event. Primitive events can be combined to form composite events using disjunction and sequence operators. In the HiPAC execution model, rules are fired as nested transactions. When a rule is triggered, a transaction is created to evaluate the rule's condition. If the condition is satisfied, another transaction is created to execute the rule's action. The coupling modes control the time when the condition or action is scheduled to be executed.

If more than one rule is triggered, a condition evaluation transaction is created for each rule. For the set of rules with the same E-C coupling

mode, the evaluation of conditions will be executed concurrently. Similarly for the execution of actions. In this way, rules are fired concurrently as sibling transactions and the HiPAC transaction manager is responsible for insuring serializability. Since the action of a rule may contain operations that trigger other rules, cascading rule firings are possible and produce a tree of nested transactions.

The HiPAC rule system is interesting in the use of an object-oriented knowledge model to represent rules. However, it is a *passive objects passive rules*(POPR) model in the sense that both data and rules are passive entities to be interpreted by the HiPAC system. Since the database transaction mechanism is used to evaluate and execute the rules, serializability is still the sole correctness criteria.

2.2.4 The POSTGRES Rule System

POSTGRES [39, 145, 151] is one of the so-called *next-generation database systems* [21] designed to support non-traditional applications such as CAD/CAM, CASE, office automation, and engineering applications. The fundamental goal of POSTGRES is to provide data, object, and knowledge management services for such applications. Instead of giving a complete description of POSTGRES, we will again focus on the POSTGRES rule system [146, 147, 148, 149, 150].

The POSTGRES rule system is designed to be a general-purpose rule system in the sense that all the functions of view management, triggers, integrity constraints, referential integrity, protection, and version control, can be achieved using the rule system. Therefore, it is tightly integrated with the POSTGRES query language POSTQUEL.

Similar to the HiPAC rule, a POSTGRES rule is triggered by event which may be retrieve, replace, delete, append, new (i.e., replace or append) or old (i.e., delete or replace) to a data object. The condition to be evaluated after a rule is triggered is an arbitrary POSTQUEL qualification with no additions or changes. The action part is a set of POSTQUEL commands. In general, rules are for specifying additional actions to be taken as a result of user updates. These actions may activate other rules and result in forward chaining style of reasoning. On the other hand, POSTGRES allows backward chaining style of rule firing for deriving information from existing data. The programmers must determine and specify whether forward chaining or backward chaining is desired.

For the implementation of POSTGRES rules, two complementary methods are provided. The first is through *record level* processing which is called when individual records are accessed, deleted, inserted or modified. The second one is through a *query rewrite* module that converts a user command to an equivalent form which is suitable for optimization and efficient execution. The record level processing method is especially efficient when there are a large number of rules each of which covers only a few data instances. On the other hand, the query rewrite method works better if there are a small number of rules with large-scope. A *rule chooser* is planned for suggesting the best implementation for any given rule. Different policies to determine when the rules are actually activated similar to the coupling modes of HiPAC rules are also being explored.

2.2.5 Set-Oriented Rules in Starburst

The goal of the Starburst project [58, 59, 88, 89, 91, 129] at IBM Almaden Research Center is to build from scratch an *extensible* DBMS that supports new applications as addressed by the next-generation database systems [21] and, at the same time, provide a testbed for the research and experiments of new DBMS technologies. In contrast to other next-generation database systems that are object-oriented, functional, or based on nested relational model, Starburst is based on relational model and extensions to SQL. This allows Starburst to take advantages of proven relational database technology and facilitates porting existing applications to Starburst. A distinctive feature of the Starburst project is the effort to make it extensible *at every level*. In here, we will discuss the extension of Starburst to include user-defined rules [159, 160, 161].

Similar to HiPAC and POSTGRES rules, a Starburst rule has a *trigger clause*, a *condition clause*, and an *action clause*. A rule is triggered by one or more SQL operations (INSERT, DELETE, or UPDATE) on a relation which is called the rule's *trigger table*. The condition is evaluated at the end of the transaction that triggers the rule. The rule's condition clause is any SQL query on the database state or on a special type of relations called *transition tables*. Transition tables maintain records of the most recently updates to the rule's trigger table so that the rule can reference the data that are changed by the triggering operation. When the result of the query in the condition clause is nonempty, the action clause, which is a sequence of database commands, is executed. The actions may abort the transaction or perform further modifications to the database, which may in turn trigger the same or other rules. Any

modifications made by the actions are part of a transaction and can be rolled back. In case that more than one rules are triggered and satisfied, one rule is selected for execution after which other rules are reevaluated for eligibility. The process continues until no rules are satisfied. A partial order may be specified on the rules with the *precedes* and *follows* clauses to assign relative priority.

In addition to the user-defined rule which can only be triggered by built-in operations on a stored table, the Starburst's Alert trigger system [127] extends the rules system a step further to allow user-defined *Alert rule* to be triggered by any event(s) which may be an invocation of a user-defined *method* that may update many tables.

Even though the Starburst rules system is not a multiple rules firing system, it is still quite relevant to our work because Starburst rules are inherently set-oriented in the sense that they can be triggered by arbitrary sets of changes to the database and may perform sets of changes. The Starburst experience also pointed out that to achieve extensibility, it must be a fundamental goal and involve every aspect of the system design.

2.2.6 LDL

LDL (Logic Data Language) [24, 25, 108, 154] is a Datalog-like language extended with constructs such as negation, updates, control structures, and type constructors for developing intelligent data-intensive applications. The goal is to design a logic-based query language that combines the benefits of logic programming languages such as PROLOG, with the ease of use, the suitability for parallel processing, and secondary storage management of the relational systems. The LDL system is probably the first efficient realization of the concept of deductive databases [47].

Instead of describing the language in details, we will highlight the most important features of LDL, especially those that are related to our work.

- Unlike PROLOG, LDL is based on *pure Horn clause logic*. The dependence of order of rules in a program or subgoals within a rule has been removed. All extra-logical constructs (such as the *cut*) are discarded and the sequential execution-control model of PROLOG is no longer assumed. This results in a pure declarative language with clean semantics.

- *Complex terms* can be used in both facts and rules. This allows data to be structured and inferenced upon in a more natural and organized way.

- In order to integrate with relational database systems, sets data objects are introduced which can be used directly in the facts or rules. Sets can be explicitly enumerated or generated by rules. The response to a query is the set of *all* possible answers that can be deduced from the base relations. Aggregate operations such as `cardinality` can be used on the set objects.

- A negation with set-difference semantics is used instead of the negation by failure semantics of PROLOG [90].

- Definition and update facilities are provided for database schema, base relations, as well as derived relations.

As a related research to our work, the most interesting part of the LDL approach is its compilation techniques. The compilation process consists of the phase for compiling the rule program and the phase for the compilation and optimization of queries. The rule program is first transformed into a *predicate connection graph* [71] which is for storing the relationship between terms and the clause-heads that can potentially be unified. The structure is also used to maintain the entry points for queries. Recursive rules are compiled by means of naive evaluation and magic sets methods [7, 8]. Then for a given query, the system generates all possible proof plans in the form of proof schema and compiles a single relational algebra program (RAP) which, when executed, produces the set of all possible answers to the query. The RAP is further optimized using relational database techniques before execution. The final answers may be combined with intermediate results and proof schema to provide explanations about how the answers are derived. While other deductive database systems usually need a considerable amount of query-time deductive search to derive the answers, the LDL approach transfers much of this rule manipulation cost from query-time to compile-time. Together with the additional optimization phase, this approach has been proved to be quite effective [25].

Other techniques that are also important to LDL include the unification of *complex terms*, the compile-time analysis to detect unsafe queries, and the compiling of safe queries. These are discussed in [165, 166].

2.2.7 RDL1 and RDL/C

Unlike LDL [108, 154] or other deductive database systems [155], RDL1 [32, 76] rules are not in clausal form in the style of PROLOG or DATALOG, but closer to forward chaining rule based languages like OPS5. Again,

the design goal is to integrate production rule language with a relational DBMS. A rule consists of a *condition* part which is any tuple relational calculus expression with the constrain of being range restricted, and an *action* part which is a sequence of insertions and deletions of tuples in the database relations. The semantics of a RDL1 rule is defined as a mapping over database states. First, the *valuation* of a condition is an assignment to the constants, functions and free variables in the condition. A free variable x over a relation R is assigned with a tuple in the domain of R. Then the condition is interpreted under the current database state to determine whether the condition is satisfied. Similarly, the valuation of an action is the replacement of tuple variables with tuples from the proper domains. When the condition is evaluated to true, the tuples valuated in the action are inserted or deleted to the corresponding relations. An important feature of RDL1 rules is the atomic semantics of rule execution. The action part is not performed as a sequential execution of insertions and deletions. The whole action is considered as an *atomic database update*. Therefore the order of insertions and deletions are irrelevant. Only *net effect* is actually materialized. A finite set of rules defines a rule program which is executed in interpret-select-execute cycles similar to OPS5. Only one rule whose interpreted condition is true is selected for execution. This process repeats until a *stable state* is reached which is a state in which either every interpretation of every condition is false or no new database state can be produced using any action.

The most interesting feature of RDL1 is the modeling of rules using a special type of Predicate Transition Nets (PrTN) called *Production Compilation Network (PCN)* [31]. Structurally, a PCN represents the interrelationship between rules and relational predicates as specified by a rule program. It is also a pre-compiled form of the rules and allows incremental updates to the rule base. Dynamically, a PCN is also an execution model for the program it represents. The process of valuations and cyclic execution are actually done on the PCN. The net-based approach not only provides an efficient way of executing rule programs, it also means that all available techniques and tools for the analysis, transformation, and optimization of PrTN can be used on PCN. The reader interested in the details of PCN, including how to transform a rule program into and execute it on a PCN, is referred to [31, 32]. Another facility provided by RDL1 is a control language for annotating PCN with information of flow of control and the sequence of rule firing. For a complete and formal description of the language, see [30].

In summary, RDL1 is a powerful rule language that supports negations in conditions and allows sequence of insertions and deletions in the actions. It has been proved to be more expressive than DATALOG^{neg} [132]. The PCN model and the compilation technique provide a good basis for rule/query optimization and efficient execution. Though primitive, the control language offers programmers a way of specifying the order of execution of the rules and is also a tool for describing and developing query processing strategies and transformations over the PCN.

Even though the RDL1 rule programs and the associated PCNs are potentially parallelizable, it is not a multiple rule firing system since only one rule is selected for execution in each cycle. Furthermore, two main limitations of the RDL1 approach are reported [75]:

- The rule interface is DBMS-dependent.
- The rule language does not provide programming features usually found in expert system shells such as control structures, main memory variables, connections with a procedural language, and user interaction.

In response to these deficiencies of the RDL1 approach, a new rule language compiler called RDL/C [23, 74, 75] is developed which not only supports procedural constructs and C language interface, but also enable a program to run on top of any relational DBMS because the interface between the rule language and the DBMS is SQL. More importantly, RDL/C provides language constructs, execution model, and run-time environment for supporting rule-level parallelism. We now briefly describe the RDL/C approach.

RDL/C is derived from RDL1. The language supports both declarative programming in the form of production rules, and procedural programming based on C code. The original design of RDL/C does not include constructs for parallelism. Similar to RDL1, the condition part of a RDL/C rule is a tuple relational calculus expression with the declaration of range variables in front. The expression is to qualifies the tuples for participating in the rule's firing. The actions can be insertions, deletions, C-like variable assignment and external procedural calls. The semantics of RDL/C rules is set-oriented in the sense that when a condition is evaluated against the database, the set of instances satisfying the condition is returned. Similarly, when an action is executed against the database, it is executed for all the values which appear as arguments in the action. In this respect, RDL/C rules are similar to a construct we provide called ALL combinator. For the execution of a rule program,

one rule is randomly selected for execution among all firable rules in each cycle. Program execution terminates when no rule is firable. A control sub-language is also provided as in RDL1 for explicit control over the rule execution. Two particular expressions BLOCK and SEQ specify non-deterministic or sequential execution.

As already mentioned earlier, the original design of RDL/C does not include constructs for supporting parallelism. In [23] the basic design is extended to include constructs for supporting parallelism. First, on the PCN model, which is the execution model for RDL/C language, sufficient conditions are identified for parallelizable transitions which correspond to rules in the program. To inform the compiler that the rule module is to be run in parallel, the ON statement is provided to specify the servers on which the module is to be executed. Then the PAR structure is added to the control sub-language to specify how rules are to be run in parallel. A run-time library is provided to actually manipulate parallelism during execution.

Though primitive, the RDL/C approach to parallelism is actually a rudimentary form of semantic level parallelism. However, the parallelism is only supported at the rule level and the programmer must transform application specific knowledge into an explicit specification of partial order among rules using the control sub-language. On the contrary, our approach is intended to be a much more general and comprehensive one that exploits semantic parallelism at every level of production system. The application specific knowledge is also expressed and used in a much more natural way.

2.2.8 Gordin and Pasik's Work

In [51, 52], Gordin and Pasik have developed several set-oriented constructs and showed how these constructs can be added to a DBMS implementation of OPS5. The new constructs are implemented by an extended version of the Rete algorithm [44].

A condition element (CE) is set-oriented if it is enclosed in square brackets. The semantics of a set-oriented CE is to match with all consistent WME's to be associated with a single rule instantiation. From the database point of view, if a rule contains only set-oriented CEs, then the entire relation with tuples satisfying the CEs is generated with the instantiation when the rule is matched against the database. However, a LHS can contain both set-oriented and regular CEs. In this case, the regular CEs can be considered as partitioning the relation into smaller relations, or equivalently, the set-oriented

CEs can be seen as combining the tuples forming the regular instantiations into aggregated instantiations.

Similarly, a pattern variable (PV) is set-oriented if it occurs within a set-oriented CE. The domain of values of a set-oriented PV is the set of values occurring in the WME's satisfying the corresponding CE. When a set-oriented PV occurs in more than one CEs, a join is performed. When a PV occurs in both a set-oriented CE and a regular CE, it is bound to the value in the WME matching the regular CE. A PV in a set-oriented CE can be forced to be non-set-oriented by listing it in the `:scalar` clause. The effect is to partition the relation induced by the LHS into separate instantiations. Aggregate operators such as *count, min, max, sum*, and *avg* are provided for more expressive LHS.

For the RHS actions, two types of capabilities are provided for accessing set-oriented PVs or CEs. First, aggregate operations such as *set-remove* and *set-modify* are added to operate on an entire set. Then a **foreach** iterator construct is provided to execute its body on each subset of the instantiation, having a distinct value for a specified set-oriented PV. The partitioning is similar to the SQL **group-by** and the iterator can access the items in ascending, descending, or default order. By matching on a set of values and iterating over them, subinstantiations correspond to distinct values of the set-oriented PV can be access in a single rule firing. The **foreach** operator can be nested to have compositional effect. The operator can also be applied on set-oriented CEs with the semantics of iterating through the matching WME's rather than values.

For the implementation, an extended version of the Rete algorithm is developed for processing set-oriented constructs. A discussion of how to integrate this work to the DIPS system [131] is also given to show that OPS5 with the proposed constructs can be used as the language basis for expert database systems.

As a comparison, our approach differs in that we take an object-based approach rather than a relational database approach. Our design is to have a more expressive LHS and a SPMD semantics for RHS instead of the iterator approach which is contrary to the declarative nature of production systems. As an example of more expressive LHS, the semantics of `DISJOINT` construct can not be expressed by their corresponding LHS set-oriented constructs. Most importantly, our major concern is concurrency which is not the design goal of Gordin and Pasik's work. No discussion was given about the interference analysis of rule and multiple rule firing.

2.3 Chapter Summary

We discussed rule systems for traditional and data base applications. The key to successful parallelism is the independence of parallel activities. This in turn manifests itself on the disjointness of data objects accessed by the parallel activities. We therefore conclude that the lack of mechanism for the specification of decomposition is the main reason for the limited speedup in previous work. A general approach toward data and functional decomposition is required to have significant performance improvement on production system programs. The main contributions of this research are to provide a general framework and proper abstraction mechanisms for such purpose. Another goal of this research is also to demonstrate that decomposition abstraction is the missing layer which needs to be superimposed upon the familiar procedural, control, and data abstractions to achieve truly portable parallel programming using any language.

Chapter 3

A General Object-Based Framework

The principles of decomposition abstraction are language independent. It is best to discuss it under a general framework rather than under the context of any specific rule language. Such a framework must cover all essential features of production systems. In this chapter, we propose a general object-based framework and a generic rule notation. The intention is to provide a language independent context for our discussion and to facilitate the applicability of our results to any rule language.

3.1 Object Model and the Abstract Rule Notation

We have built our framework on top of a unified object model which can be used to characterize all entities in a rule system. The basic object model is inspired by [6, 14] and is comprised of the following sets of symbols:

$\mathcal{A}$: attribute names,

$\mathcal{C}$: class names,

$\mathcal{I}$: identifiers,

$\mathcal{M}$: method names,

$\mathcal{R}$: rule names,

$\mathcal{V}$: variable names.

Definition 1 (Methods) *A* method definition *is a triple* (M, P, B) *where* M *is a method name,* P *is a set of parameter specifications, and* B *is the definition of operations performed by the method (usually called the* body *or* implementation *of the method). A* method invocation *is a method name with necessary parameters fully supplied.* □

We have deliberately left out the details of how a parameter or body of a method is actually specified. No restriction is placed on the way actual arguments are passed in a method invocation. These issues are not essential to our discussion and thus our results are independent of any specific method definition or invocation mechanism.

Definition 2 (Classes) *A* class *defines a set of objects with similar structure and behavior.*

- **INT**, **FLOAT**, *and* **STRING** *are* primitive classes *representing the set of integers, floats, and character strings, respectively.*
- *An* attribute definition *is a pair* (a, C) *where* a *is an attribute name and* C *is a class name.*
- *A set-valued attribute can be defined by adding a "*$*$*" at the end of an attribute name.*
- *A class is a triple* (C, A, M) *where* C *is a class name,* A *is a set of attribute definitions, and* M *is a set of method definitions.* □

In terms of our abstract rule notation, a class C with attribute definitions $(a_1, C_1), \ldots, (a_n, C_n)$ and method definitions $M_1, \ldots, M_k$ is defined as follows.

class C {
  attributes ($a_1 : C_1, \ldots, a_n : C_n$)
  methods ($M_1, \ldots, M_k$)
}

The sets of attributes and methods of C are denoted by $A(C)$ and $M(C)$ respectively.

Definition 3 (Objects and WME's) *Objects are defined to model WME's. They are the basic units of information and behavior encapsulation. Inheritance is not considered since it is not an essential part of the production system model.*[1]

[1] However, our formalization is general enough to be extended later to include inheritance.

- *Integers, floats, and character strings are* primitive objects.

- *If* $a_1, a_2, \ldots, a_n$ *are the attribute names of a class* C *and* $O_1, O_2, \ldots, O_n$ *are objects, then:*

  $$O = (\ a_1 : O_1,\ a_2 : O_2,\ \ldots,\ a_n : O_n\ )$$

  is a structural object. *The object is an* instance *of the class* C.

- *Objects are the generalization of WME's. Each object has a unique identifier associated with it.* Working memory *is a set of objects.*

- *If* $O_1, O_2, \ldots, O_n$ *are objects of a class* C, *then*

  $$\{\ O_1, O_2, \ldots, O_n\ \}$$

  is a set object. O_i*'s are* elements *of the set object. Note that elements of a set object must be instances of the same class.* □

Definition 4 (Rules) *A* rule *has a triggering condition and an action component. Conditions may be positive or negative.*

- *An* expression *is a quantifier-free first order formula.*

- *If* v *is a variable name,* C *is a class name and* E *is an expression, then* $(v : C :: E)$ *is a* positive condition *and* $-(v : C :: E)$ *is a* negative condition.

- *If* P *is a condition, then* $v(P)$, $C(P)$, *and* $E(P)$ *denote the variable, class, and expression components, respectively, of the condition.*

- *A* rule *is a triple* (P, N, M) *where* P *is a set of positive conditions,* N *is a set (possibly empty) of negative conditions, and* M *is a set of method invocations.*

- *A positive or negative condition is termed a* condition element. *The set of all condition elements is called the* antecedent. *The set of method invocations is called the* consequent. □

Apparently a rule with empty consequent has no effect, therefore M is usually non-empty. Since methods can only be invoked on objects selected by the positive conditions, P must be non-empty as well.

In the rule notation, if r is a rule name, $P_1, P_2, \ldots, P_n$ $(n \geq 1)$ are positive conditions, $N_1, N_2, \ldots, N_m$ $(m \geq 0)$ are negative conditions, and $M_1, M_2, \ldots, M_k$ $(k \geq 1)$ are method invocations, then a rule is defined as follows with $\rightarrow$ delimiting the antecedent and consequent.

rule r {
$P_1, P_2, \ldots, P_n,$
$N_1, N_2, \ldots, N_m$
$\rightarrow$
$M_1, M_2, \ldots, M_k$
}

Definition 5 (Program and System) *A* program *is a pair* **(C, R)** *where* **C** *is a set of class definitions and* **R** *is a set of rule definitions. A* rule system *is also a pair* **(O, P)** *where* **O** *is a set of objects and* **P** *is a rule program.* □

3.2 Execution Model and Semantics

We specify the semantics of rules by considering rule antecedents as queries to the working memory for selecting a consistent set of objects. The execution of a rule system is defined in terms of state transitions between working memory states.

Definition 6 (State) *The* state *of a rule system is the set of objects in working memory.* □

Definition 7 (Selection and Instantiation) *Pattern matching is modeled by object selection. The following definitions are defined assuming a given state* S.

- *A positive condition element* $(v : C :: E)$ *is* satisfied *in* S *if there exists an object of class* C *such that* E *is evaluated to true. The object (which can be referenced by the variable* v*) is said to be* selected *by the condition element.*

- *A negative condition* $-(v : C :: E)$ *is satisfied in* S *if there does not exist any object of class* C *such that* E *is evaluated to true.*
- *A rule is satisfied in* S *if there exists at least one set of objects in* S *such that all condition elements in the antecedent are satisfied. The set of objects selected by the positive condition elements is called an* instantiation *of the rule.* □

Formally, a rule as defined in Definition 4 is satisfied if the following formula is true.

$$\begin{aligned}
&\exists\, v(P_1), \ldots, v(P_n)\ (\\
&\quad v(P_1) \in C(P_1)\ \wedge\ \ldots\ \wedge\ v(P_n) \in C(P_n)\ \wedge\\
&\quad E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge\\
&\quad \nexists\, v(N_1), \ldots, v(N_m)\ (E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)))
\end{aligned}$$

Each set of n objects satisfying the formula is an instantiation of the rule.

Operationally, a rule can be considered as a query to the working memory. The result of the query is a class whose instances are instantiations of the rule. In other words, the set of all instantiations of a rule r, denoted $\mathbf{Inst}(r)$, can be formally characterized as the set

$$\mathbf{Inst}(r) \equiv \{\ t \mid t \in Inst\_of\_r\ \wedge\ A\theta\ \}$$

where $Inst\_of\_r$ is the class

$$\begin{aligned}
&\textbf{class}\ Inst_of_r\ \{\\
&\quad \textbf{attributes}\ (\ v(P_1) : C(P_1),\ \ldots\ , v(P_n) : C(P_n)\ )\\
&\quad \textbf{methods}\ (\ \bigcup_{i=1}^{n} M(C(P_i))\ )\\
&\}
\end{aligned}$$

and θ is the variable substitutions and A is a formula representing the tests in the antecedent. That is,

$$\begin{aligned}
\theta &= \{\ v(P_1)/t.v(P_1), \ldots, v(P_n)/t.v(P_n)\ \}\\
A &= E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge\\
&\quad \nexists\, v(N_1), \ldots, v(N_m)\ (E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)).
\end{aligned}$$

Note that the value of each attribute in the class is an object selected by the corresponding positive condition element, and the methods are the union of all methods that can be invoked on the selected objects.

Definition 8 (Rule Firing) *Let S be a state, r be a rule which is satisfied in the state, and i be an instantiation of r. The result of* firing *the rule instantiation is a new state S' obtained from S by invoking the methods in the consequent of r on the set of objects in i. We denote such a rule firing by $S' = S(i)$.* □

Definition 9 (Execution) *An* execution *of a rule system is a sequence of rule firings that transforms the system from a state to another state. A state is a* terminal state *if no rule is satisfied under that state. An execution is a* terminal execution *if the last state in the sequence of rule firings is a terminal state.* □

Note that an execution is not required to be a terminal execution. This is to allow systems that do not terminate. It is also important to note that in the definitions of rule firing and execution, no restriction is placed on how objects are selected or on which rule instantiation to pick. In other words, no matching technique or conflict resolution strategy is assumed.[2]

The framework and execution model above characterize the core concepts and essential features of a sequential production system. We extend the model to allow simultaneous firing of multiple rule instantiations.

Definition 10 (Interference) *If i_1 and i_2 are instantiations of two (possibly the same) rules that are satisfied in a state S, then i_1* interferes with *i_2 if any one of the following conditions is true:*

1. *The execution of i_1 prevents i_2 from being an instantiation in the new state resulting from i_1's execution, or vice versa.*

2. *There exist methods invoked by i_1 and i_2 that modify the same object.* □

Since a newly created object is always assigned a unique identifier, object creations do not contribute to any interference except when Condition 1

[2] In fact, our language model to be discussed later does not even have a conflict resolution phase. The idea of generating a bunch of instantiations and then resolving the conflict is considered a waste of computation resource. Our approach is to generate only those instantiation(s) that is(are) actually fired.

is true. Identical objects with different identifiers are allowed to coexist in our model, which is consistent with most rule languages.

We note that it is possible to weaken Condition 2 above since we need only to avoid conflicting methods to be invoked on the same object. However, such fine-grained parallelism can be easily overwhelmed by the potential complexity. We reserve this issue for future research.

Definition 11 (Compatibility) *Two instantiations are said to be* compatible *if they do not interfere with each other. A set of instantiations is compatible if the instantiations are pair-wise compatible.* □

Since compatible instantiations do not interfere with each other, they can be executed in parallel. Our definitions of interference and compatibility are similar to the corresponding definitions in [70, 80, 84, 123] which are all essentially originated from Bernstein's conditions [10] and database concurrency control theory [11, 117]. However, we formalize it to a general object-based context which allows any type of method instead of just the add, delete, and modify operations as in most previous work on parallel production systems.

Definition 12 (Parallel Rule Firing) *The result of* parallel firing *of two compatible instantiations in a state is a new state obtained by invoking all methods on corresponding objects of the two instantiations. Likewise, the parallel firing of a set of compatible instantiations I in a state S is to invoke all methods on corresponding objects of all instantiations. The parallel firing is denoted by $S' = S(I)$.* □

Because of the non-interference requirement between parallel executable instantiations, the resulting state of the parallel firing is the same as the result of execution of the set of instantiations in sequence following any order. To state it more precisely, if $I = \{i_1, \ldots, i_n\}$ is a set of parallel executable instantiations in a state S, then

$$S(I) \;=\; S(i_{j_1})(i_{j_2}) \ldots (i_{j_n})$$

where $j_1, j_2, \ldots, j_n$ is any permutation of n.

3.3 Chapter Summary

In this chapter, we presented a general formalization of production system on top of an object model. An abstract rule notation is introduced to facilitate the coming discussion of language mechanisms in a language independent way. The greatest benefits of this approach are the simplification of discussion and the general applicability of results. It will be clear in later chapters that our framework and rule notation greatly simplify the presentation of semantics of our language mechanisms. We will also demonstrate the generality of our results by showing how to adopt our mechanisms to convert sequential rule languages into parallel rule languages.

Chapter 4

Decomposition Abstraction Mechanisms

Decomposition abstraction mechanisms are language mechanisms that assist the programmers in the abstraction process for parallel decomposition. Even though language constructs for parallel decomposition have been in existence for quite a while, none of them seem to fit under the context of production system. This is primarily due to the fundamental differences in computation model. The design of decomposition abstraction mechanisms for production system must be in harmony with the essence of production system and its distinctive computation module. In this chapter, we first present a systematic analysis of the types of parallelism in production systems to derive a set of design criteria. We then introduce a small but powerful set of language-independent abstract mechanisms for parallel decomposition. Actual constructs for parallel decomposition in any rule language can be easily designed by adopting these mechanisms.

4.1 Parallelism in Multiple Rule Firing Systems

When rule instantiations are allowed to fire in parallel, various opportunities for parallelism arise at different levels of the production system model. This analysis of parallelism is different from Gupta's analysis [55] in that we examine this issue from a semantic point of view. In particular, we focus on the patterns of computation and programming style that naturally map to the familiar notions of data and function decomposition. Unlike such notions like *node parallelism* from Gupta's analysis, our analysis is independent of any match or rule evaluation scheme.

4.1.1 Data Level Parallelism

In a sequential environment, WME's are processed one at a time. By *data level parallelism* we mean that different sets of WME's can be processed in parallel similar to the SPMD or data parallel systems [27, 63]. However, unlike other languages where data resides in regular data structures, this type

of parallelism in the context of production systems usually manifests in the form of multiple (either all or subset of) instantiations of the same rule. Specifying this type of parallelism with data declarations is not likely to work since the same set of WME's may need to be processed sequentially for some rules while they may be fully decomposable for some other rules. We will show in later sections that declarations on a per rule basis turn out to be the most natural way for covering this level of parallelism.

4.1.2 Rule Level Parallelism

This is the concurrency observed when instantiations of multiple rules are fired in parallel. It can be the result of both data and function decomposition depending on whether the rules are designed for similar or different functionalities. For exploiting this level of parallelism, the main issues are the possible interference between different instantiations and the correctness of parallel execution. Straightforward specification is clearly inappropriate because the complexity of reasoning about concurrency and interference is likely to be too heavy a burden for the programmers. The preferred way is to have the language system derive the concurrency, possibly with the help of a minimum amount of semantic information provided by the programmers, and maintain the correctness of parallel execution. We will show that relationships between data objects provide the key gateway to the exploitation of rule level parallelism in production systems.

4.1.3 Program Level Parallelism

Finally, a problem can often be decomposed into subproblems such that part or all of them may be processed in parallel. This corresponds to the program level parallelism where the structuring of program provides valuable hints for function decomposition. However, the basic production system model does not have any notion of modules or rule groups, which is certainly a disadvantage from this point of view. Mechanisms for rule structuring would certainly help the programmers in program development and the system in uncovering this type of parallelism.

4.2 Design Criteria

From the analysis of the types of parallelism presented in the previous section, we derive a set of criteria that must be met by any proper decomposition abstraction mechanism for a parallel rule language.

What vs. How The mechanisms must be declarative in nature. Specify what type of decomposition naturally exhibits the parallelism in the application independent of how the decomposition is actually achieved.

Consistent with Pattern Matching Paradigm The whole idea of production system is centered around pattern matching. The mechanisms for parallel decomposition must also be expressed under the pattern matching paradigm.

Conciseness As simple as possible, but no simpler. The mechanisms must be conceptually simple and intuitively appealing. The burden of reasoning about concurrency placed on the programmers should be minimized to the extent that only natural parallelism in the application semantics need be considered.

Versatility The set of mechanisms must be semantically rich and powerful enough to express as many types of parallelism as possible.

Compatibility The mechanisms should be compatible with sequential semantics. All sequential programs should still run correctly. Parallel programs should be able to run correctly even if executed sequentially.

Effective Implementation Any set of mechanisms for parallel decomposition should be feasible.

4.3 Parallel Structuring Mechanisms

Following the design criteria, we propose a set of abstract mechanisms for expressing decomposition strategies. The formal semantics of the mechanisms are specified under our object-based model with illustrative examples using the abstract rule notation.

4.3.1 Set Selection Conditions

In most (if not all) sequential rule languages, each positive condition element matches a single data object from a specified class. Then actions in the consequent are applied on the instantiation composed of selected objects, one from each positive condition element. This implies that only one object

from each class can be processed at a time. On the other hand, it is quite natural for an application to adopt a basic problem solving strategy such that all objects satisfying certain conditions in a specified class need to be processed. This apparent mismatch between the language model and the application semantics is almost always circumvented by firing the same rule repeatedly until all qualified objects have been processed. For example, the rule

rule $Raise\_Poor\_Employee$ {
    ($d : Department$),
    ($e : Employee :: e.dept == d.name \ \wedge \ e.salary < 10000$)
$\rightarrow$
    $e.salary = e.salary + e.salary/10$
}

will fire repeatedly on each "poor" employee in all departments to raise his/her salary by 10%. This type of rule can be found in almost all rule programs. If each employee belongs to exactly one department, the rule actually represents a perfect case of DOALL loop [167] in which parallelism can be fully exploited. On the other hand, this is not at all obvious for many parallelizing compilers of sequential rule languages since the possibility of interference can not be ruled out at compile-time. If the application semantics implies that no interference can occur, then there is no reason to be so conservative. What we need here is a mechanism for specifying the exact semantics of the application as to whether a rule is to be applied on all or just the selected object one at a time.

For achieving the purpose above, we found that enriching the semantics of the rule antecedent to allow a positive condition element to match not just one but all satisfying objects solves the problem naturally and elegantly. Using the abstract notation and the example above, the rule below specifies that for a department d, select all poor employees and raise the salary of each one of them by 10%.

rule $Raise\_All\_Poor\_Employees$ {
    ($d : Department$),
    $[\![$ $e : Employee :: e.dept == d.name \ \wedge \ e.salary < 10000$ $]\!]$
$\rightarrow$
    $e.salary = e.salary + e.salary/10$
}

A positive condition element enclosed in square brackets is a *set selection condition* denoting that all qualified objects should be processed by the consequent and that they can be processed independently. In other words, the selected set of data objects is fully decomposable and can be processed in parallel.

The square bracket notation is adopted from [52] for its conciseness. However, the semantics is rather different. A relational semantics was taken in their set-oriented constructs to facilitate the integration with database systems. Our set selection condition is a mechanism for expressing parallelism. The implication that objects in the selected set can be processed in parallel is not in their relational semantics.

Formally, a rule r with positive conditions $P_1, \ldots, P_i$, set selection conditions $P_{i+1}, \ldots, P_n$, and negative conditions $N_1, \ldots, N_m$, defines the following class.

class $Inst\_of\_r$ {

attributes ($v(P_1) : C(P_1),\ \ldots\ , v(P_i) : C(P_i),$
$v(P_{i+1})* : C(P_{i+1}),\ \ldots\ , v(P_n)* : C(P_n)$)

methods ($\bigcup_{i=1}^{n} M(C(P_i))$)

}

Each instance of the class represents a *set instantiation* composed of objects (one from each regular condition) and set objects (one for each set selection condition). Set instantiations are, as the name suggests, representations of sets of *ground instantiations*, which are instantiations of the *ground rule* of r obtained by treating all set selection conditions as regular conditions. The set of all ground instantiations can be characterized by the set

$$\{\ (g_1, \ldots, g_n) \mid \exists\ t\ (\ t \in Inst\_of\_r\ \wedge\ g_k = t.v(P_k),\ 1 \le k \le i\ \wedge\ g_k \in t.v(P_k)*,\ i+1 \le k \le n\ \wedge\ A\theta\ )\ \}$$

in which

$$\begin{aligned} \theta &= \{\ v(P_1)/g_1, \ldots, v(P_n)/g_n\ \} \\ A &= E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge \\ &\quad \nexists\ v(N_1), \ldots, v(N_m)\ (E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)). \end{aligned}$$

We note that when more than one set selection condition coexists in a rule, a *join* semantics similar to a relational join is implied. This is consistent with sequential semantics in that the set of ground instantiations is exactly the same as that of the rule with set selection conditions treated as regular conditions. The difference is that the former is fully parallel decomposable while the later can only be processed one at a time.

Among previous work on parallel production systems, van Biema et al. [156] were probably the first to point out the issue and provide constructs for set-oriented processing in rule-based programming. Our set selection condition is similar to their universal quantification. However, the exact semantics of the universal quantification has not been formally specified as we have done here. This can easily result in ambiguous and complicate rules.

4.3.2 Aggregate Operators

The set of objects selected by a set selection condition can also be processed as a whole by *aggregate operators* such as **count**, **sum**, **max**, **min**, and **avg**. This provides a new dimension of language constructs that greatly simplify rule-based programming. The fact that efficient parallel algorithms can be used to implement these operators further increase the value of set selection mechanism in rule languages.

Using our abstract rule notation, in a set selection condition [v : C :: E], v denotes an individual and $v*$ the whole set of selected objects, respectively. For example, if the number of poor employees in a department is desired, the following rule does exactly what we want.

rule $Count\_Poor\_Employees$ {
($d : Department$),
[$e : Employee :: e.dept == d.name \ \wedge \ e.salary < 10000$]
$\rightarrow$
$d.poor\_emps = \mathbf{Count}(e*)$
}

Without set selection conditions and aggregate operators, the same effect would require two rules, where one rule fires repeatedly on each poor employee to increment a counter and another semantically redundant rule is used purely for testing whether all poor employees have been counted.

4.3.3 ALL Combinators

Examining the *Count_Poor_Employees* rule above reveals an additional, unexploited level of parallelism. Using only the set selection condition and aggregate operators, all poor employees in a department can be processed as a whole using an efficient parallel algorithm. However, different departments are still processed sequentially. To specify that both conditions are set selection conditions does not work since the join semantics would mean that the total number of poor employees in the company, not any department, is set to the *poor_emps* attribute of each department. Instead, a new mechanism is needed here to specify the intended patterns of decomposition among selected sets of objects. In the example above, we want to specify that not only poor employees in a department need to be considered, but that the same thing can be applied on all departments independent of each other. In other words, as long as the selected employees are decomposed or grouped by the department, different groups of objects can be processed in parallel since no interference can occur.

For the purpose above, we found that a natural way to specify the desired semantics is to group several condition elements together which characterizes the desired patterns of decomposition. To illustrate this, the following rule specifies that the number of poor employees of each departments can be computed in parallel.

rule $Count\_All\_Poor\_Employees$ {
  ALL (($d : Department$),
    [$e : Employee :: e.dept == d.name \ \wedge \ e.salary < 10000$])
$\rightarrow$
  $d.poor\_emps = \mathbf{Count}(e*)$
}

The *ALL combinator* groups together several condition elements into an *ALL condition* to denote that any consistent collection of objects and set objects (for set selection conditions) can be considered independent, and therefore all of them can be processed in parallel without worrying about interference.

Formally, a rule r with positive conditions $P_1, \ldots, P_i$, set selection conditions $P_{i+1}, \ldots, P_n$, negative conditions $N_1, \ldots, N_m$, and an ALL condition consisting of positive conditions $AP_1, \ldots, AP_j$, set selection conditions

$AP_{j+1}, \ldots, AP_k$, and negative conditions $AN_1, \ldots, AN_l$, defines a class:

class $Inst\_of\_r$ {
  attributes ($v(P_1) : C(P_1),\ \ldots\ , v(P_i) : C(P_i),$
  $v(P_{i+1})* : C(P_{i+1}),\ \ldots\ , v(P_n)* : C(P_n),$
  $all* : AllClass\_of\_r$)
  methods ($\bigcup_{i=1}^{n} M(C(P_i)) \ \cup\ \bigcup_{i=1}^{k} M(C(AP_i))$)
}

where $AllClass\_of\_r$ is the class

class $AllClass\_of\_r$ {
  attributes ($v(AP_1) : C(AP_1),\ \ldots\ , v(AP_j) : C(AP_j),$
  $v(AP_{j+1})* : C(AP_{j+1}),\ \ldots\ , v(AP_k)* : C(AP_k)$)
  methods ($\bigcup_{i=1}^{k} M(C(AP_i))$)
}

The set of all ground instantiations can be characterized by the set

$$\{\ (g_1, \ldots, g_n, a_1, \ldots, a_k) \mid \exists\ t, u\ (\ t \in Inst\_of\_r\ \wedge\ u \in t.all*\ \wedge\ g_h = t.v(P_h),\ 1 \le h \le i\ \wedge\ g_h \in t.v(P_h)*,\ i+1 \le h \le n\ \wedge\ a_h = u.v(AP_h),\ 1 \le h \le j\ \wedge\ a_h \in u.v(AP_h)*,\ j+1 \le h \le k\ \wedge\ A\theta\ )\ \}$$

where

$$\begin{aligned} \theta &= \{\ v(P_1)/g_1, \ldots, v(P_n)/g_n,\ v(AP_1)/a_1, \ldots, v(AP_k)/a_k\ \} \\ A &= E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge\ E(AP_1)\ \wedge\ \ldots\ \wedge\ E(AP_k)\ \wedge \\ &\quad \nexists\ v(N_1), \ldots, v(N_m),\ v(AN_1), \ldots, v(AN_l)\ (\\ &\qquad E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)\ \vee\ E(AN_1)\ \vee\ \ldots\ \vee\ E(AN_l)\ ). \end{aligned}$$

We note that there is no need to have more than one ALL condition in a rule. It is not difficult to prove that for a rule with multiple ALL conditions, the set of ground instantiations is exactly the same as the rule with all condition elements in each combinator placed under one ALL condition. In other words, if $C_1, \ldots, C_n$ are sets of condition elements, then

$$ALL(C_1) \wedge \ldots \wedge ALL(C_n) \;\equiv\; ALL(C_1 \wedge \ldots \wedge C_n).$$

On the other hand, while nested ALL conditions may seem to provide more expressive power than a single level one, they unnecessarily complicate the semantics. This is certainly against our conciseness criterion. We will show in later sections that the combination of set selection conditions, aggregate operators, and combinators (including the DISJOINT combinator to be introduced next) is versatile enough to express all sources of data level parallelism discussed in Section 4.1.

4.3.4 DISJOINT Combinators

The semantics of both set selection conditions and the ALL combinator imply that any consistent set of objects satisfying the conditions is a valid unit of decomposition and all such units can be processed in parallel. This is desirable when there is no worry about the repetition of selected objects between different sets as in all example rules above. However, when it is possible to have the same object selected to different units, the semantics above may not be exactly what we want as demonstrated in the following rule.

rule $Team\_Employees$ {
($e1 : Employee$ :: $e1.dept == \text{“}research\text{”} \wedge$
$e1.team ==$ **unknown**),
($e2 : Employee$ :: $e2 \neq e1 \;\wedge$
$e2.dept == \text{“}research\text{”} \wedge$
$e2.team ==$ **unknown**),
($e3 : Employee$ :: $e3 \neq e2 \;\wedge\; e3 \neq e1 \;\wedge$
$e3.dept == \text{“}research\text{”} \wedge$
$e3.team ==$ **unknown**)
$\rightarrow$
$e1.team =$ **new** $Team(e1, e2, e3)$,
$e2.team = e1.team$,
$e3.team = e1.team$
}

The purpose of the rule is to team up all employees in the research department such that each employee is in only one team and each team has exactly three employees. We can not use set selection conditions or the ALL combinator here since the same employee could be assigned to multiple teams. The key to the decomposition in this case is the disjointness of selected employees between different sets. Indeed, selecting disjoint sets of data objects for processing is a commonly used strategy in rule-based problem solving. This entitles a new mechanism for specifying the disjoint decomposition, which we call *DISJOINT combinator*. For the example above, the rule below specifies that all teams can be formed at the same time as long as the selected employees are mutually disjoint (i.e., no two sets of selected objects have employees in common).

rule $Team\_All\_Employees$ {

DISJOINT (($e1 : Employee$ :: $e1.dept == \text{“}research\text{”} \wedge$
$e1.team ==$ **unknown**),
($e2 : Employee$ :: $e2 \neq e1 \wedge$
$e2.dept == \text{“}research\text{”} \wedge$
$e2.team ==$ **unknown**),
($e3 : Employee$ :: $e3 \neq e2 \wedge e3 \neq e1 \wedge$
$e3.dept == \text{“}research\text{”} \wedge$
$e3.team ==$ **unknown**))

$\rightarrow$

$e1.team =$ **new** $Team(e1, e2, e3)$,

$e2.team = e1.team$,

$e3.team = e1.team$

}

Similar to the ALL combinator, the DISJOINT combinator is used to combine several condition elements into a *DISJOINT condition* for denoting that objects matching the enclosed conditions are to be decomposed in a disjoint pattern. In other words, for any two instantiations of a rule with DISJOINT combinator, as long as the selected set of objects for the enclosed conditions are disjoint (i.e., no object in common), they are parallel executable. The true power of this mechanism is to reduce combinatorial explosive number of possibly interfering and mostly redundant instantiations into an exact set of all necessary and parallel executable instantiations. As an example, for a rule with n condition elements similar to the rule above, traditional methods will

generate $n!$ instantiations as opposed to only one instantiation using the DISJOINT combinator. The $n! - 1$ redundant instantiations will either have to be detected by interference analysis or removed by meta rules as in the PARULEL [143] approach, both at the cost of excessive run-time overhead.

The formal semantics of the DISJOINT combinator can be defined in a similar way as the ALL combinator except that the disjointness property must be clearly specified. For this purpose, we define an additional notation $\mathbf{Objs}(a)$, for an object a, to denote the set of objects which are values of non-set attributes of a or elements of set-valued attributes of a. The semantics of the DISJOINT combinator can now be defined as follows. A rule r with positive conditions $P_1, \ldots, P_i$, set selection conditions $P_{i+1}, \ldots, P_n$, negative conditions $N_1, \ldots, N_m$, and a DISJOINT condition combining positive conditions $DP_1, \ldots, DP_j$, set selection conditions $DP_{j+1}, \ldots, DP_k$, and negative conditions $DN_1, \ldots, DN_l$, defines a class:

$$
\begin{array}{l}
\textbf{class } Inst\_of\_r \ \{ \\
\quad \textbf{attributes } (\ v(P_1) : C(P_1), \ \ldots \ , v(P_i) : C(P_i), \\
\qquad\qquad\qquad\quad v(P_{i+1})* : C(P_{i+1}), \ \ldots \ , v(P_n)* : C(P_n), \\
\qquad\qquad\qquad\quad disjoint* : DisjointClass\_of\_r \ ) \\
\quad \textbf{methods } (\ \bigcup_{i=1}^{n} M(C(P_i)) \ \cup \ \bigcup_{i=1}^{k} M(C(DP_i)) \ ) \\
\}
\end{array}
$$

where $DisjointClass\_of\_r$ is the class

$$
\begin{array}{l}
\textbf{class } DisjointClass\_of\_r \ \{ \\
\quad \textbf{attributes } (\ v(DP_1) : C(DP_1), \ \ldots \ , v(DP_j) : C(DP_j), \\
\qquad\qquad\qquad\quad v(DP_{j+1})* : C(DP_{j+1}), \ \ldots \ , v(DP_k)* : C(DP_k) \ ) \\
\quad \textbf{methods } (\ \bigcup_{i=1}^{k} M(C(DP_i)) \ ) \\
\}
\end{array}
$$

The set of all ground instantiations can be characterized by the set

$$
\begin{array}{l}
\{ \ (g_1, \ldots, g_n, d_1, \ldots, d_k) \mid \exists \ t, u \ (\\
\quad t \in Inst\_of\_r \ \wedge \ u \in t.disjoint* \ \wedge \\
\quad \forall x \in t.disjoint* \ \ (x \neq u \Rightarrow \mathbf{Objs}(x) \cap \mathbf{Objs}(u) = \emptyset) \ \wedge
\end{array}
$$

$$
\begin{aligned}
& g_h = t.v(P_h),\ 1 \leq h \leq i\ \wedge \\
& g_h \in t.v(P_h)*,\ i+1 \leq h \leq n\ \wedge \\
& d_h = u.v(DP_h),\ 1 \leq h \leq j\ \wedge \\
& d_h \in u.v(DP_h)*,\ j+1 \leq h \leq k\ \wedge \\
& A\theta\ )\ \}
\end{aligned}
$$

where

$$
\begin{aligned}
\theta &= \{\ v(P_1)/g_1, \ldots, v(P_n)/g_n, v(DP_1)/d_1, \ldots, v(DP_k)/d_k\ \} \\
A &= E(P_1)\ \wedge\ \ldots\ \wedge\ E(P_n)\ \wedge\ E(DP_1)\ \wedge\ \ldots\ \wedge\ E(DP_k)\ \wedge \\
&\quad \nexists\ v(N_1), \ldots, v(N_m), v(DN_1), \ldots, v(DN_l)\ (\\
&\qquad E(N_1)\ \vee\ \ldots\ \vee\ E(N_m)\ \vee\ E(DN_1)\ \vee\ \ldots\ \vee\ E(DN_l)\ ).
\end{aligned}
$$

Analogous to the ALL combinator, a single DISJOINT combinator is enough since

$$
\begin{aligned}
& DISJOINT(C_1) \wedge \ldots \wedge DISJOINT(C_n) \\
\equiv\ & DISJOINT(C_1 \wedge \ldots \wedge C_n).
\end{aligned}
$$

Nesting is not recommended either because of the conciseness criterion. We can also have both ALL and DISJOINT combinators in the same rule with well-defined semantics. However, the resulting decomposition patterns seem to be too complex and not at all intuitively appealing for a programmer to conceive and use.

4.3.5 Contexts

Rules are not designed completely independently of each other. A common programming style for rule programs is to decompose the problem solving process into *contexts*. A set of rules is then written for each context to serve the functionality of that context. Opportunities for parallelism are presented when this level of application semantics is taken into consideration [82, 84]. In particular, causally independent contexts can often be processed in parallel. There is a catch if we are to be consistent with our goals. We should avoid providing any procedural-oriented or control-oriented mechanism. Thus, we provide mechanisms that designate the context for which a rule is intended and that specify the causal dependency between different contexts. The actual

control dependency implied by the semantic information above should be left for the language system to derive.

A rule of the form

rule r **in context** T { ... }

denotes that the rule r is designed for context T. All rules designated to the same context are for solving the same subproblem. A *context rule* of the form

$$T \vdash T_1, T_2, \ldots, T_n$$

specifies that context T is causally dependent on contexts $T_1, T_2, \ldots, T_n$ which means, to solve the problem for which T is designed, all the subproblems for which $T_1, T_2, \ldots, T_n$ are designed must be solved first. Note that a context rule specifies a causal dependency rather than an implication. The subproblem represented by context T must still be solved after solving all dependent contexts. The following example denotes that the rule $Raise\_All\_Poor\_Employee$ belongs to the context $Salary\_Adjustment$ which is a collection of rules for adjusting salary.

rule $Raise\_All\_Poor\_Employees$ **in context** $Salary\_Adjustment$ {
($d : Department$),
[[$e : Employee :: e.dept == d.name \ \wedge \ e.salary < 10000$]]
$\rightarrow$
$e.salary = e.salary + e.salary/10$
}

The context rule below specifies that before working on salary adjustment, we must perform profit evaluation and salary survey.

$$Salary\_Adjustment \vdash Profit\_Evaluation,\ Salary\_Survey$$

The context mechanisms proposed above provide a simple way to specify function decomposition of a problem into subproblems. A set of context rules specifies a partial order that must be observed between subproblems to correctly solve the entire problem. Parallelism at the problem solving level can then be exploited by processing independent contexts in parallel.

4.4 Chapter Summary

The parallel structuring mechanisms presented in this chapter are designed centered around the semantics of applications and the characteristics of production system. They are much more powerful than the examples can demonstrate when it comes to real programming. Programmers can easily pick up the ideas and use them effectively. In later chapters, we will show that just this simple set of mechanisms can significantly improve the performance of parallel rule programs.

Chapter 5

Semantic-Based Interference Analysis

An equally important technique in our decomposition abstraction approach is a semantic-based interference analysis technique which derives information about run-time parallel structure from associative relationships among data objects. We present the technique in this chapter.

5.1 A Motivating Example

More often than not, class relationships provide valuable hints on data decomposition patterns that actually happen at run time but are not necessarily clear at design or compile time. In particular, this information can often be used in determining the semantic compatibility (i.e., parallel executability) of instantiations of the same rule or between different rules.

As an intuitive example, consider the following rule from the corporation application domain which is to raise the salary of all under-paid employees in a team.

rule $Team\_Fairness$ {
$(t : Team)$,
$[e : Employee :: e.team == t.name \ \wedge \ e.salary < t.min\_wage]$
$\rightarrow$
$e.salary = t.min\_wage$
}

In general, different instantiations of this rule can not be executed in parallel because the same employee may be a member of different teams. On the other hand, if each team is associated with a unique and disjoint set of employees, then different instantiations will select different teams with disjoint set of employees. Apparently, all such instantiations can be fired in parallel.

In fact, many rule programs are written with similar implicit assumptions but lack of any mechanism to specify them. In the example above, the key

point is on the relationship between instances of the *Team* and the *Employee* class. We call this relationship a *functional dependency* which turns out to be the vital part of our semantic-based dependency analysis technique.

5.2 Functional Dependency

We characterize the idea of functional dependency by defining relations among classes. It helps in understanding the following definitions by comparing the class names, classes, and schemes with the attribute names, domains, and relation schemes in relational database.

Definition 13 (Class Relations and Schemes) *A* class relation scheme *or simply* scheme, *is an ordered set of class names. A* class relation *on a class relation scheme with n class names is an n-ary relation among instances of the corresponding classes.* □

For a class relation A, we denote the scheme on which A is defined by $Sch(A)$. A class relation can be considered as a collection of classes with a certain relationship. Note that an element of an n-ary class relation is an ordered set of n objects, one from each corresponding class in the scheme. An object here can be either a structural object or a set object. For an element $a \in A$ and a scheme $X \subseteq Sch(A)$, the notation $a(X)$ denotes the ordered collection of objects in a which are from classes in X. We note that from the definition above, $a(X) \subseteq a$ and $a(Sch(A)) = a$.

Definition 14 (Functional Dependency) *Let X and Y be the schemes of two class relations R_x and R_y. The* functional dependency

$$X \rightarrow Y$$

holds on R_x and R_y if

1. *Each element in R_x is associated with a unique element in R_y.*
2. *For all a_1, a_2 in R_x and the associated b_1, b_2 in R_y,*

$$a_1 \neq a_2 \Rightarrow b_1 \cap b_2 = \emptyset.$$

□

As an example from the corporation application domain discussed earlier, the functional dependency $\{Team\} \rightarrow \{Employee\}$ holds when each team is associated with a unique and disjoint set of employees.

Since an instantiation can also be considered as an ordered set of objects (one for each positive or set selection condition), a rule r actually defines a class relation whose elements are exactly the set of instantiations of the rule, i.e., **Inst**(r). The scheme of **Inst**(r), denoted by **Scheme**(r), is the ordered set of class name components of positive and set selection conditions of r. For example,

$$\textbf{Scheme}(Team\_Fairness) = \{Team, Employee\}.$$

Definition 15 (Rule Specific Functional Dependency) *Let X and Y be two class relation schemes, and r be a rule. The* rule specific functional dependency

$$X \rightarrow Y \ in \ r$$

holds if both $X \subseteq$ **Scheme**(r) *and $Y \subseteq$* **Scheme**(r) *and for all i, j in* **Inst**(r),

$$i(X) \neq j(X) \Rightarrow i(Y) \cap j(Y) = \emptyset.$$

□

It is *rule specific* because the dependency only needs to hold on all instantiations of r. It may or may not hold on collections of objects that are not instantiations of r.

Definition 16 *Let R be a set of rules, X and Y be two class relation schemes, then*

$$X \rightarrow Y \ in \ R$$

holds if $X \rightarrow Y$ in r holds for each rule r in R. □

Except for borrowing the terminology, functional dependency as defined here is quite different than in databases [92]. In database systems, the notion of functional dependency is defined at the attribute level and is used primarily in the normalization process. We generalize the concept to the class level and use it to identify the parallelism in rule systems. Functional dependencies are considered as specifications of data decomposition across class boundaries, which are shown below to play a crucial role in determining the compatibility between instantiations of the same or different rules.

5.3 Interference Analysis with Functional Dependency

To prove that two distinct instantiations can be fired in parallel, we need to show two things, from Definition 10:

1. The execution of one does not affect the satisfiability of the other, and vice versa.

2. They do not modify the same object.

One of the biggest obstacles in proving validity of these conditions at compile time is the nondeterministic nature of matching. By merely looking at the syntactic structure of rules, it is often the case that we can not completely rule out the possibility of self-interference or interference between different rules. This is the place where functional dependency provide us with the greatest help we need — decomposition. The idea is that if we can determine the disjointness of objects modified by different instantiations, it is very likely that they can be executed in parallel. This section presents the insight and techniques of how this could be accomplished.

Definition 17 *Let r be a rule. The* access set *of r, denoted by* **Access**(r)*, is the set of all class names referenced in the antecedent of r. The* write set *of r, denoted by* **Write**(r)*, is the set of class names with objects that are modified (including creation and deletion) in the rule.* □

Definition 18 (Dominant Set) *Let r be a rule and C be a class relation scheme. C is a* dominant set *of r if:*

1. $C \subseteq$ **Scheme**(r),

2. *for all* $i, j \in$ **Inst**(r) $(i \neq j \Rightarrow i(C) \neq j(C))$. □

A dominant set of a rule is simply a set of class names sufficient to discriminate between different instantiations of the rule.

Theorem 1 (Self Compatibility) *Let r be a rule and A, B, C be three class relation schemes that are subsets of* **Scheme**(r) *satisfying the following conditions:*

1. *C is a dominant set of r and $C \subseteq A$*

2. *$A \rightarrow B$ or $A \rightarrow B$ in r*

3. *$\forall c \in \mathbf{Write}(r)(c \in B \vee c \notin \mathbf{Access}(r))$*

then all instantiations of r are compatible (i.e., parallel executable).

Proof: In any given state, let i and j be instantiations of r such that $i \neq j$.

$$
\begin{array}{lll}
 & i \neq j & \\
\Rightarrow & i(C) \neq j(C) & (* \text{ Condition 1 } *) \\
\Rightarrow & i(A) \neq j(A) & (* \ C \subseteq A \ *) \\
\Rightarrow & i(B) \cap j(B) = \emptyset & (* \text{ Condition 2 } *) \\
\Rightarrow & i(\mathbf{Write}(r)) \cap j(\mathbf{Write}(r)) = \emptyset & (* \text{ Condition 3 } *) \\
\Rightarrow & i \text{ and } j \text{ do not interfere with each other} & (* \text{ Condition 3 } *) \\
\Rightarrow & i \text{ and } j \text{ are compatible} & (* \text{ Definition 10 } *). \ \Box
\end{array}
$$

The central idea of this theorem is that functional dependency implies disjoint decomposition of objects selected by the instantiations of a rule. As long as the objects modified in the consequent belong either to the decomposition or to classes which do not affect the satisfiability of the rule, no instantiations will interfere with each other. There will be examples later in this section. We first generalize this idea to the analysis of interference between multiple rules.

Definition 19 (Partially Mutual Exclusion) *Let p, q be rules and C be a class relation scheme. We say that p and q are* partially mutual exclusive on *C, denoted by $p ><_C q$, if*

1. *$C \subseteq \mathbf{Scheme}(p)$ and $C \subseteq \mathbf{Scheme}(q)$*

2. *For any two instantiations i, j of p and q respectively, $i(C) \neq j(C)$.* $\Box$

Partially mutual exclusion simply means that p and q can not have instantiations containing the same set of objects of classes in C. The simplest and most common case is when C contains a single class referenced in both p and q but tested on disjoint values of the same set of attributes. Since the values are disjoint, p and q can not select the same object in C.

Note that no requirement is placed on selected objects that are not of the classes in C. Therefore, partially mutual exclusive rules may still interfere with each other. However, in many cases, partially mutual exclusive rules can be determined to be parallel executable with the help of functional dependencies as indicated by the following theorem.

Theorem 2 (Pair-Wise Compatibility) *If p, q are two distinct rules, and A, B, C are class relation schemes that are subsets of both* **Scheme**(p) *and* **Scheme**(q) *such that the following conditions are satisfied:*

1. $p >\!\!<_C q$
2. $C \subseteq A$
3. $A \rightarrow B$ *or* $A \rightarrow B$ *in* $\{p, q\}$
4. $\forall c \in \mathbf{Write}(p)(c \in B \vee c \notin \mathbf{Access}(q))$
5. $\forall c \in \mathbf{Write}(q)(c \in B \vee c \notin \mathbf{Access}(p))$

then p and q are compatible and therefore parallel executable.

Proof: In any given state, let i be an instantiation of p and j be an instantiation of q.

	$p >\!\!<_C q$	
$\Rightarrow$	$i(C) \neq j(C)$	$(*$ Definition 19 $*)$
$\Rightarrow$	$i(A) \neq j(A)$	$(*$ $C \subseteq A$ $*)$
$\Rightarrow$	$i(B) \cap j(B) = \emptyset$	$(*$ Condition 3 $*)$
$\Rightarrow$	$i(\mathbf{Write}(p)) \cap j(\mathbf{Write}(q)) = \emptyset$	$(*$ Condition 4 and 5 $*)$
$\Rightarrow$	p and q are compatible	$(*$ Definition 10 $*)$. $\square$

Again, the central idea of this theorem is that as long as objects modified in p and q can be determined as non-overlapping with the help of functional dependency, instantiations of p and q do not interfere with each other.

Even with their general applicability to many cases, the two theorems above are less complicated than they appear. Continuing with our examples in the corporation application domain, if a team is associated with a set of disjoint employees as team members, then the functional dependency

$\{Team\} \rightarrow \{Employee\}$ holds. We note that this semantic information can be easily supplied by the programmer (similar to the identification of key attributes in database systems). With functional dependency and the fact that a team can be uniquely identified by its name, we can immediately determine that all instantiations of the *Team_Fairness* rule can be fired in parallel using Theorem 1.

As another example, the following two rules can be determined to be parallel executable by Theorem 2.

rule *Facilities_Research* {
($t : Team :: t.dept == \text{“}research\text{”}$)
[$e : Employee :: e.team == t.name$]
$\rightarrow$
$e.equipment = \text{“}AXP500X(Alpha)\text{”}$
}

rule *Facilities_Sales* {
($t : Team :: t.dept == \text{“}sales\text{”}$)
[$e : Employee :: e.team == t.name$]
$\rightarrow$
$e.equipment = \text{“}PowerBook\text{”}$
}

In this case, the two rules are partially mutual exclusive on $Team$. With the help of functional dependency, they can be statically determined to be parallel executable.

As simple and natural as it may seem to be, without the knowledge of functional dependency between the $Team$ and the $Employee$ classes, it is very difficult, if not impossible, for a parallelizing compiler or any other static transformation technique to identify the parallelism underlying these rules.

In general, any type of class relationship which implies certain patterns of association or partitioning in the application domain is of great help in the determination of proper decomposition for parallel processing. Mechanisms for expressing these relationships are therefore of great value to the decomposition abstraction process in program development.

We can now refine the definition of program in our framework to include semantic specifications expressed by the decomposition abstraction mechanisms.

Definition 20 (Program) *A program is a quadruple* $(\mathbf{C}, \mathbf{R}, \mathbf{D}, \mathbf{T})$ *where* $\mathbf{C}$ *and* $\mathbf{R}$ *are class and rule definitions, respectively.* $\mathbf{D}$ *is a set of* class relationship definitions *which specify how objects are related to each other between different classes.* $\mathbf{T}$ *is a set of* context rules *that specify the causal dependencies between different contexts.* □

5.4 Chapter Summary

To the best of our knowledge, this chapter presents the first use of functional dependency in the derivation of parallelism. We emphasize that functional dependencies are the direct result of application semantics and program design. They are not artificially made up just for the sake of parallelism. It is therefore very easy and natural for the programmers to supply such information. Actually, from all the benchmarks and our own experience, when functional dependency implies parallel executability, programmers indeed want to fire all the parallel executable instantiations in the first place.

The mechanisms proposed in previous chapter and the semantic-based inference analysis technique presented in this chapter constitute a powerful set of tools for the programmers to exploit application parallelism in production systems. In next chapter, we will show how they can be easily applied on existing sequential rule programs and how to write efficient parallel rule programs from scratch.

Chapter 6

Programming with Decomposition Abstraction

From the programming point of view, the decomposition abstraction approach strongly suggests a declarative programming methodology that significantly simplifies the resulting rules and reduces the need to emulate imperative constructs. In this chapter, we evaluate the quality of the proposed mechanisms and present our experiences on programming with decomposition abstraction.

6.1 The Power of Decomposition Abstraction

To assess the quality of the abstract mechanisms, we evaluate them against the design criteria listed in Section 4.2. The expressive power of the mechanisms is judged by evaluating how well they cover different types of parallelism discussed in Section 4.1.

What vs. How All proposed mechanisms are purely declarative. They characterize core decomposition concepts that can be used to specify various types of parallelism. How they are actually implemented is independent of the decomposition strategies they represent. Programmers can use them entirely at the conceptual and problem level without worrying about any implementation details.

Consistent with Pattern Matching Paradigm A set selection condition element selects objects in exactly the same way as a normal condition element except that all matching objects are selected. An aggregate operator derives information on a set of objects selected by pattern matching. Both ALL and DISJOINT combinators simply group together several condition elements to partition the objects selected by pattern matching in the enclosed condition elements. The pattern matching paradigm is left completely intact. The additional power added by the decomposition abstraction mechanisms includes the abilities to partition the objects selected by pattern matching and to organize the program at the problem solving level.

Conciseness The set of mechanisms is conceptually simple and easy to use in the sense that all of them correspond naturally to the patterns of decomposition commonly used in rule-based systems. When specified using our mechanisms, the decomposition patterns map naturally to parallelization strategies which are then exploited by the system. Programmers are not required to specify any synchronization or communication constraints explicitly. There is no need to worry about interference beyond the level of application semantics.

Versatility This criterion is about the expressive power of the mechanisms. Under the context of rule languages, it can be evaluated by considering how well these mechanisms support the different types of parallelism discussed in Section 4.1. The combination of set selection conditions, aggregate operators, ALL and DISJOINT combinators is capable enough to specify all sources of data level parallelism we found. As indicated by the example rules in previous chapters, they are much richer than many corresponding constructs, such as DOALL loop, DECOMPOSITION, or PARTITION statements in other explicit parallel languages. We conclude that using pattern matching to specify decomposition is much more flexible and powerful than partitioning merely on array index.

For rule level parallelism, the use of class relationships to discover semantic compatibility between rules is very effective, especially when used in conjunction with the existing techniques on syntactic dependency analysis [70, 80, 84, 101, 123]. In particular, most of the parallel executable rules in our benchmark programs can be successfully identified.

For program level parallelism, the context mechanism provides a convenient way to specify any causal relationship or partial order among the problem solving stages. Independent contexts representing independent subproblems can then be identified easily by a topological sort on the partial order.

The set of mechanisms covers all three levels of parallelism found in production systems, and is certainly versatile and expressive enough for our purpose.

Compatibility The mechanisms are designed in such a way that programs not using any of these mechanisms are still perfectly correct programs. In particular, all sequential programs are legal programs. Many sequential programs can be transformed directly into parallel programs as indicated by the examples in Section 4.3 and in [164]. Above all, by simply removing the mechanisms in a parallel program, we return the program to its original sequential form. The set of mechanisms is therefore fully compatibility with sequential semantics.

Effective Implementation The set of mechanisms can indeed be effectively implemented on shared memory multiprocessors. Several alternative implementation strategies and their effectiveness are discussed in Chapter 8 with performance results.

6.2 From Sequential to Parallel

One of the major design goal for the DA mechanisms is to minimize the effort it takes to transform a sequential rule program into a parallel program. From the experience we gained in transforming sequential benchmark programs, we learned that this is a fairly strait forward and natural thing to do if the problem and the design decisions of the sequential programs are well understood. We have collected a set of heuristics to assist the programmer in making the transformation. In this section, we present the set of heuristics with illustrative examples. They are gathered from identifying common programming styles and idioms that exhibit the opportunity for parallel execution. They can also be used in automatic transformation systems such as STAR [46].

6.2.1 Repeatedly Firing Rules

Commonly, rules that fire repeatedly in a sequential program can actually be fired in parallel. This is quite normal under the production system paradigm since there is no notion of loop in sequential rule languages. The only way to transform a collection of data objects by the same process is to fire the same set of rules repeatedly until all the data have been transformed. If the transformation of different data objects do not interfere with each other, all transformations can be performed in parallel. For example, the following rule calculates the GPA for all students.

```
rule  Calculate_GPA  {
    ( s : Student :: s.GPA_calculated == NO )
-->
    s.GPA = calculate_GPA(s),
    s.GPA_calculated = YES
}
```

Note that there is a flag to indicate whether the GPA of a student has been calculated. This is probably the most common way to do the same operation

on a set of data objects. The flag is used to prevent the same object from being processed more than once.

This type of rules can be easily transformed into DA rules using set selection condition. For the example above, this yields

```
rule  DA_Calculate_GPA  {
    [ s : Student :: s.GPA_calculated == NO ]
-->
    s.GPA = calculate_GPA(s),
    s.GPA_calculated = YES
}
```

Sometimes it takes more than one rule to do the job. One rule initializes the loop. One or more rules constitute the loop body and the last one detects the end of the loop. For example, the following three rules do the GPA calculation under a task control.

```
rule  Calculate_GPA_Loop_Init  {
    ( c : CurrentTask == PREVIOUS_TASK )
    ( s : Student :: s.GPA_calculated == NO )
-->
    c.task = CALCULATE_GPA
}

rule  Calculate_GPA_Loop_Body  {
    ( c : CurrentTask :: c.task == CALCULATE_GPA ),
    ( s : Student :: s.GPA_calculated == NO )
-->
    s.GPA = calculate_GPA(s),
    s.GPA_calculated = YES
}

rule  Calculate_GPA_Loop_End  {
    ( c : CurrentTask :: c.task == CALCULATE_GPA ),
   -( s : Student :: s.GPA_calculated == NO )
-->
    c.task = NEXT_TASK
}
```

This type of simulated loop can also be easily transformed into DA rules using set selection conditions. For the example about, the rules can be transformed into a single DA rule as follows.

```
rule  DA_Calculate_GPA  {
    ( c : CurrentTask :: c.task == CALCULATE_GPA ),
    [ s : Student :: s.GPA_calculated == NO ]
-->
    s.GPA = calculate_GPA(s),
    s.GPA_calculated = YES,
    c.task = NEXT_TASK
}
```

Heuristic 1 *If a rule fires repeatedly on a class of objects, try to transform the rule by changing the condition that matches the class into a set selection condition.*

6.2.2 Accumulation Rules

Even though rule languages are primarily for symbolic computation, number crunching is still frequently needed. The most commonly used numerical computation are the aggregation operations such as counting, computing the sum, maximal, minimal, average, etc. Once again, there is no construct in sequential rule languages to do this type of computation directly. Instead, they are "simulated" by a set of rules similar to the simulated loop in last section. For example, to compute the number of strait A students after finishing the calculation of GPA, we can use a counter and a loop.

```
rule  Count_Straight_A_Students_Init  {
    ( c : CurrentTask :: c.task == CALCULATE_GPA ),
   -( s : Student :: s.GPA_calculated == NO )
-->
    c.task = COUNT_STRAIGHT_A,
    count = 0
}

rule  Count_Straight_A_Students_Body  {
    ( c : CurrentTask :: c.task == COUNT_STRAIGHT_A ),
```

```
    ( s : Student :: s.GPA == 4.0 && s.counted == NO )
-->
    count = count + 1,
    s.counted = YES
}

rule  Count_Straight_A_Students_End  {
    ( c : CurrentTask :: c.task == COUNT_STRAIGHT_A ),
   -( s : Student :: s.counted == NO )
-->
    print_count(count),
    c.task = NEXT_TASK
}
```

This is the standard way of doing accumulation in sequential rule languages. A set of rules for accumulation similar to the example above can be transformed into a single DA rule using the set selection conditions and aggregate operators. For the counting rules above, we can use the **Count** operator as the following rule.

```
rule  DA_Count_Straight_A  {
    ( c : CurrentTask :: c.task == COUNT_STRAIGHT_A ),
    [ s : Student :: s.GPA == 4.0 ]
-->
    count = Count(s*),
    print_count(count),
    c.task = NEXT_TASK
}
```

Note that the DA rule is clearly much more intuitively appealing and concise. Other types of accumulation, such as sum, maximal, minimal, average, etc. can be transformed in similar way.

Heuristic 2 *Transform a set of rules for accumulation into a single DA rule by a combination of set selection conditions and aggregate operators. More specifically, change the condition that matches the class of objects to be accumulated into a set selection condition and use appropriate aggregate operators on the selected set in the consequent.*

6.2.3 Nested Rules

By nested rules we means rules that simulate the nested loop in imperative languages. This type of rule is often used when objects of several related classes need to be processed repeatedly. Sometimes the set selection conditions along may not express the exact semantics of the rules. This is where the ALL combinator comes into place. The following example should make the point clear. Suppose we want to count the number of students in all departments.

```
rule  Count_Students_Init  {
    ( d : Department :: d.students_counted == NO )
-->
    d.count = 0
}

rule  Count_Students_Body  {
    ( d : Department :: d.students_counted == NO ),
    ( s : Student :: s.dept == d.name && s.counted == NO )
-->
    d.count = d.count + 1,
    s.counted = YES
}

rule  Count_Students_End  {
    ( d : Department :: d.students_counted == NO ),
   -( s : Student :: s.dept == d.name && s.counted == NO )
-->
    d.students_counted = YES
}
```

This is a standard nested loop over two classes of objects. They can be transformed into a single DA rule as follows. Note that the DA rule does not even need a flag to record whether a student has been counted or not.

```
rule  DA_Count_Students  {
    ALL ( ( d : Department :: d.students_counted == NO ),
          [ s : Student :: s.dept == d.name ] )
-->
```

```
    d.count = Count(s*),
    d.students_counted == YES,
}
```

Heuristic 3 *Transform rules that simulate a nested loop into a single DA rule using the ALL combinator. Enclose all conditions that match the target objects into the combinator and see if set selection conditions and/or aggregate operators need to be used.*

6.2.4 Disjointness Rules

Rules that fire repeatedly on disjoint partitions of data objects are also frequently used in rule-based programs. Using sequential rule languages, the disjointness property must be specified explicitly with some types of inequality tests in the antecedent. As an example, the following rule assigns projects to groups of three students.

```
rule  Assign_Projects  {
    ( p  : Project :: p.assigned == NO ),
    ( s1 : Student :: s1.assigned == NO ),
    ( s2 : Student :: s2.assigned == NO && s2 != s1 ),
    ( s3 : Student :: s3.assigned == NO && s3 != s1
                                        && s3 != s2 )
-->
    p.assigned = YES,
    s1.project = s2.project = s3.project = p.name,
    s1.assigned = s2.assigned = s2.assigned = YES
}
```

This type of rule is inefficient, hard to read and counter intuitive. With DISJOINT combinator, the rule above can be transformed into a much better DA rule.

```
rule  DA_Assign_Projects  {
    DISJOINT ( ( p  : Project :: p.assigned == NO ),
               ( s1 : Student :: s1.assigned == NO ),
               ( s2 : Student :: s2.assigned == NO ),
               ( s3 : Student :: s3.assigned == NO ) )
-->
```

```
    p.assigned = YES,
    s1.project = s2.project = s3.project = p.name,
    s1.assigned = s2.assigned = s2.assigned = YES
}
```

Heuristic 4 *Transform rules that fires repeatedly on disjoint partitions of data objects using DISJOINT combinator. Enclose all conditions involving the disjointness test and remove the test.*

6.2.5 From Secrete Messages to Explicit Contexts

The use of so-called "secret-messages" [118] is a common technique in sequential rule-based programming to emulate procedural control. This technique employs a designated WME (usually called the *goal element* or **context**) to control the phases of execution. By matching different values of the goal element, rules are effectively partitioned into functional components. The flow of execution is controlled by phase changing rules that change the value of the goal element.

This programming style can be easily mapped into DA context mechanism. For example, the following two rules perform the computation and phase change, respectively.

```
rule  Calculate_GPA  {
    ( g : Goal :: g.current_task == CALCULATE_GPA ),
    ( s : Student :: s.GPA_calculated == NO )
-->
    s.GPA = calculate_GPA(s),
    s.GPA_calculated = YES
}

rule  Calculate_GPA_to_Print_Results  {
    ( g : Goal :: g.current_task == CALCULATE_GPA ),
   -( s : Student :: s.GPA_calculated == NO )
-->
    g.current_task = PRINT_RESULTS
}
```

They can be transformed into a single DA rule and a context rule as follows.

```
rule  DA_Calculate_GPA  in_context  CALCULATE_GPA  {
    [ s : Student :: s.GPA_calculated == NO ]
-->
    s.GPA = calculate_GPA(s),
    s.GPA_calculated = YES
}

PRINT_RESULTS  |-  CALCULATE_GPA
```

Heuristic 5 *Transform rules that match goal elements into DA rules that use explicit contexts. Replace phase changing rules with context rules. With DA mechanisms at hand, there should be no need for goal element.*

6.2.6 Remarks

As the examples demonstrated, the DA rules are much more intuitively appealing and efficient than the corresponding sequential rules. The transformation is straightforward and requires only minor changes to the sequential rules. Transforming a sequential program into a DA program is therefore a smooth and painless process. The most important things are to understand the problem domain and the program design. Once the application semantics and the program structure are fully understood, the transformation can be done very quickly.

6.3 Programming in Parallel

Even though the transformation of sequential rules into DA rules is relatively easy, decomposition abstraction is really for parallel structuring and program design. In this section, we discuss issues about programming parallel rule systems using decomposition abstraction. We suggest several steps that leads programmers to the effective use of decomposition abstraction mechanisms.

6.3.1 A Simple Course Scheduling System

We use a course scheduling application as an example for our discussion. The problem is to schedule courses of several departments by assigning instructors, time, and classrooms. Each instructor can teach a number of courses.

An instructor should be assigned no more than three courses. A student is registered in a unique department but can take a number of courses that may or may not be offered by the same department. Each course is two hours long and to be assigned to the time slot of 8:00am, 10:00am, 1:00pm, or 3:00pm during weekdays. Each department building has a number of classrooms. A classroom has a fix capacity. Some classrooms have special equipments that other classrooms may not have. Courses should be scheduled without any conflict on instructor and classroom.

6.3.2 Identify Application Objects

Just as the design of object-oriented software systems, the very first step is to analyze the problem and identify application objects. Any object-oriented analysis and design techniques can be applied here. The key is to think in terms of application objects. No implementation issue need to be worried about at this stage.

In our course scheduling application, any noun that is mentioned more than once in the problem statement is probably a good candidate. We identify the following application objects:

departments Where courses are offered.

courses The targets to be scheduled.

instructors Those who teach.

students Those who learn.

time slots When.

classrooms Where.

Each type of objects should be defined by a proper class definition.

Note that some nouns are better represented as attributes of application objects. For example, the special equipment should be an attribute of the classroom objects rather than a stand along object by itself.

6.3.3 Identify Functional Dependency

The next thing to do is to identify the relationships between application objects. For the purpose of DA programming, we are particularly interested in the functional dependencies between objects of different classes. In general, this is less complicated than it seems. A good starting point is to examine the relationships between each pair of classes. More dependencies may be identified along the way, especially when we start writing rules. Some dependencies may turn out to be less useful but that's not to be worried about at this point. Just list all dependencies that can be identified.

For the sample application, we can identify the following function dependencies.

- A department has a unique set of classrooms in the department building.
- A department is offering a unique set of courses.
- Departments have disjoint sets of students.

As demonstrated here, functional dependencies are easily identifiable in most cases. The dependencies can be specified as follows.

```
{ Department } --> { Classroom }
{ Department } --> { Course }
{ Department } --> { Student }
```

6.3.4 Identify Tasks

With data objects and their relationships clearly identified, the next step is to figure out a solution plan to solve the problem. Dividing a complex problem into subproblems is a powerful technique in both sequential and parallel programming. This can be done by identifying the transformations that need to be applied on the data objects to produce the desired results. Each transformation corresponds to a task that need to be performed to solve the whole problem. A task can be further decomposed into subtasks by identifying the transformations within the task. This process can proceed repeatedly until a subtask is manageable.

In the process of decomposition abstraction, we provide the context mechanism for designing the solution plan as described above. Each task can be represented by a context. Each context can be designed separately. For the purpose of illustration, we adopt a simple solution plan for our example problem. We identify the following possible tasks:

Gather Information: To gather/derive any information that are needed for scheduling courses.

Schedule Special Courses: Some courses need special equipments that are available only in certain classrooms. These courses can be scheduled directly.

Schedule Senior: Schedule courses for senior faculty first.

Schedule Popular Courses: Popular courses should be given higher priority too.

Schedule Regular Courses: Other courses still need to be scheduled, of course.

Print Results: Print the schedule.

Each task can be represented by a context.

The process of task identification normally involves the identification of task structure as well. The task structure should be constructed with parallelism in mind. We discuss the related issues in next section.

6.3.5 Identify Parallelism through Decomposition

The arguably most important step, as performance is concerned, is to identify potential parallelism in the applications. An effective technique is to analyze the problem alone two dimensions: *function decomposition* and *data decomposition*.

Function Decomposition Function decomposition involves the partitioning of the solution plan into subtasks as we did in last section, and the identification of task structure. Task structure is represented by the causal dependencies between tasks. By causal dependencies we mean the natural restrictions on the order of execution between tasks. They can be identified by examining the intended functionality of each individual task and analyzing the necessary conditions for a task to apply correctly.

In the process of decomposition abstraction, the causal dependencies between tasks can be specified by the context rules. For the example problem, it is fairly straight forward to identify the following rules:

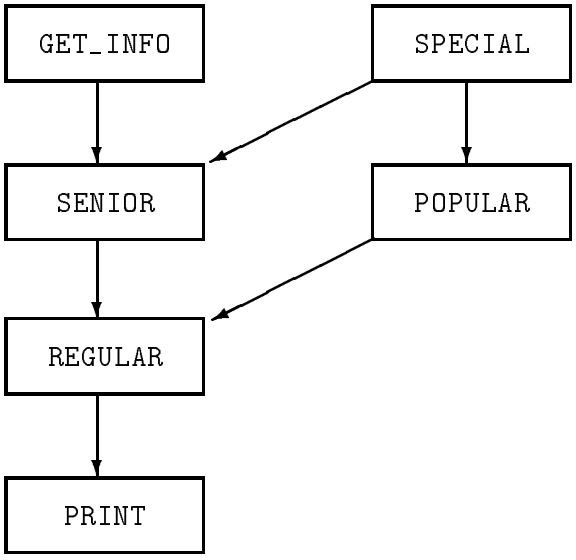

Figure 6.1: Partial Order Derived from the Causal Dependencies between Contexts of the Course Scheduling Problem

```
SINOR     |-  SPECIAL
POPULAR   |-  GET_INFO, SPECIAL
REGULAR   |-  GET_INFO, SPECIAL, SINOR, POPULAR
PRINT     |-  SPECIAL, SINOR, POPULAR, REGULAR
```

Note that some of the dependencies may be redundant. We suggest to list all dependencies that correspond naturally to the application semantics. For example, we can only print the schedule when all types of courses have been scheduled. We therefore specify all of them in the last context rule even though only the REGULAR context is required.

The set of context rules specifies a partial order that must be satisfied when solving the tasks. This is a valuable information since independent contexts (i.e. contexts that are not related to each other in the partial order) can be executed in parallel. Figure 6.1 shows the partial derived from the causal dependencies between contexts of the example problem. Clearly, the GET_INFO and SPECIAL contexts can be executed in parallel. So does the SENIOR and POPULAR contexts.

Data Decomposition Data decomposition involves the partitioning of data objects for data-parallel or SPMD style computation. When rule languages are used in data intensive applications such as expert database systems or data/knowledge based systems, this is usually the most effective way of decomposition.

Data partitioning for SPMD style computation in rule languages is achieved through pattern matching in the antecedents and concurrent execution of multiple instantiations. Rather than trying to decompose data objects directly, it is easier to consider the transformations that need to applied on the data. By identifying the:

- types of objects to be transformed,
- the exact conditions and constraints for selecting the objects, and
- the actions to be performed on the selected objects.

Then each transformation can be represented as a rule. The pattern matching will dynamically decompose the data into desired partitions for processing in parallel. This is much more general as well as adhered to the application semantics (thus easier to specify) than the explicit partitioning of arrays or tables (which is at the implementation level) in most imperative parallel languages. We discuss the design of rules in next section.

6.3.6 Writing DA Rules

After all the steps in previous sections have been performed, writing DA rules is more a specification process than a design process. The rules simply specify the application semantics and identified transformations. There are some guidelines to follow though:

1. Write a set of rules for each context.
2. Within each context, write a rule for each transformation identified in the data decomposition process.
3. Within a rule (or transformation), add a positive condition element for each type of objects to be transformed. Add negative condition elements as required to specify the conditions and constraints for selecting the objects to be transformed.

4. Spell out the actions to be performed in the consequent.

5. According to the application semantics, determine whether a transformation is a SPMD style transformation.

6. Use DA mechanisms to specify the SPMD rules.

We present some rules for the sample application to illustrate the process of writing DA rules. See Appendix A for a complete listing of the course scheduling system.

In the GET_INFO context, we want to gather information for used in the course scheduling process. In particular, we need the number of registrants for each course. This can be considered as transformations on the courses. When computing the number of registrants of a course, the transformation obviously involves two types of objects: Course and Student. The condition for a particular transformation to apply is that the course has not been counted yet. The constraint for a student to be counted is that he/she must have registered for the course. This is represented by an attribute of the student object. Since we want all the students, the corresponding condition element is designated as a set selection condition. The number is counted with aggregate operator in the consequent. A closer look at the transformation suggests that all courses can be transformed at the same time. We therefore use the ALL combinator to specify the SPMD rule as follows. Note that we use `<|` instead of $\in$ as the set membership operator for the latter does not have corresponding key on the keyboard.

```
rule  Count_Registrants  in_context  GET_INFO  {
    ALL ( ( c : Course :: c.registrants_counted == NO ),
          [ s : Student :: c.name <| s.take* ] )
-->
    c.registrants = Count(s*),
    c.registrants_counted = Yes
}
```

The SPECIAL context is special in that some courses need special equipments that are available only in certain classrooms. For example, chemistry lab courses can only be scheduled in the chemistry lab. These courses can be scheduled directly, as in the following rule.

```
rule  Schedule_Special  in_context  SPECIAL  {
    ( t : Time ),
    DISJOINT ( ( c : Course :: c.scheduled == NO
                    && c.special_equip != NULL ),
               ( i : Instructor :: c.name <| i.teaches*
                    && i.assigned < 3 ),
               ( r : Classroom :: t.time <| r.slots*
                    && c.special_equip <| r.equip* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.empty_slots* = r.empty_slots* - t.time
}
```

Yes. A single rule is enough. For each special course, we select an instructor who teaches that course and not yet assigned more than 3 courses, a classroom with proper equipment, and a time slot that the classroom is still not occupied. Note that it is not necessary to enclose the time object in the DISJOINT condition since two courses can be assigned to the same time slot as long as the instructor and classroom are not the same. We also assume that none of the senior instructors teach special courses since these are mostly lab courses.

For the SENIOR context, we schedule just one course for whatever a senior instructor wants to teach. The number of registrants must not exceed a fixed limit called LIMIT.

```
rule  Schedule_Senior  in_context  SENIOR  {
    ( t : Time ),
    DISJOINT ( ( c : Course :: c.scheduled == NO
                    && c.registrants < LIMIT ),
               ( i : Instructor :: i.is_senior == YES
                    && c.name <| i.teaches*
                    && i.assigned < 1 ),
               ( r : Classroom :: t.time <| r.slots* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time
}
```

The POPULAR context schedules courses with number of registrants exceeding a threshold named THRESHOLD. This can be designed similar to the SENIOR context. Note that LIMIT is always smaller than THRESHOLD. So there is no worry about assigning a senior instructor to a popular course.

```
rule  Schedule_Popular  in_context  POPULAR  {
    ( t : Time ),
    DISJOINT ( ( c : Course :: c.scheduled == NO
                    && c.registrants > THRESHOLD ),
               ( i : Instructor :: i.is_senior == NO
                    && c.name <| i.teaches*
                    && i.assigned < 3 ),
               ( r : Classroom :: t.time <| r.slots* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time
}
```

Now rest of the courses can be assigned simply by pattern matching. Again, we need only one rule for the REGULAR context.

```
rule  Schedule_Regular  in_context  REGULAR  {
    ( t : Time ),
    DISJOINT ( ( c : Course :: c.scheduled == NO ),
               ( i : Instructor :: i.is_senior == NO
                    && c.name <| i.teaches*
                    && i.assigned < 3 ),
               ( r : Classroom :: t.time <| r.slots* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time
}
```

Finally, we print the result of the course scheduling. The printing must be done sequentially. Otherwise the output would be unreadable.

```
rule  Print_Result  in_context  PRINT  {
    ( c : Course :: c.scheduled == YES && c.printed == NO )
-->
    print_schedule(c),
    c.printed == YES
}
```

6.4 Chapter Summary

Decomposition abstraction and the mechanisms we proposed raise the level of abstraction from implementation level to application level. Rules are much easier to write since they are closer to application semantics. The number of rules tend to be significantly less than corresponding sequential program. In writing the rules, however, it is still the programmer's responsibility to ensure the correctness of the specified semantics. For example, if a functional dependency is specified, it must hold throughout the entire execution of the whole program. If an execution of a rule may violate the dependency, any parallel structure derived form the dependency may no longer hold. These type of errors can be treated as programming errors.

On the other hand, it is possible to detect violation of specified semantics before the program execution. One of our primary future research directions is to develop theories and techniques for consistency checking and correctness validation of DA programs. This is a necessary component for the decomposition abstraction approach to be a complete parallel programming paradigm.

In this chapter, we have demonstrated how decomposition abstraction can be used in both the transformation of sequential programs and the development of parallel programs. All the transformation and development processes are the direct results of our experiences in dealing with sequential benchmark programs and writing parallel programs from scratch. Another direction of our future research is to formally compare the programmability, complexity, and effectiveness of different approaches including static transformation [101], meta-rule programming [143], source-to-source transformation [119], rule rewriting [46], and decomposition abstraction.

Chapter 7

Performance Assessment

With decomposition abstraction, programmers can specify the parallelism inherent in the problem domain, thereby increasing the concurrency that can be exploited by the language system. However, just like any explicit parallel language, the actual performance gain depends heavily on the implementation strategy. This is especially the case in rule systems because rules tend to have large variation of processing requirements [56] and irregular patterns of decomposition. Systems that fire multiple instantiations in parallel tend to incur large run-time overhead [111]. Granularity control and proper scheduling strategy are therefore of crucial importance to a multiple rule firing production system with decomposition abstraction mechanisms. In this chapter, we employ a unique software engineering technique to evaluate the performance of several alternative implementation strategies on a parallel rule execution engine. The technique is unique in that it enables rapid system development and assessment without the possible inaccuracy of traditional simulation as well as the high cost of full-fledged system implementation.

7.1 A Parallel Rule Execution Engine

The central idea of our performance assessment technique is to be as close as possible to the real execution environment. For such purpose, we developed a *parallel rule execution engine* that actually executes multiple rule instantiations in parallel on our target machine, the Sequent Symmetry shared memory multiprocessor. A *work load generator* generates work from sequential execution trace files. This approach is different from traditional simulation technique in several respects.

- WME's are actually added, removed and modified. All rule actions are executed as they would be in a real parallel environment. In other words, the time accumulated is the actual execution time.
- All communication and synchronization operations needed to maintain the correct parallel execution are actually performed when firing multiple

rule instantiations. This provides us with accurate measurement of the scheduling overhead.

- With the same set of data, the parallel rule execution engine will terminate with exactly the same result as in a sequential execution except, of course, the execution time. With this, we can be sure that the parallel execution is correct.

The only important factor which is not accounted for is the time spent on matching. To include matching would make it a real parallel inference machine, which will take much longer to develop before any experiment can be done on it. With the already existing good results on both sequential and parallel algorithms for matching [87, 97, 100], the performance of the real system may be even better with parallel matching since the effect is at least additive, if not multiplicative.

Figure 7.1 depicts the simulation method we developed. The parallel rule execution engine is a set of programs built on top of a C++ based object-oriented light weight thread package called PRESTO [13, 41]. A sequential rule program[1] and its execution trace are translated into a PRESTO program which is the parallel version of the program integrated with the PRESTO run-time libraries. More specifically, each data definition[2] is translated into a C++ class definition which is called a *WME class*. Each rule definition is translated in to a C++ function which is termed a *rule function*. The work load generator generates a call to a rule function (with the WME's accessed by the rule as arguments) for each rule firing in the execution trace. Successive rule firings are translated into a sequence of asynchronous thread invocations to execute the corresponding rule functions if the rules are compatible as determined by the decomposition abstraction specification. Synchronization points are inserted whenever it is necessary to synchronize incompatible rules. With different scheduling strategies to be discussed later, one or more rule function calls can be assigned to a thread for executing sequentially within the thread.

To experiment with the effect of granularity of rules on system performance, a controllable dummy loop (a FOR loop with empty body and a

[1] At this moment, the parallel rule execution engine takes only OPS5 programs. However, the same approach can be applied to any sequential rule language.

[2] In the case of OPS5, it is an *element class* defined by the **literalize** command.

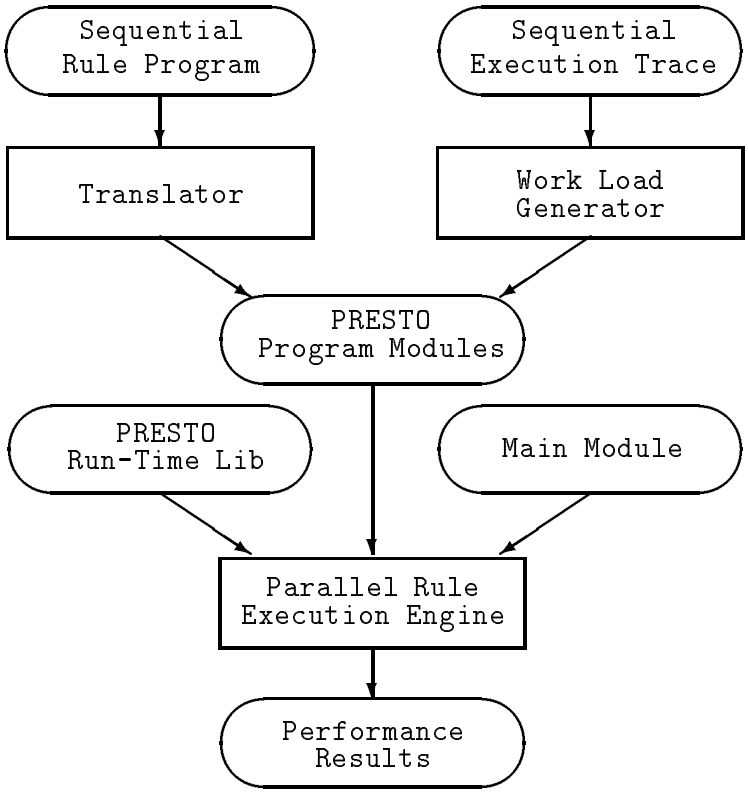

Figure 7.1: Parallel Rule Execution Engine and the Simulation Method.

parameter controlling the number of iterations) is added to the action part of each parallel rule in the PRESTO program. The time to execute the action part of a rule can then be controlled by varying the number of iterations for the dummy loop. Also, by varying the number of threads created for processing parallel instantiations and the granularity of work (number of instantiations) assigned to each thread, we have measured the performance of four alternative scheduling strategies.

- **Strategy 1: Maximal Parallelism** Create a new thread for each parallel instantiation.
- **Strategy 2: Fixed Granularity** Create as many threads as required except that each thread is assigned a fixed number of instantiations to execute sequentially. This is to increase the granularity of work assigned to each thread so as to reduce the total number of threads in comparison with Strategy 1.
- **Strategy 3: Supervisor/Worker** Create a fixed number of worker threads and a scheduler thread. The scheduler thread keeps dispatching instantiations, one instantiation for each worker, as long as there are idle workers. If no idle worker exists, the scheduler executes the instantiation itself.
- **Strategy 4: Supervisor/Worker with Packing** Same as Strategy 3 except that each time an idle worker is given a fixed number of instantiations to work on sequentially. The number is called the *grain size* [78] whose purpose is to increase the granularity of work assigned to a worker thereby reducing thread management and scheduling overhead.

Even though previous work on task scheduling shows that the performance of the supervisor/worker model can be significantly affected by the communication cost [38], this model can still benefit from parallel execution when the computation cost is high enough. We have demonstrated this in the set of experiments on rule granularity.

To evaluate the effectiveness of different scheduling strategies, we collect the performance results from the execution of three benchmark programs drawn from the Texas OPS5 Benchmark Suite [16] and listed in Table 7.1. All three programs are executed with increasing numbers of processors on different problem sizes and different grain sizes. We also compare their relative performance, scalability, as well as sensitivity to granularity change. The actual time to execute the dummy loop for various number of iterations are also measured and listed in Table 7.2.

Program	No. Rules	Description
LIFE	16	A simulation program implements Conway's LIFE.
WALTZ	33	A constraint satisfaction problem using Waltz's algorithm for scene labeling [158].
MANNERS	8	A combinatorial search problem for seat assignment.

Table 7.1: Benchmark programs used in the simulation.

No. Iterations	Time (ms)
0	0.00
1000	2.57
3000	7.64
5000	12.71
7000	17.78
10000	25.38
20000	50.75

Table 7.2: Time to execute the dummy loop for specified number of iterations.

7.2 The Benchmark Programs

Identifying the characteristics of an application is of crucial importance in the process of decomposition abstraction. In this section, we analyze the concurrent behavior of each benchmark program and point out key issues to the successful application of proposed mechanisms.

7.2.1 MANNERS

MANNERS was derived from an example program in [76] which employed a combinatorial search for solving a seat assignment problem among a number of guests. The seats must be assigned such that neighbors are of opposite sex and share at least one common hobby. This simple program, containing just 8 rules, is a very good test program for evaluating the effectiveness of a parallel production system. It consists of a hot spot rule that fires repeatedly and consumes over 90% of sequential execution time for problem size of 64 guests or more. The larger the problem size, proportionally more time is consumed by this rule, which is used to maintain partial solutions. Although all instantiations of this rule can be fired in parallel, it is very difficult to recognize this opportunity by pure syntactic techniques at compile-time without the

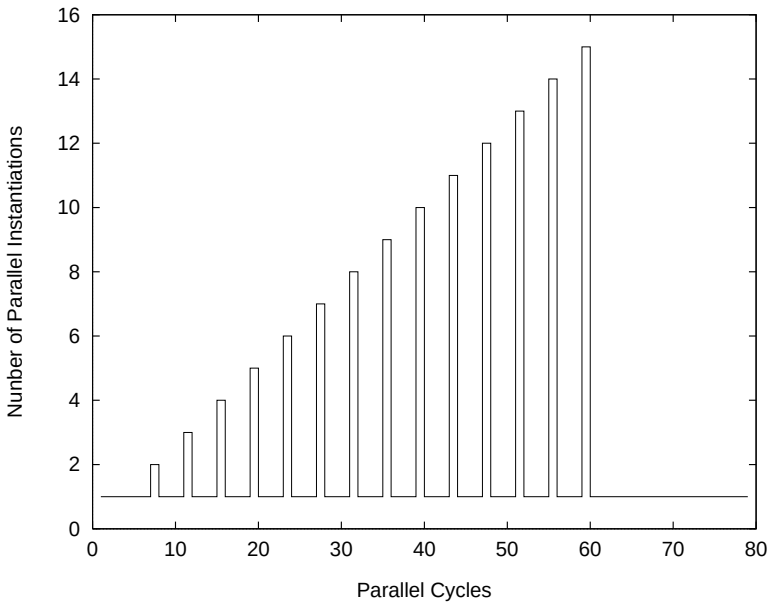

Figure 7.2: MANNERS16 Concurrency Profile.

information provided by our mechanisms. The concurrency exhibited is also exceptional. It is regular but not evenly distributed. The execution starts with a very low degree of parallelism which then gradually increases toward the end. More specifically, the program starts with only 2 instantiations that can be executed in parallel, then 3 instantiations, then 4, 5, ... etc. Figure 7.2 is the concurrency profile of MANNERS with 16 guests. This is highly challenging since a parallel production system must not only detect the hot spot rule, but also exploit effectively a rather peculiar pattern of parallelism.

7.2.2 LIFE

LIFE is a simulation program that simulates the existence of bacteria in a rectangular grid of cells for a specified number of generations. Whether a cell stays alive across a generation is determined by the number of neighbors it has. A living cell is born in an empty cell if it has exactly 3 neighbors. Since all decisions can be made locally, LIFE exhibits a high degree of data level parallelism. However, the available concurrency has not been effectively exploited in previous research. The difficulty of detecting it by syntactic analysis alone is again the key reason. Another probably even more important reason is that a parallel production system must have the ability to perform set-oriented and aggregate operations to exploit the available concurrency. Figure 7.3 is the concurrency profile of a 10x10 LIFE execution trace without showing the

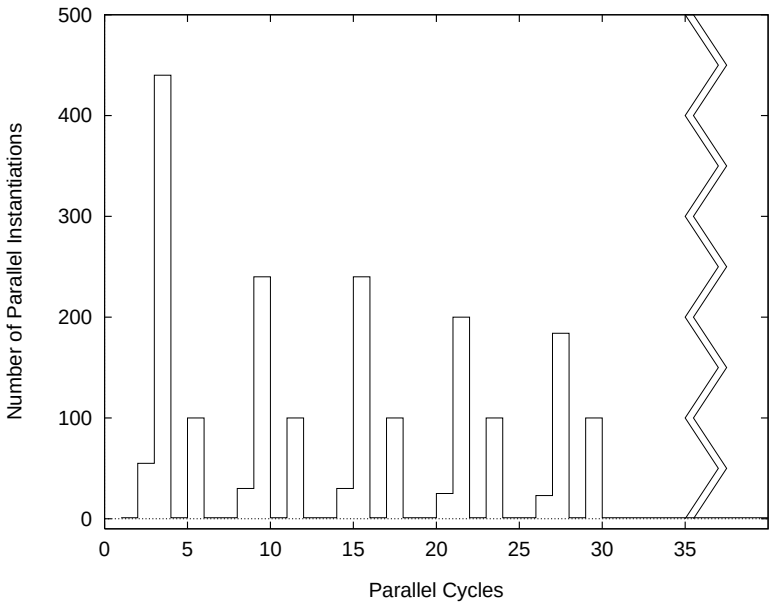

Figure 7.3: LIFE10 Concurrency Profile (with Sequential Printing at the End trimmed by the zigzag line).

sequential printing at the end of execution. Because of the rather evenly distributed pattern of parallelism, keeping processors busy doing useful work at all time is the primary issue.

7.2.3 WALTZ

The frequently studied WALTZ program is also selected here to serve both as a test program to evaluate the effectiveness of our mechanisms and as a benchmark program to compare our results with others. This is a constraint satisfaction problem that implements Waltz's algorithm for labeling of line drawing scenes. The algorithm propagates labels based on local decisions and therefore exhibits both SPMD- and MIMD-style of parallelism (i.e., parallel instantiations of the same or different rules working on different parts of the scene). The available parallelism is again quite high as depicted in Figure 7.4, which is the concurrency profile of a 10 regions execution trace with sequential printing of results at the end excluded.

Because, syntactically, a number of similar constraints appear in many rules, the rules are highly interfering with themselves. Most of the existing parallel production systems can only handle this at run-time resulting in excessive overhead. However, the constraints are disjoint, most of the run-time overhead is superfluous. The functional dependency declaration and DISJOINT

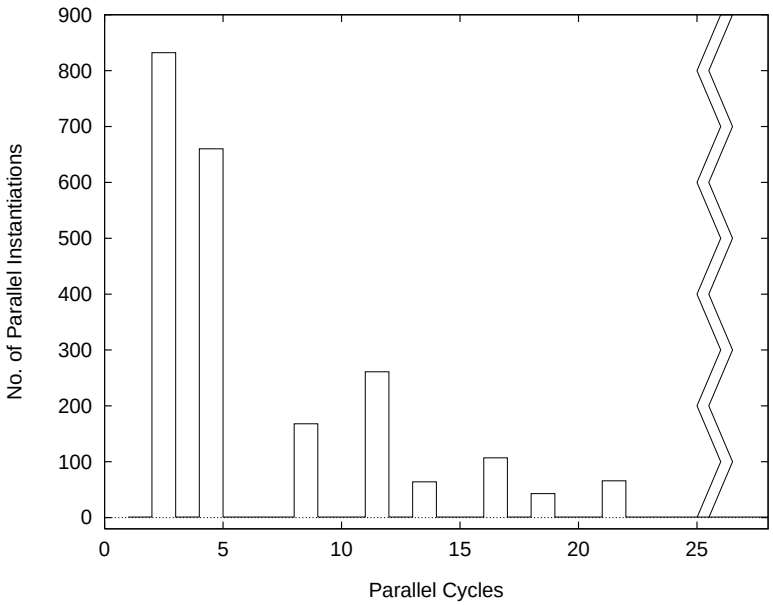

Figure 7.4: WALTZ10 Concurrency Profile (with Sequential Printing at the End trimmed by the zigzag line).

combinators express this to reveal a compile-time parallel structure. A system capable of forming disjoint partitions of consistent data objects can then effectively exploit this available concurrency without the overhead of run-time interference detection.

7.3 Results

In this section, we demonstrate and analyze the performance results of three OPS5 benchmark programs on our parallel rule execution engine. Experiments were conducted on various dimensions affecting the selection of implementation strategies. The speedup is measured against the execution time of the execution engine with a single processor instead of an optimized uniprocessor OPS5 compiler such as OPS5c [103] because the former is two to three orders of magnitude faster than the latter. The difference is primarily due to the additional matching performed by the uniprocessor compiler. As a remedy to this lack of match phase, we artificially increase the per rule processing time using dummy loop as described earlier.

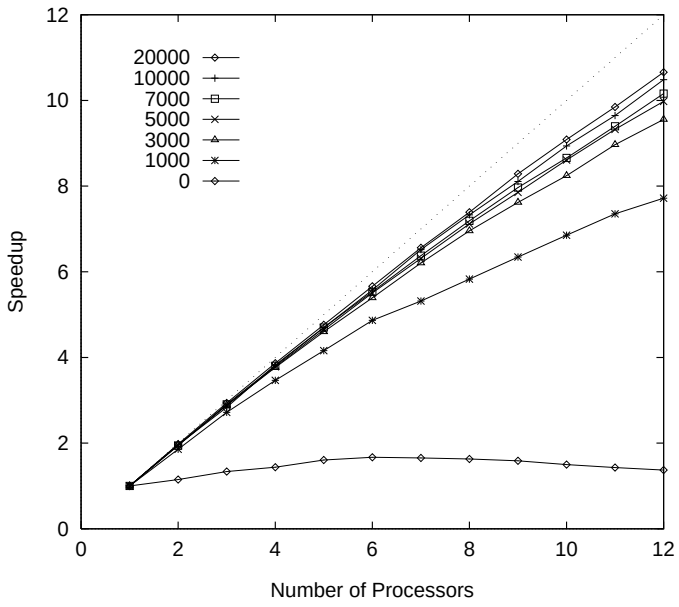

Figure 7.5: MANNERS512 Speedup with Varying Granularities.

7.3.1 The Effect of Rule Granularity

To understand the effect of rule granularity (i.e., the time to process a rule) on the system performance, we select the supervisor/worker with packing scheduling strategy while varying the granularity of a rule by changing the number of iterations in the dummy loop. Figure 7.5 shows the results on MANNERS512 (i.e., 512 guests). The performance improves significantly with larger granularity. Nearly ideal speedup is achieved when the granularity per rule is increased to 20000 (i.e., 50.75ms). As a comparison, the average cycle time of the same program and data set running under OPS5c on a much faster CPU (SUN4 workstation vs. the Intel 80386 on Sequent Symmetry) is about 210ms. This implies that the overhead of scheduling and thread management is very low, and as long as we can keep it low in a real implementation it is very likely to get even better results since the granularity per rule is expected to be much higher than 50ms when matching is included.

Figure 7.6 and Figure 7.7 are the results of similar experiments on LIFE (40x40) and WALTZ (30 regions), respectively. In both cases, performance improvement is observed with increasing granularity.

Among all three test cases, LIFE achieves the highest speedup with granularity 20000. This is plausible because the run-time behavior of LIFE exhibits the highest and the most regular pattern of concurrency as depicted

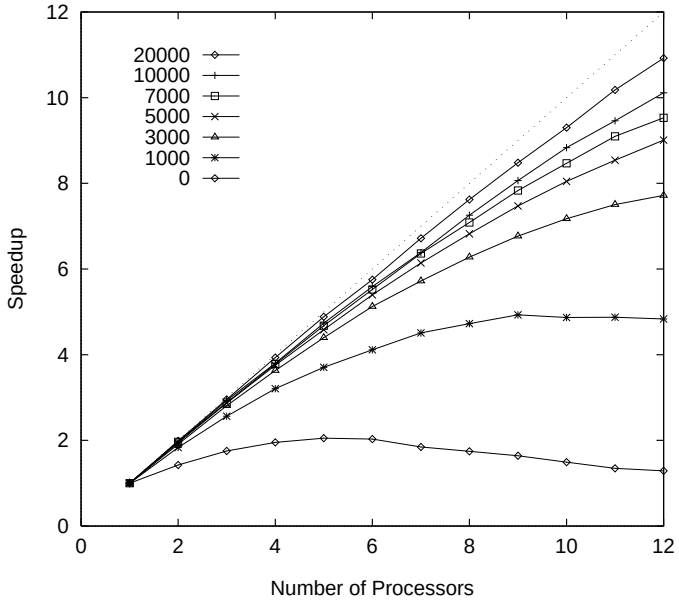

Figure 7.6: LIFE40 Speedup with Varying Granularities.

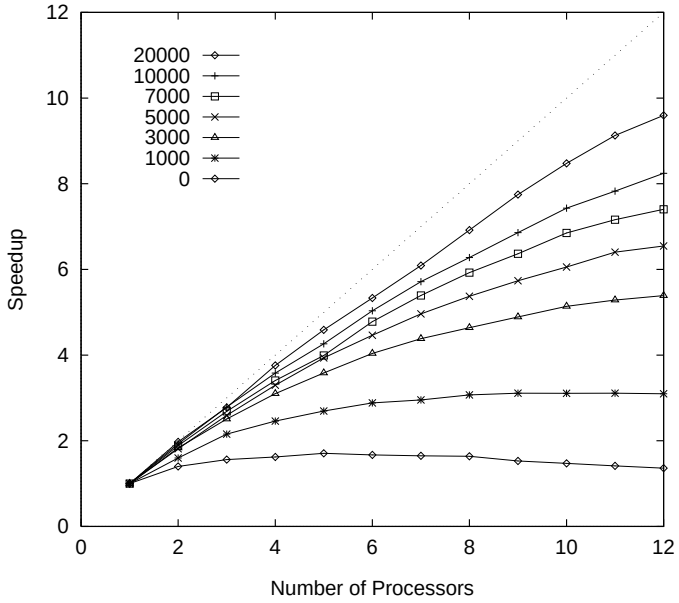

Figure 7.7: WALTZ30 Speedup with Varying Granularities.

earlier in Figure 7.3. On the other hand, WALTZ requires larger granularity to achieve the same level of performance. We were puzzled by this unexpected result at first since from the characteristics of the Waltz's algorithm, there should not be that much difference. Later on we found that the available parallelism of a WALTZ program execution depends heavily on the data set (i.e., the scene to be label). The data generator we use (and used by other researchers as well) introduces a sequential factor that severely restricts the available parallelism. The generated scene consists of two arrays of rectangular blocks growing linearly according to the given problem size parameter. This linear factor contributes to the performance difference between WALTZ and the other two programs. We plan to develop a new data generator that generates scenes without this linear factor.

7.3.2 Scalability: The Effect of Problem Size

An important criterion when evaluating the effectiveness of a parallel system is its scalability. When the available parallelism increases, a parallel processing system must be able to effectively exploit it and achieve better performance. For the three benchmark programs, a common characteristic is that available parallelism increases with the problem size. Therefore, we tested our mechanisms and system on three programs with increasing problem sizes where the problem size to MANNERS, LIFE and WALTZ are the number of guests to be assigned, the grid size, and the number of regions[3], respectively. All programs were tested under the supervisor/worker with packing scheduling strategy. The grain size was set to 5 with rule granularity fixed at 20000 (i.e., 50.75ms).

Figure 7.8 illustrates the performance results of MANNERS on different problem sizes and graphically displays scalable speedup of our scheme. Figure 7.9 and Figure 7.10 are the results of similar experiments on LIFE and WALTZ. For all three programs, we achieved near linear speedup when problem size was large enough. The speedup achieved on smaller problems is lower because the available parallelism is not enough to keep all processors busy. When problem size becomes larger, processor utilization increases and so does the speedup achieved. The trends also indicate that the system can achieve even better results with problem size larger than the largest size conducted in the simulation.

[3]In the scene generated by the data generator for WALTZ, a region consists of 72 line segments.

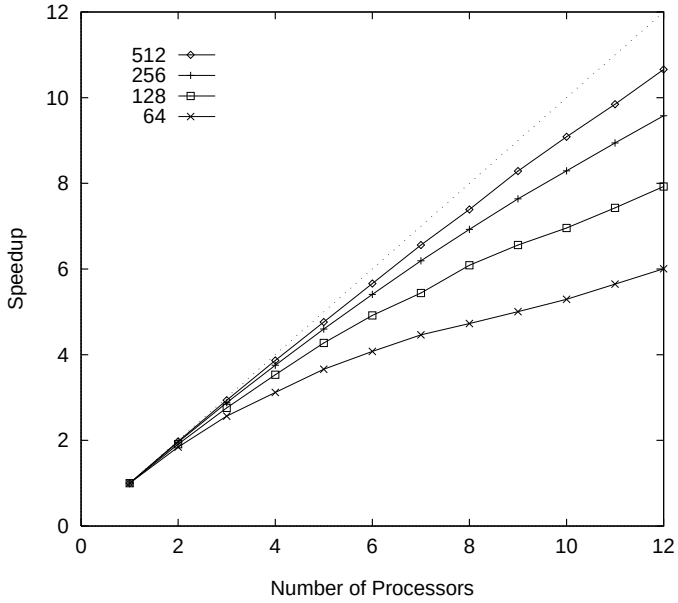

Figure 7.8: MANNERS Speedup on Different Problem Sizes.

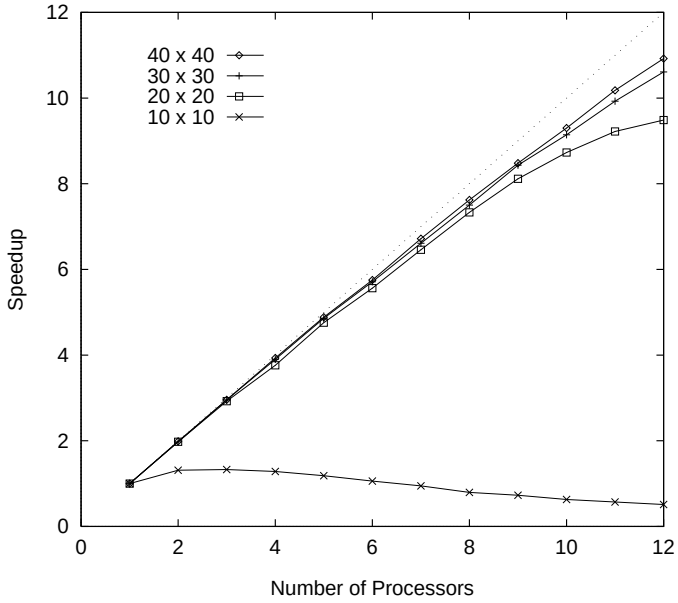

Figure 7.9: LIFE Speedup on Different Problem Sizes.

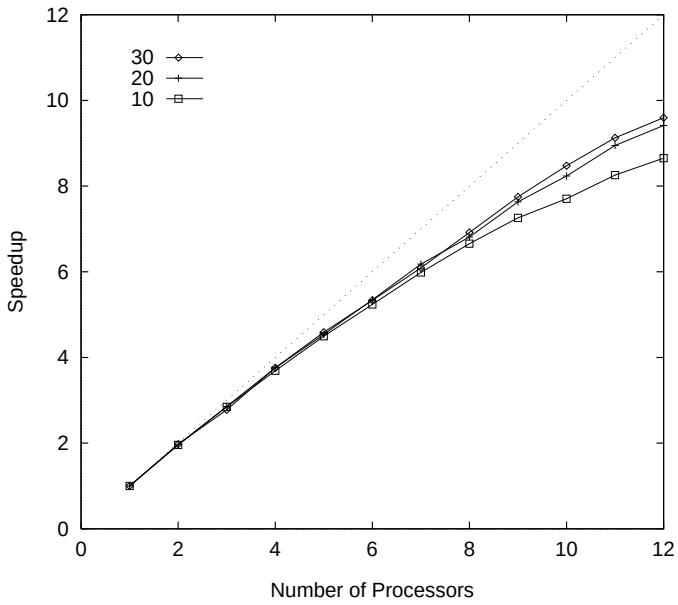

Figure 7.10: WALTZ Speedup on Different Problem Sizes.

7.3.3 Controlled vs. Unrestricted Parallelism

Too much water drowned the miller. If the available parallelism is not exploited appropriately, the benefit of parallel processing can easily be overwhelmed by the scheduling and synchronization overhead. In our case, since the embedded parallelism in the application is fully expressed, the key issue comes down to efficiently processing the collection of parallel instantiations on available computation resources. For a thread-based implementation like ours, this issue manifests itself in a tradeoff between parallel processing of as many instantiations as possible and controlling the number of concurrent threads. If a new thread is created for each parallel instantiation (i.e., Strategy 1), we get maximal parallelism on the one hand but highest thread management overhead on the other hand. Using the supervisor/worker scheduling strategy, the number of threads is fixed but the communication and synchronization cost increase because of the need to partition and dispatch parallel instantiations to the worker threads.

To understand the effect of thread management overhead on system performance, we compare the results between applying Strategy 1 (maximal parallelism) and Strategy 4 (supervisor/worker with packing). Figure 7.11 is a 3-D display of two sets of experiments on WALTZ10. The timing curves on the base plane are the execution time while the B-spline surfaces are to demonstrate

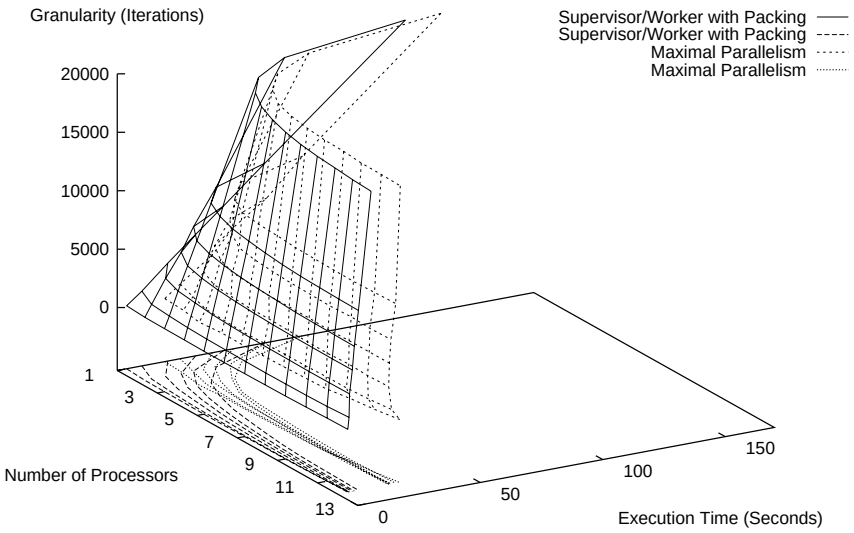

Figure 7.11: 3-D Display of Controlled vs. Maximal Parallelism on WALTZ10.

the performance differences. It is quite evident that supervisor/worker with packing outperforms maximal parallelism by a substantial margin. Figure 7.12 presents the results of similar experiments on LIFE30. The difference is smaller but still perceptible. This suggests that throttled parallelism is much better than unrestricted parallelism.

Just when we expect to observe a similar type of performance difference on MANNERS program, it does not happen to be the case. Figure 7.13 is the 3-D graph of the similar experiments as above. The performance of maximal parallelism is not only comparable to that of supervisor/worker with packing, it actually performs better when the granularity of rules and the number of processors increases. To more clearly show this phenomenon, we demonstrate the timing curves on a 2-D diagram in Figure 7.14. A closer look at the problem reveals a peculiar pattern of concurrent behavior of the MANNERS program. As depicted in Figure 7.2, the number of parallel executable instantiations is quite small in the early stage of the execution. When packing is applied, the available parallelism is left unexploited while under the maximal parallelism strategy these instantiations are always processed in parallel.

As a summary, the supervisor/worker strategy which creates only a limited number of threads is, in general, better than the maximal parallelism

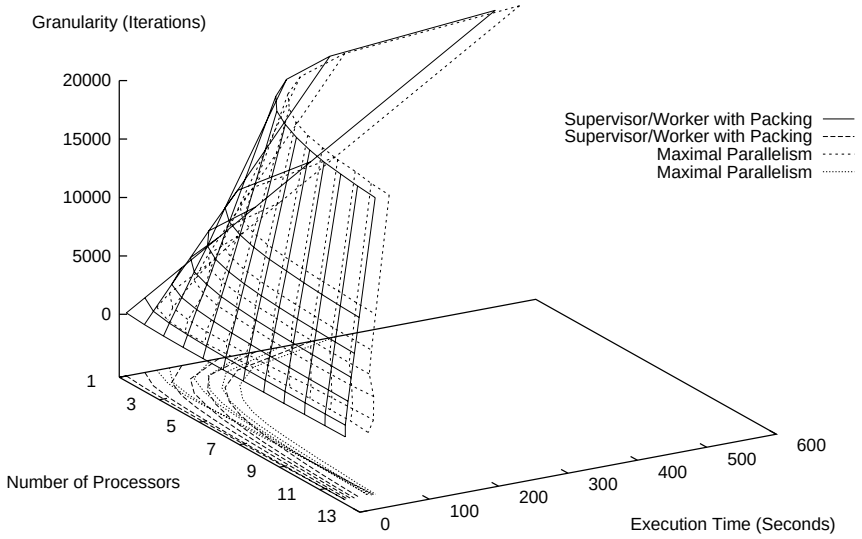

Figure 7.12: 3-D Display of Controlled vs. Maximal Parallelism on LIFE30.

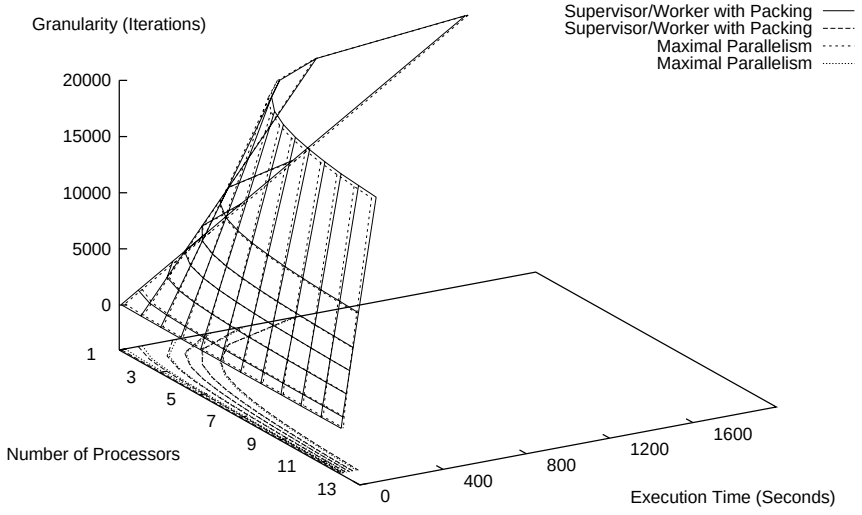

Figure 7.13: 3-D Display of Controlled vs. Maximal Parallelism on MANNERS256.

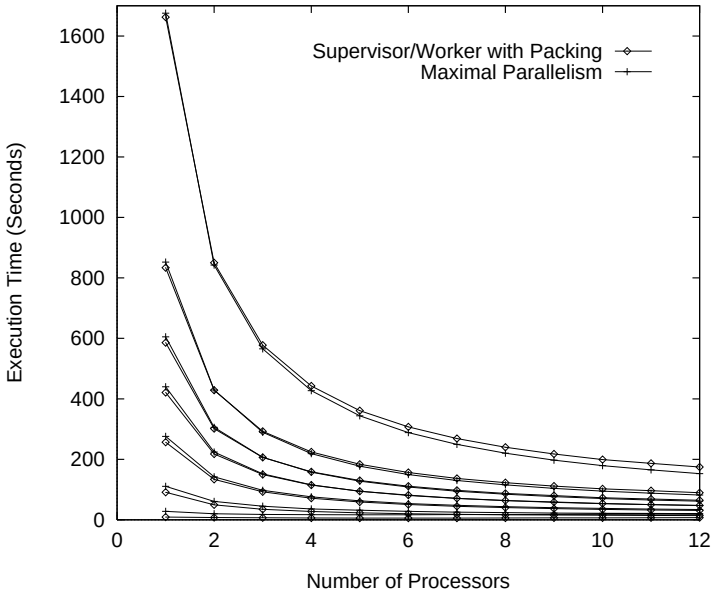

Figure 7.14: 2-D Display of Controlled vs. Maximal Parallelism on MANNERS256.

strategy which creates as many threads as the number of parallel instantiations. However, the actual performance gain still depends on the characteristics of the underlying program. When the degree of concurrency in an application is low, the maximal parallelism strategy is likely to be better.

7.3.4 The Effect of Grain Size

In last section, supervisor/worker with packing appeared to be the winner in overall performance. The question of determining appropriate grain size follows. It is unlikely that a single grain size is optimal for every program. We need, at a minimum, to determine if performance as a function of grain size is well behaved enough to offer a system default. We test the benchmark programs with grain sizes of 1 (i.e., no packing), 5, 10, and 20. Each one of them is tested with a fixed level of rule granularity. To understand the correlation of packing with respect to rule processing time, the same set of experiments is carried out with different levels of rule granularity.

Figure 7.15 is the results on MANNERS512 with rule granularity set to 1000. We can observe quite clearly that packing is always better than no packing at this level of granularity. A grain size of 5 provides the best performance. On the other hand, with the granularity of a rule raised to

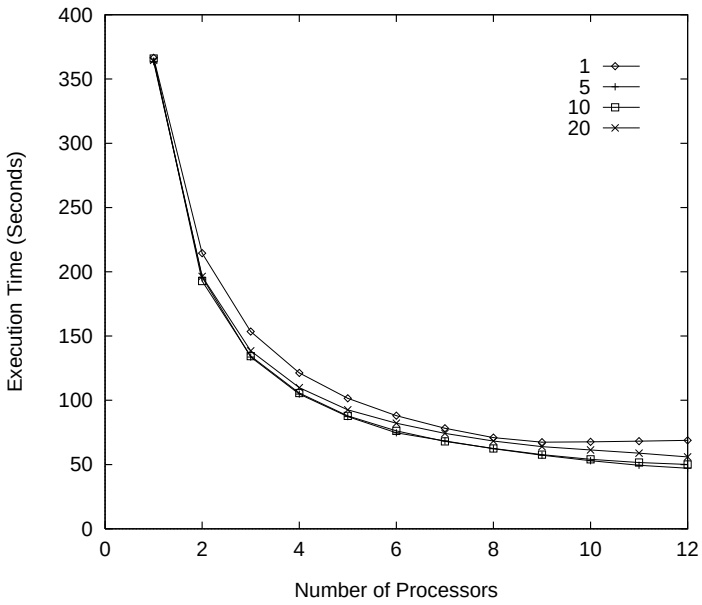

Figure 7.15: MANNERS512 Execution Time on Different Grain Sizes (Rule Granularity = 1000).

20000, the best performance is obtained when packing is not applied as shown in Figure 7.16. Similar results can be observed on both LIFE (Figure 7.17, Figure 7.18) and WALTZ (Figure 7.19, Figure 7.20) except that grain size of 5 is not necessarily a clear winner over other grain sizes.

7.4 Summary and Analysis

In summary, reducing the granularity of rules improves the results we get from packing. This is primarily because of the reduction in thread management, communication, and synchronization overhead. However, when the average granularity of the rules or the grain size becomes larger, the loss of parallelism offsets the benefit of packing. In general, packing with grain size 5 provides the best performance when the rule granularity is smaller than 5000. When the granularity is larger than 5000, it is better to do without packing. In other words, packing is good for cases where the per rule scheduling overhead is comparable or larger than that of the granularity of rule. This result suggests a dynamic scheduling strategy with either user-specified granularity assignment to each rule or an automatic estimation done by the system. We are investigating this issue in our real implementation of an object-based parallel rule language equipped with decomposition abstraction mechanisms.

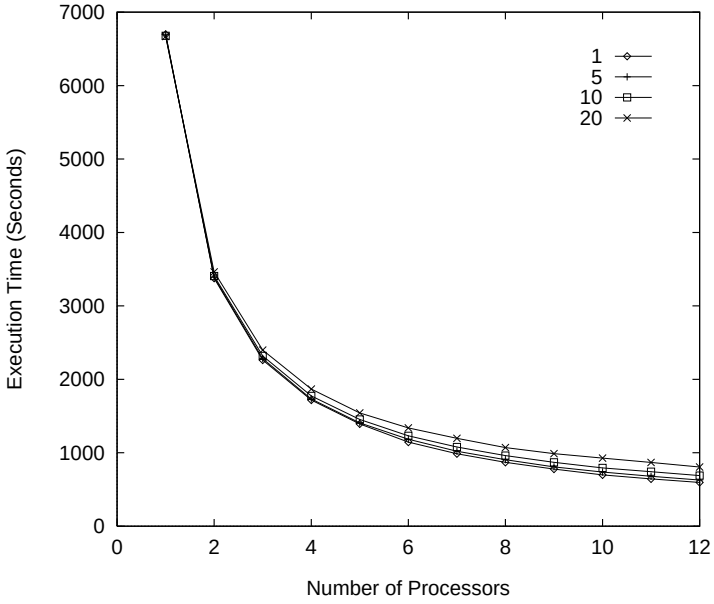

Figure 7.16: MANNERS512 Execution Time on Different Grain Sizes (Rule Granularity = 20000).

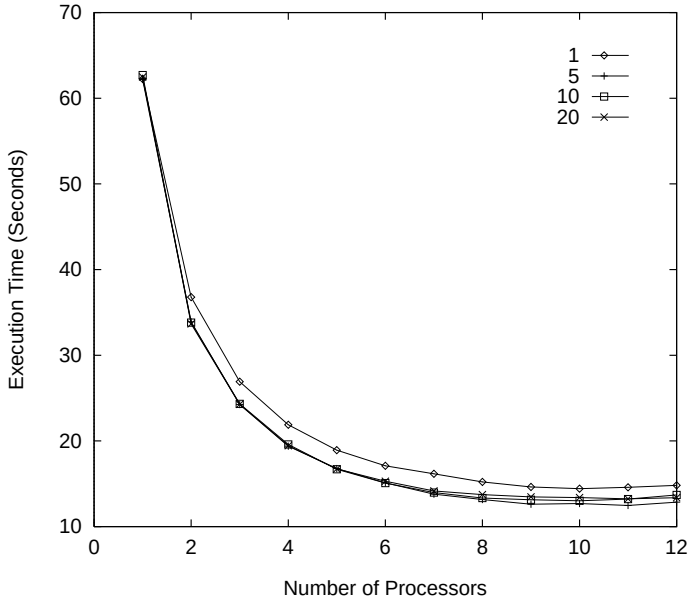

Figure 7.17: LIFE40 Execution Time on Different Grain Sizes (Rule Granularity = 1000).

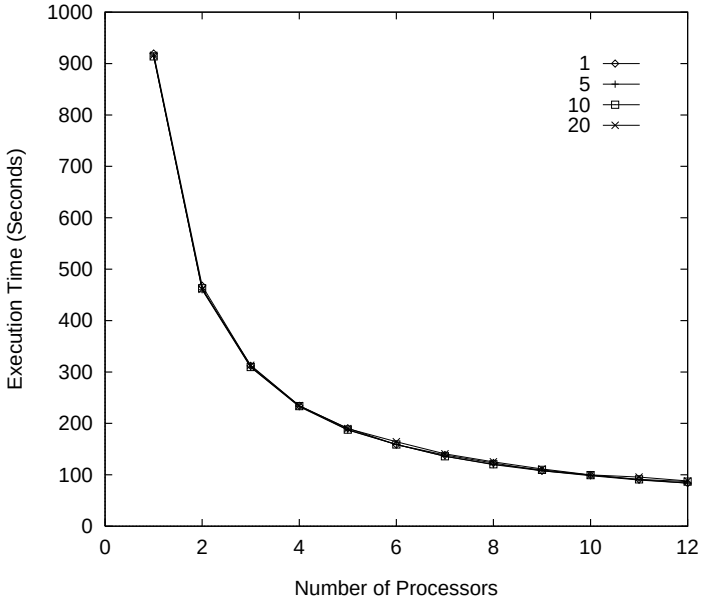

Figure 7.18: LIFE40 Execution Time on Different Grain Sizes (Rule Granularity = 20000).

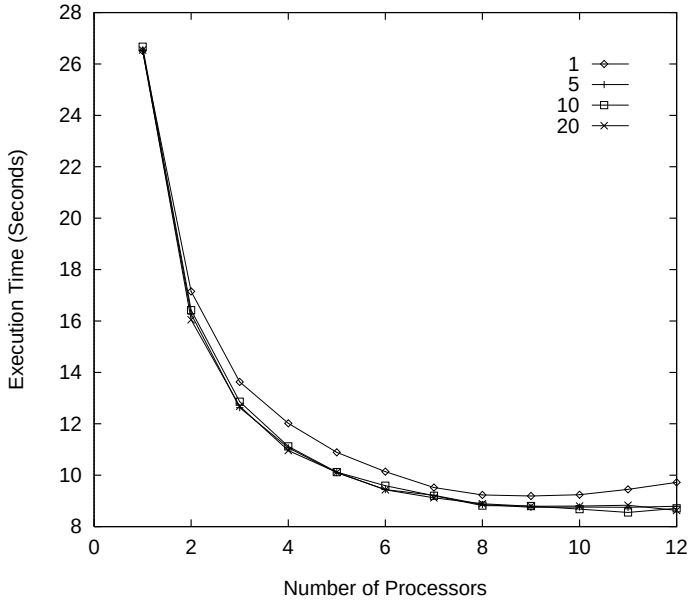

Figure 7.19: WALTZ30 Execution Time on Different Grain Sizes (Rule Granularity = 1000).

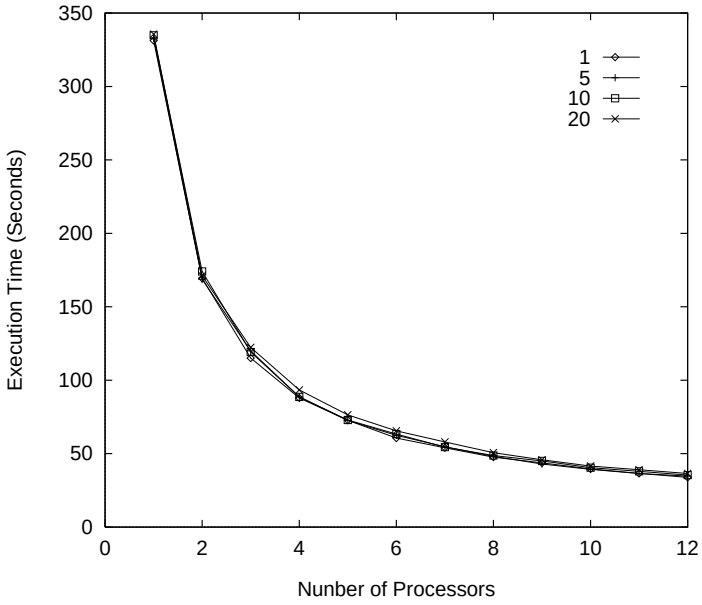

Figure 7.20: WALTZ30 Execution Time on Different Grain Sizes (Rule Granularity = 20000).

Chapter 8

Implementation

No matter how good a simulation is, it is still a simulation. Only a real implementation tells the real story. Based on the experiences gained from the simulation discussed in last chapter, we develop Venus/DA, an object-based parallel rule language with decomposition abstraction mechanisms, on Sequent Symmetry shared memory multiprocessors. In this chapter, we discuss the design and implementation of Venus/DA.

8.1 Form Venus to Venus/DA

To demonstrate that the idea of decomposition abstraction is universally applicable to any sequential rule language, we extend a sequential rule language with DA constructs rather than build a new parallel rule language from scratch. This is also to give an example of how to turn a sequential rule language into parallel rule language encompassing constructs for decomposition abstraction. The sequential rule language we choose is Venus, a C/C++-based modular rule language [18]. Venus is probably the first sequential rule-based programming language to provide both a declarative syntactic and semantic mechanism to support top-down modular design of rule-based programs. The ability to inference upon both primitive and complex C++ objects is particularly attractive to us. In stead of going into details of the Venus language which can be found in the cited paper, we show how straightforward it is to turn Venus into Venus/DA. It only takes minor changes to the syntax while results in substantial enrichment of the semantics. We present the differences between Venus and Venus/DA by listing the grammar rules of Venus/DA that are different from Venus. All other rules are exactly the same.

When specifying the Venus/DA grammar rules, we use the following conventions similar to those used in Stroustrup's C++ book [152]. That is,

- Things in italic are nonterminals.
- Everything else are terminals.

- Alternatives are listed on separate lines.
- Optional items are indicated by the subscript "*opt*".

8.1.1 Functional Dependency Declarations

Global declarations in Venus consist of `#include` *directives*, *constant declarations*, *enumerated type declarations*, and *type declarations*. All of them are retained in Venus/DA. The only new global declaration syntactic construct in Venus/DA is the *functional dependency declarations*. In other words, global declarations in Venus/DA are exactly the same as in Venus with the addition of functional dependency specifications. This implies that, when transforming a Venus program into a Venus/DA program, all data declarations can stay the same. The only thing we need to do is to figure out and specify functional dependencies between different types of working memory objects.

A functional dependency is specified as follows.

$functional\_dependency$:
    funcdep { $identifier\_list$ } → { $identifier\_list$ } ;
$identifier\_list$:
    $identifier$
    $identifier\_list$, $identifier$

The identifiers must be names declared in the type declarations (i.e. class names). The following functional dependency declaration specifies that the class `Department` functionally determines the classes `Office` and `Classroom`.

$funcdep$ – $Department$ " → – $Office$, $Classroom$ " ;

The declaration tells the system that different departments have disjoint set of offices and classrooms.

8.1.2 Rule Definitions

A *rule definition* in Venus consists of a *rule header*, an *alias declaration* section, a *left-hand-side (LHS) expression*, and a sequence of *right-hand-side (RHS) actions*. Continuing with the approach in last section to keep all data declarations of Venus intact, all rule definition constructs are retained with

two additions — an optional *DA expression* in the LHS and a set of *aggregate actions*. We list only those grammar rules that have DA constructs.

The LHS of a Venus/DA rule has the DA expression as an additional construct for specifying decomposition abstraction. The followings are the grammar rules involving the DA expression.

rule_definition :
    rule_header *alias*$_{opt}$ *conditions* *actions* ;
conditions :
    if (*expression* *and_DA_expression*$_{opt}$)
and_DA_expression :
    && *DA_expression*
DA_expression :
    select_op (*select_arg_list* :: *expression*)
select_op :
    SelectALL
    SelectDISJOINT
select_arg_list :
    select_arg
    select_arg_list , *select_arg*
select_arg :
    set_variable
    variable
set_variable :
    variable∗
variable :
    identifier
    eq_pattern_variable
    uq_pattern_variable

The RHS of a Venus rule is a list of *actions* enclosed in braces. All types of actions in Venus are retained in Venus/DA with an additional type of action called *aggregate operation*. This type of operation operates on a set of

objects rather than a single objects. The only aggregate operation supported right now is **Remove** which is to remove all objects in a set of objects selected in the LHS. Other operations are certainly possible. They are left as future extensions.

action :
    ⋮
    aggregate_op_call
aggregate_op_call :
    aggregate_op (*identifier*) ;
aggregate_op :
    Remove

The primitive element of an expression is called a *factor*. Similar to the discussion above, all factors in Venus are retained in Venus/DA with the addition of a new type of factor called *aggregate function call*, which is simply to call an *aggregate function* with arguments properly supplied. An aggregate function computes a value based on a set of values passed as arguments. The aggregate functions are **Count**, **Avg**, and **Sum** with usual meanings.

factor :
    ⋮
    aggregate_function_call
aggregate_function_call :
    aggregate_function (*arg_list*)
aggregate_function :
    Count
    Avg
    Sum

As an example to the DA constructs above, the following rule is a slightly modified, Venus/DA version of the rule `DA_Count_Students` presented in Chapter 6.

```
rule  DA_Count_Students;
from  CurrentTask[?]  t;
      Department[?]   d;
      Student[?]      s;
if ( t.task == COUNTING &&
     SelectALL ( d, s* ::
         d.students_counted == NO &&
         s.dept == d.name ) )
{
    d.count = Count(s*),
    d.students_counted == YES,
}
```

8.1.3 Remark

The key point we want to emphasize is that it is relatively straightforward to transform a sequential rule language into a DA language. It only takes a minimal syntactic changes. We actually have a syntax for adding DA constructs to OPS5, which can be called OPS5/DA. One can easily have CLIPS/DA or AnyLang/DA where AnyLang is any sequential language. Because of this consistency with sequential rule languages, it is possible to have a single run-time system for all the extensions. We will discuss more about this in later sections.

8.2 A Thread-Based Execution Environment

We select the Sequent Symmetry multiprocessor as our target machine because the shared-memory model of parallel computation is closer to the production system model than distributed memory machines and Symmetry was the only shared-memory machine we had at the time we started programming. The implementation is based upon an object-based thread package called PRESTO [13, 41]. This C++-based package provides C++ objects to server as our underlying object system. It also includes a variety of thread management primitives and synchronization objects such as condition variables, monitors [65], and locks. Because of the object-oriented nature of the package, it is easy to build more complex synchronization objects for various purposes such as barrier synchronization. Even though it is not as effective as another very efficient C-based thread package called FastThreads [3, 4] developed by the same research group, PRESTO is still nearly two orders of magnitude faster than the

process-based parallelism provided by Sequent's Dynix operating systems [41]. Its flexibility and extensibility also make it a better choice than FastThreads.

Another system issue that has significantly impact on our implementation is that Venus/DA is based upon a GNU G++ implementation of the Venus language. The run-time system is therefore written partially in Sequent C++ and partially in GNU G++. This took us tremendous amount of efforts in making things work together, especially because Sequent C++ does not support templates which are used heavily in Venus implementation. Details of the software system related issues such as this one, however, will not be discussed any further.

8.3 Implementation Framework

The implementation techniques developed from OPS5c [103, 104] and Venus [18] provide a general and effective framework for the implementation of rule-based languages. As depicted in Figure 8.1, the framework suggests a *common intermediate form* as the interface between front end parser and back end code generator. Using this approach, the same code generator and run-time libraries (RTL) can be used to implement different rule languages. All we need to write for a new language is a new parser for the specific language syntax. This claim is supported by the existing parsers for OPS5, CLIPS, and Venus as presented in the figure. The intermediate form is designed in such a way that it is easily extensible to incorporate new features of new languages. The code generator and RTL can also be extended to handle these features. This is the original approach we intended to take which is delineated in Figure 8.2.

When working on the real implementation, we actually started from the RTL since all parallel execution related codes are in the RTL. For testing the RTL implementation and to gather performance results, we developed an alternative approach toward the Venus/DA implementation. Since all of the test programs are directly translated from Venus programs for performance comparison, we developed a code translator to translate the C++ code generated by the Venus compiler into corresponding code that are supposed to be generated by the Venus/DA compiler. This is done with the help of a simple configuration file that can be either hand edited or directly generated from the Venus/DA source code. The translator is implemented by a set of Perl scripts [157]. Figure 8.3 presents this code translation approach. It has the advantage of fast implementation and evaluation. Performance results can be gathered

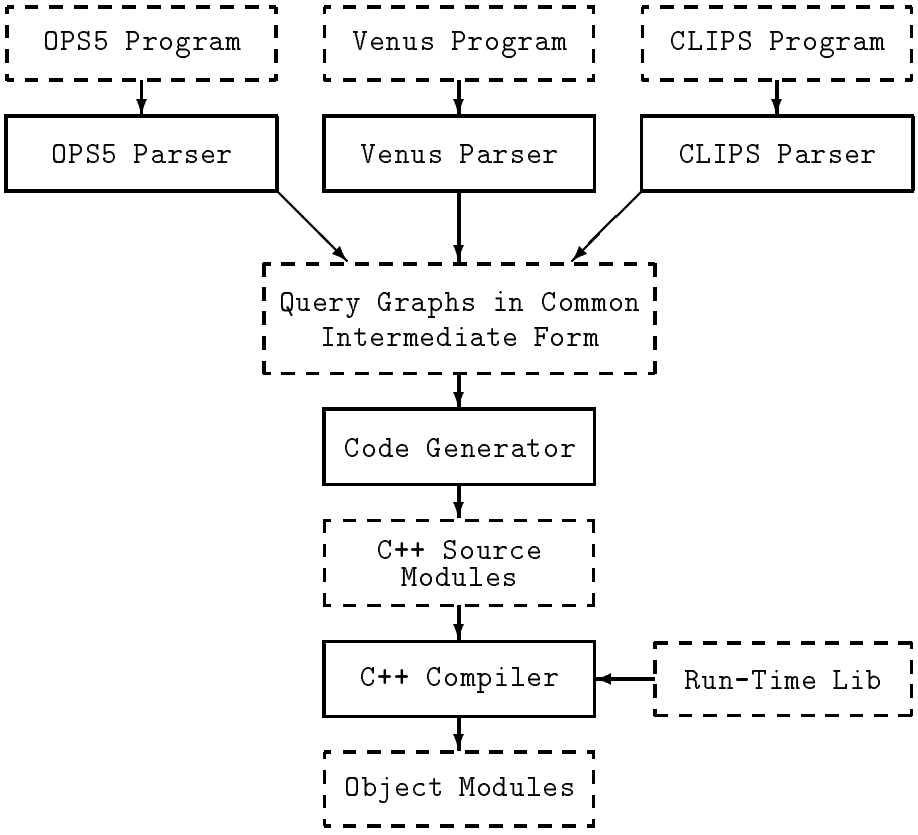

Figure 8.1: A General Framework for the Implementation of Rule-Based Languages.

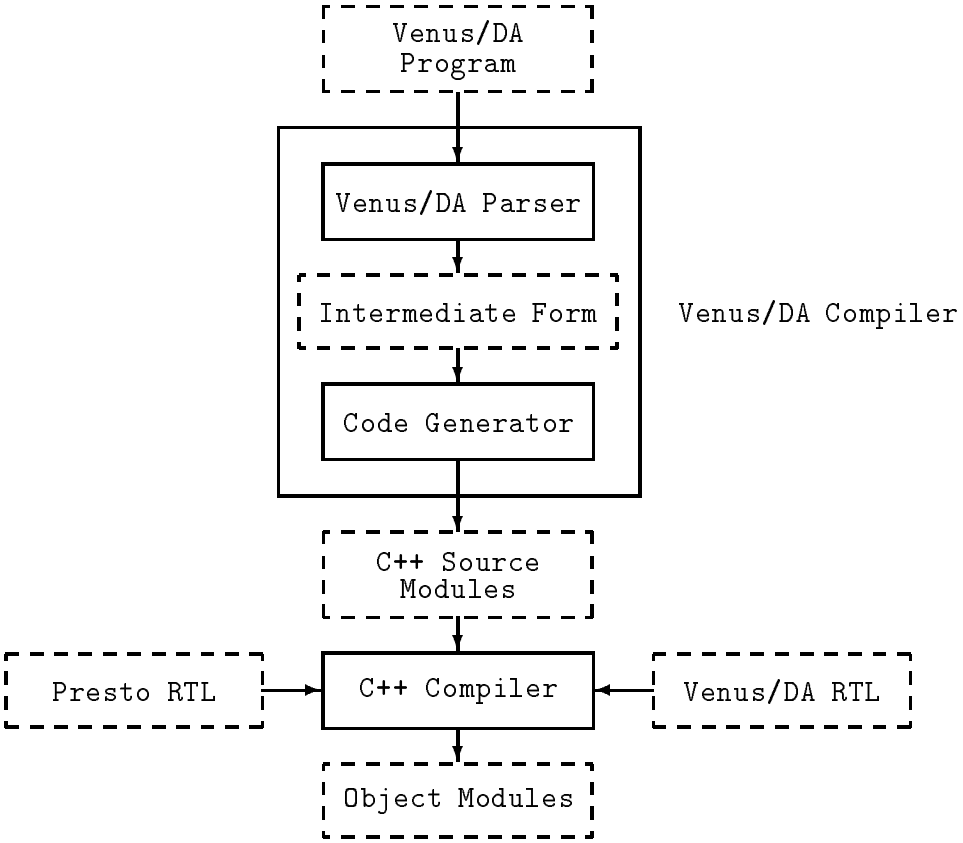

Figure 8.2: Venus/DA Implementation.

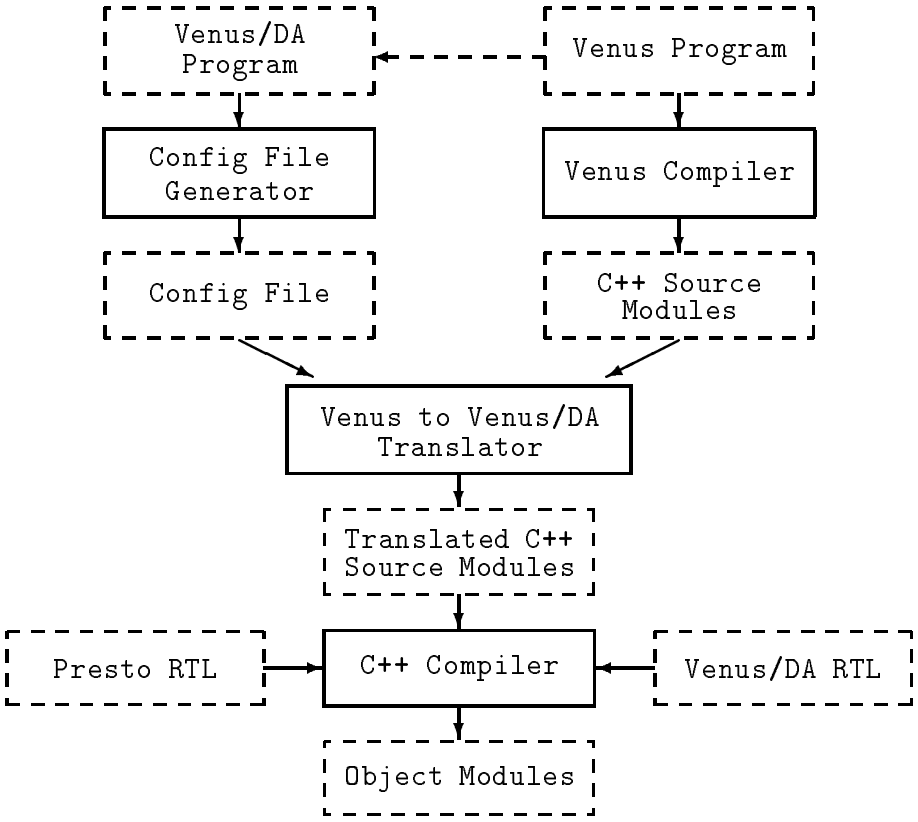

Figure 8.3: Venus/DA Implementation: An Alternative Approach.

while other parts (such as parser and code generator) are still being implemented. This is also facilitated by our DA approach being a natural extension to sequential rule languages.

8.4 Run-Time System

It is fair to say that the run-time system (RTS) is the most important part of our Venus/DA implementation. The effectiveness of the DA approach can only be materialized with an efficient RTS implementation. In particular, the algorithms used in the parallel match and parallel execution have decisive impacts on whether the parallelism expressed by the programmers can be effectively exploited. In this section, we discuss the design and implementation of the Venus/DA run-time system.

8.4.1 Implementation Strategies

We highlight the main strategies used for the Venus/DA implementation. The general goals are to simplify the implementation and to reduce the synchronization cost as much as possible. The details and rationales for adopting these strategies are described in later sections.

- Shared-memory model of computation. All working memory objects reside in shared memory. Communications and synchronizations are done through shared variables.[1]

- SPMD style execution [27, 144]. A number of inference engines capable of matching and rule execution are working asynchronously under the shared working memory. Inference engines are synchronized only at the barriers or when accessing shared objects.

- Parallel match and asynchronous execution within parallel cycles. Barrier synchronizations are enforced only between parallel cycles.

- Eliminate the entire phase of conflict-resolution and run-time interference detection.

- Based on working technologies and existing sequential implementation. This includes the use of Lazy Match algorithm [99, 100] and the Venus implementation.

- Static work load distribution and prescheduling by *copy-and-constrain* (C&C) [118, 119, 141].

- The *nondeterministic safe* assumption, i.e. firing rules nondeterministically should not affect the correctness of the program execution. In other words, the input source program should not depend on rule priority or any implicit conflict resolution strategy for its correctness.

[1] By using Sequent Symmetry and shared memory, we avoid the problems such as working memory partitioning and localities. While these issues are as important as other issues we studied, they are not within the scope of this thesis. They are, however, certainly in our plan for future works which is discussed in Chapter 10.

8.4.2 System Architecture

Figure 8.4 is the system architecture of the Venus/DA RTS which is essentially an integration of the Venus/DA run-time structure with the PRESTO system components [12]. A number of identical *LEAPS engines* (LE's) are allocated during the initialization phase. Each LE is run on a separate PRESTO thread which can be in either *ready*, *run*, or *wait* state. All LE's are initially in ready state. When there are idle processors available, the PRESTO scheduler assigns a processor to run on a ready thread. A thread is in run state when running on a processor. Each LE is a full fledged inference engine that can match, execute rule instantiations and synchronize with other LE's. LE's run asynchronously except when waiting for a lock, a condition variable (normally associated with a monitor), or in a barrier. A waiting thread becomes ready when the lock is successfully acquired, or the condition variable is signaled, or the barrier is complete (i.e. all participating threads arrive at the barrier). Rules are disjoint partitioned and assigned to the LE's. Working memory objects are kept in shared memory. Synchronization objects also reside in shared memory. They are special objects such as locks, condition variables, monitors, and barriers, which are used exclusively for synchronization.

There are several advantages for this architecture:

- First of all, it simplifies implementation considerably. With the implementation of a single LE, we automatically have parallel match and parallel execution. The parallel implementation can base on the sequential implementation. In other words, the parallel implementation needs mostly to handle the synchronization issues rather than the match and execution which are already available from the sequential implementation.

- This architecture can easily support various rule partitioning strategies. In particular, it support CREL-style clustering [101]. This is because each LE is a full-fledged inference engine. It can be assigned any number of rules for match and execution.

- Potential contention on the stack for a LEAPS-based parallel implementation is greatly reduced because each LE has its own stack. Even though each LE may need to process some extra stack entries for the sake of parallelism, this architecture effectively partitions the sequential LEAPS stack into a number of smaller stacks equal to the number of LE's. We will discuss the stack issues in more detail when describing our parallel LEAPS-based inference system in next section.

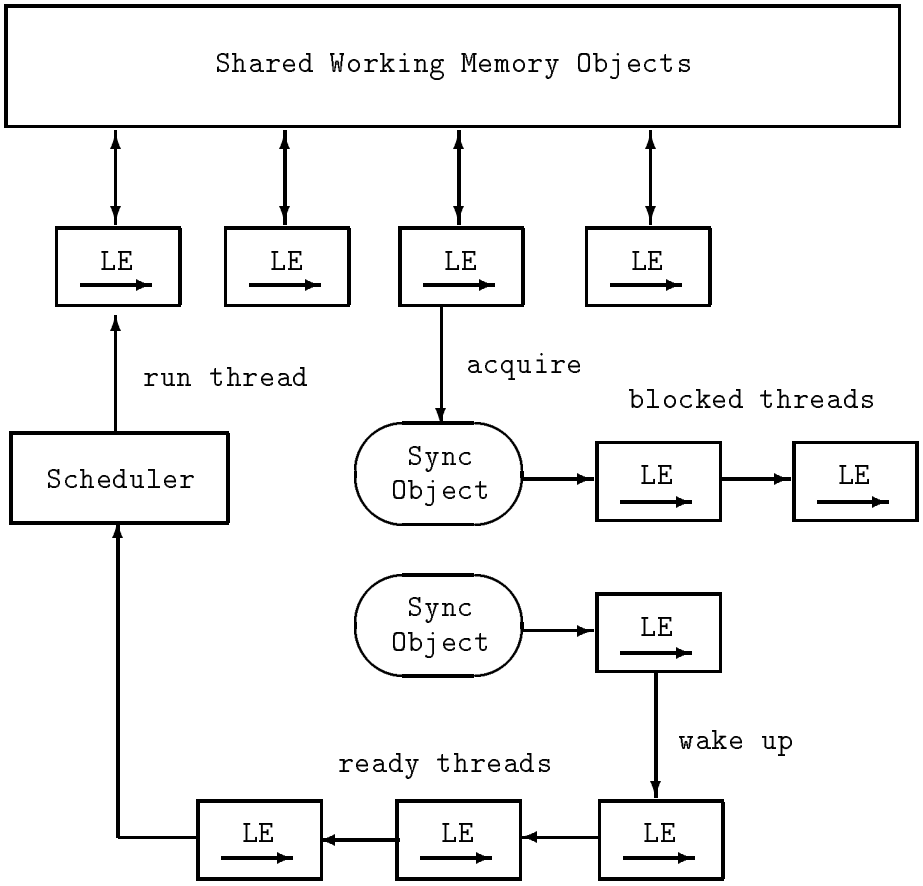

Figure 8.4: Venus/DA Run-Time System.

- This approach also facilitates scheduling and load balancing as well. Static load balancing can be done by partitioning of rules into different LE's. Run-time scheduling is straightforward since all LE's are identical.

- Asynchronous execution follows automatically with this approach since when each LE is running on a separate thread, all LE's are running asynchronously.

- Synchronization is also simplified since all LE's are the same and therefore follow the same synchronization patterns.

- With each LE capable of both match and execution, each can fire the instantiations found by itself, and therefore reduce the potential overhead of having separate matchers and executors such as the architecture adopted by Neiman [111].

In the following sections, we discuss our LEAPS-based parallel inference system and other components of the Venus/DA RTS.

8.4.3 A LEAPS-Based Parallel Inference System

The Venus/DA RTS consists of a number of identical LE's that are capable of match, firing, and synchronization. The kernel of the RTS is a LEAPS-based parallel inference system that coordinate these LE's to perform the parallel match and multiple rule firings. Each LE runs a modified version of the sequential LEAPS algorithm, which we call LEAPS/DA. By static partitioning, each LE is assigned a number of rules for processing. Each LE is responsible for keeping all stack entries generated for the rules assigned to it.

The execution proceeds in *parallel cycles*. During a cycle, LE's match and fire instantiations asynchronously until a point where barrier synchronization is needed to maintain correctness. When all LE's arrive at the barrier, a new parallel cycle begins. The execution continues until:

- an explicit **halt** statement is encountered, or

- when none of the LE's can find any instantiations to fire, or

- when the stacks of all LE's are empty.

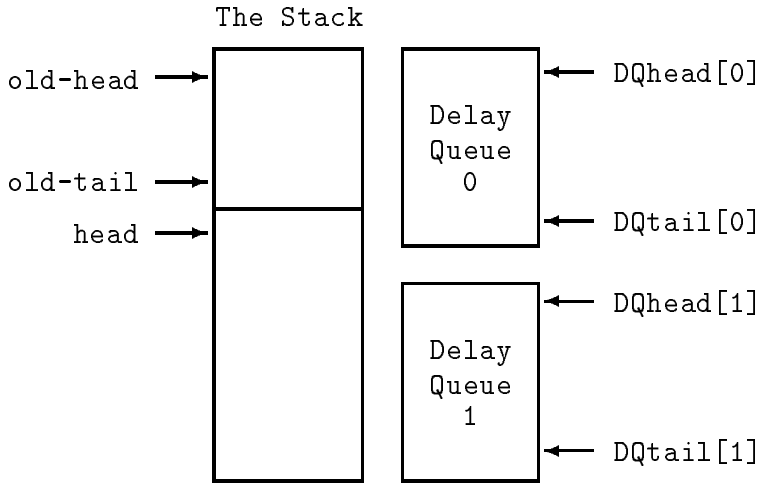

Figure 8.5: The LEAPS/DA Stack Organization.

8.4.3.1 The LEAPS/DA Stack Organization The central data structure of the LEAPS algorithm is a stack called *LEAPS stack* for maintaining the search states. The LEAPS stack in Venus is actually a priority queue since entries are sorted in predefined order. The stack organization of LEAPS/DA is essentially the same as LEAPS stack except that, for the purpose of parallel execution, a pair of *alternating delay queues* is associated with each LEAPS/DA stack. Figure 8.5 depicts the organization of a LEAPS/DA stack. The delay queues are for keeping stack entries that should not be processed during the current parallel cycle. In other words, all new stack entries pushed during the current parallel cycle are placed in the delay queues rather than pushed into the stack. This is to prevent new entries from interfering with the current search. Entries in the delay queues are actually pushed into the stack after the barrier synchronization is completed. All LE's resume by doing a stack adjustment before the search for new instantiations. Since stack adjustment of different LE's are performed asynchronously, some faster LE's may interfere with the stack adjustment of slower LE's. This is where the alternating design comes into play. The design ensures that, for each LE, all new entries are placed in a different delay queue than new entries of the last cycle. Since stack adjustment is performed only once for each cycle, a pair of alternating delay queues suffices.

There are several advantages to this design:

- First and foremost, it helps maintain the correct execution of the parallel rule programs.

- The alternating design also reduces contention considerably since the stack adjustment can be performed completely locally within each LE with no worry of any possible interference.

- It simplifies the inference algorithm (to be discussed in the next section) since it rules out a significant portion of the potential interference between rule firing and instantiation search within the same parallel cycle.

The design merely results in minor overhead since for each entry to be pushed into the stack, only an additional append operation is needed. The cost is comparatively much smaller than the cost of stack push which requires a search down the stack to find the proper place to insert the new entry.

8.4.3.2 The LEAPS/DA Inference Algorithm One thing that can be easily overlooked when designing parallel algorithms or writing parallel programs is that doing less work, and therefore less time, to achieve the same results is as efficient, and sometimes more efficient, than trying to find the best way to do things in parallel. In the design of the LEAPS/DA inference algorithm, one of the most important criteria is to eliminate as much as possible any unnecessary or redundant work. As an example, in the design of match algorithm, our primary focus is not to do the match work faster but rather to do much less match and achieve the same results, i.e. finding the same set of instantiations. By doing less work in parallel, even a simple parallelization strategy may achieve the same or superior performance than complex parallel match algorithms. In this section, we detail the LEAPS/DA inference algorithm.

Figure 8.6 presents the LEAPS/DA inference algorithm. The same algorithm is run by each LE in the RTS. After initialization, each LE that passes the context test (we will discuss more about the context mechanism later) starts by popping a stack entry from its own stack and applies the LEAPS best-first search to find an instantiation. A found instantiation is fired immediately if:

- no instantiation has ever been fired during the current cycle, or

- the instantiation is compatible with all other instantiations that are fired in the current cycle.

To test the conditions above, especially the second condition, it is not necessary to resort to expensive run-time interference detection as done by

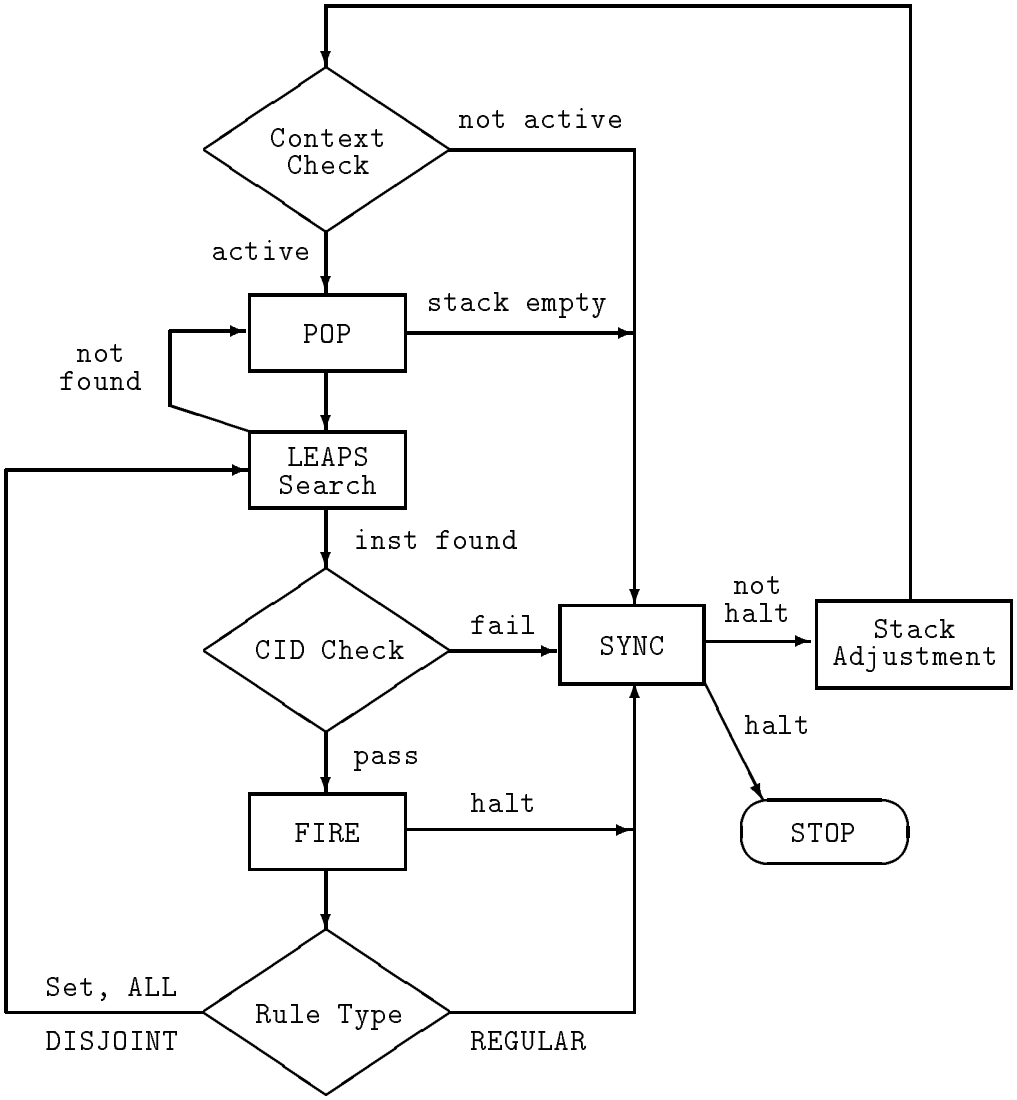

Figure 8.6: The LEAPS/DA Inference Algorithm.

most previous research. We can use a *rule compatibility matrix* which is part of the results of the compile-time semantic and syntactic interference analysis. To further reduce the run-time overhead, the Venus/DA implementation adopts an even simpler approach. Instead of the rule compatibility matrix, we introduce the notion of *compatibility graph* and *compatibility set*.

The *compatibility graph* of a rule program P is an undirected graph $G = (V, E)$ such that each vertex i in V corresponds to a rule r_i in P and there is an edge between vertex i and j if and only if r_i and r_j are compatible. A *compatibility set* of G is a complete subgraph of G. In other words, a compatibility set is a set of rules that are pair-wise compatible. The compatibility graph is just another representation of the rule compatibility matrix and therefore can be obtained from compile-time interference analysis.

For testing the conditions above, we identify the disjoint partitioning of G into a set of maximally complete subgraphs. Each subgraph is a compatibility set and is assigned an unique ID number. Each rule is assigned a *compatibility ID* (CID) which is the ID number of the compatibility set the rule resides. The RTS maintains a shared variable called *current compatibility ID* which is reset to null at the start of each parallel cycle. It is atomically set to the CID of the first instantiation fired and remains the same throughout the current cycle. Subsequent instantiations must have the same CID to be eligible for parallel execution. With this approach, the test of the second condition is simplified into just a simple comparison between the CID's.

When a compatible instantiations is found, it is fired immediately. On the other hand, if an instantiation is conflict with the current compatibility set, the LE that generates the instantiation stops immediately and waits at the barrier. After firing an instantiation, an LE keeps on searching for more instantiations if the rule just fired is a parallel rule. That is, if

- compile-time interference analysis shows that all instantiations of the rule can be fired in parallel,
- the rule's antecedent includes one or more set selection conditions, or
- the rule's antecedent includes an ALL or DISJOINT combinator.

Such LE's keep searching and firing until no more instantiations can be found. When there is nothing more to do, an LE stops and waits at the barrier. When all LE's arrive at the barrier, a new parallel cycle begins.

Figure 8.7 presents the LEAPS/DA inference algorithm in pseudo code. The subroutines used in the algorithm are explained in Figure 8.8. For readability, we use identation to denote block structures and only use braces for long while loop. In the algorithm, `halt` is a shared variable which is set to true when an explicit **halt** statement is encountered or when no instantiation is found during the entire cycle. The details of the implementation of DA constructs are discussed in the forthcoming sections.

8.4.4 Implementing Set Selection Conditions

We discuss the implementation of set selection conditions that are used purely for data parallel computation (i.e. not used with aggregate operations), Aggregate operations are discussed in later section.

With the LEAPS-based implementation, set selection conditions can be accomplished through cursor management. For each rule with set selection conditions, we record the indices of those condition elements. The search for instantiations proceeds as if all set selection conditions were regular condition elements. When the first instantiation is found and fired, it is used as a seed to find other instantiations. This is done by advancing only the cursors correspond to the set selection conditions. All instantiations thus found can be fired immediately without checking the CID's. When no more instantiation can be found from the seed, the LE stops and wait at the barrier.

This approach employs no additional data structure and incurs little overhead since it tries to find all executable instantiations directly by advancing the right set of cursors. The efficiency is gained not only from parallel execution, but also from the saving of not doing useless work.

8.4.5 Implementing ALL Combinator

The implementation of the ALL combinator is similar to that of the set selection conditions. For each rule with an ALL condition, the index of the first condition element in the ALL combinator is recorded as the *ALL combinator index* or *ACindex*. The search for instantiations proceeds in a way similar to the implementation of set selection conditions, i.e. by treating all condition elements as regular condition elements. When the first instantiation is found, it is fired and used as a seed to find other instantiations. This is done by advancing only the cursors of the condition elements with indices larger than or equal to the ACindex. In other words, advance only those cursors correspond to the condition elements enclosed in the ALL combinator.

```
algorithm leaps_da_inference;
while !halt {
    while context_active() and stack_not_empty() {
        entry = pop();
        instantiation = leaps_bfs(entry);
        while instantiation != null {
            if cid_check(entry.cid) then
                fire(instantiation);
                if halt then
                    terminate();
                else if entry.rule_type == REGULAR
                    sync_barrier_check_in();
                    break;
                else
                    instantiation = leaps_bfs(entry);
                    continue;
            else
                sync_barrier_check_in();
                break;
        }
        if parallel_instantiations_fired then
            continue;
        else
            stack_adjustment();
    }
    sync_barrier_check_in();
    stack_adjustment();
}
```

Figure 8.7: The LEAPS/DA Inference Algorithm in Pseudo Code.

```
function context_active();
// Probe the top of the stack and check if the context of
// the top entry is active.

function stack_not_empty();
// Check if the local stack is empty.

function pop();
// Pop an entry out of the local stack.

function leaps_bfs(stack_entry);
// Apply LEAPS best-first-search on the given stack entry.
// Return an instantiation if found, null otherwise.

function cid_check(cid);
// Compare the given cid with current_CID for equality.
// If current_CID has not been set yet, set its value to
// cid and return true.

procedure fire(instantiation);
// Fire the given instantiation.

procedure terminate();
// Terminate the inference algorithm.

procedure sync_barrier_check_in();
// Report arrival at the synchronization barrier.

procedure stack_adjustment();
// Adjust the local stack for the next cycle.
```

Figure 8.8: Subroutines Used in the LEAPS/DA Inference Algorithm.

We note that before firing the seed instantiation, a check of the CID must be made. It can only be fired when it has the same CID. Subsequent instantiations found can be fired directly without checking the CID's.

8.4.6 Implementing DISJOINT Combinator

Because of the disjointness requirement, the implementation of DISJOINT combinators is not as straight forward as the set selection conditions and ALL combinators. However, the work is still accomplished through cursor management.

For each rule with a DISJOINT condition, the index of the first condition element in the DISJOINT combinator is recorded as the *DISJOINT combinator index* or *DCindex*. A set called *disjoint timestamps set* is maintained to keep track of the time stamps of working memory objects in the disjoint condition of any instantiation of the rule that is actually fired.

The first instantiation (i.e. the seed) is searched in the same way as the ALL combinator. If the CID is checked and the seed instantiation fired, the other instantiations are found by an intelligent backtracking technique:

- The cursor that corresponds to a condition element in the DISJOINT combinator and has the highest position in the LEAPS search tree is located and advanced. The node in the tree is called the *disjoint root*, i.e. the root of the *disjoint subtree*.
- All other cursors correspond to the condition elements in the DISJOINT combinator are reset. The search for instantiations proceeds by advancing only these cursors.
- When doing the search, all working memory objects with time stamps appear in the disjoint timestamps set are simply skipped.
- Once an instantiation is found, it is fired immediately without the need to check CID.
- When all the cursors have been exhausted, the cursor pointing to the disjoint root is advanced again and the rest of the cursors in the DISJOINT combinator reset as above.
- The whole process repeats until no more instantiation can be found, i.e. until all cursors in the DISJOINT combinator (including the disjoint root) have been exhausted.

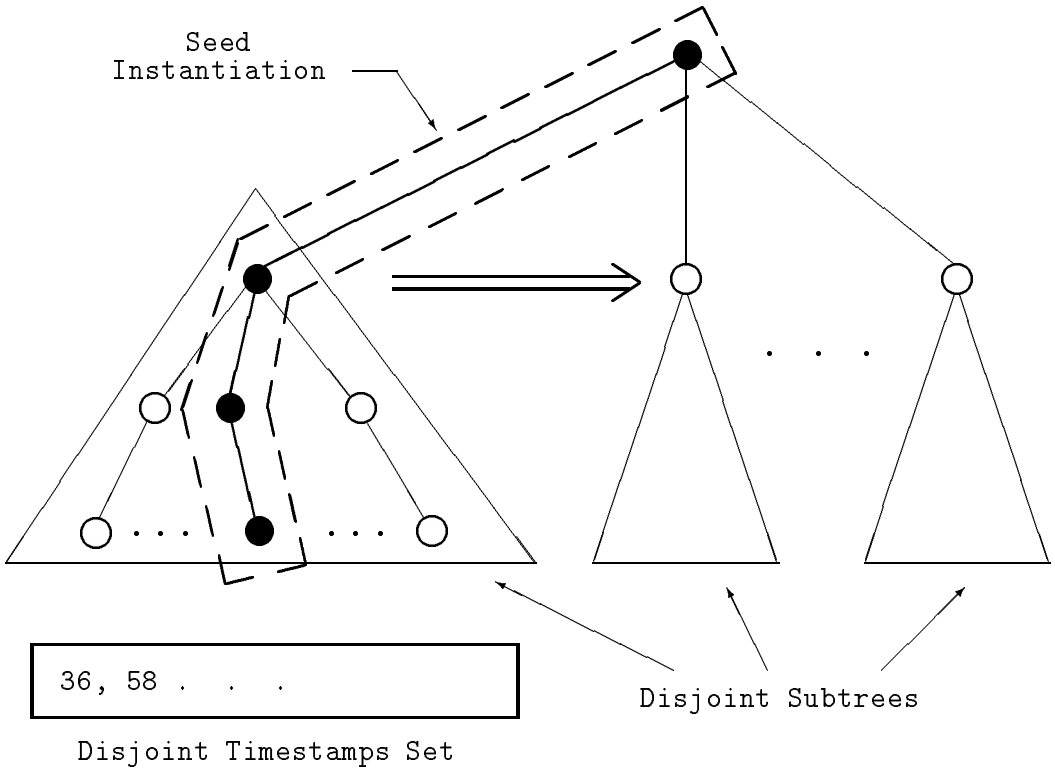

Figure 8.9: The Implementation of the DISJOINT Combinator.

The technique above is not as complex as it appears. We essentially explore the disjoint property by skipping all unnecessary joins. The advancing of the disjoint root cursor effectively skips the whole disjoint subtree under that root and starts with a new disjoint subtree. The test against the disjoint timestamps set reduces further the join work that must be performed otherwise. Figure 8.9 should help illustrate the technique and the significant saving we obtained from reducing the join work.

8.4.7 Implementing Aggregate Operations

Since aggregate operations are always used in conjunction with set selection conditions, the implementation of the two are closely related. Same as the implementation of set selection conditions, we record the indices of all set selection conditions. The seed instantiation is again found in the same way. An important difference, however, is that for a rule employing aggregate operations in its consequence, the seed is not fired when the CID check is passed. In stead,

we maintain a set of executable instantiations and an *accumulation variable* for each aggregate operation in the consequence. For example, if the operation is **Count**, we keep a counter to count the number of working memory objects satisfying the set selection condition and are part of an executable instantiation. The saved instantiations are fired when the cursors advancement leads to no more instantiations under the seed. When firing these instantiations, all aggregate operations are replaced with the values of the proper accumulation variables.

The aggregate operations have not been actually implemented yet. We have decided not to implement them in the current environment. The primary reason is that the platform and softwares we used to build the Venus/DA system is out of date. It serves our purpose well enough to build a prototype that demonstrate the effectiveness of the decomposition abstraction approach. Our plan is to rewrite the whole thing on advanced platforms (including distributed memory machines) with modern software environments.

8.4.8 Implementing Contexts

We have implemented a less general version of the proposed context mechanism for the purpose of evaluation. More specifically, we assume the existence of a designated starting context which is to read in all the data. The relationships between contexts are represented by an $n \times n$ matrix where n is the total number of contexts in the program. For each context, the matrix records the next set of contexts that should be activated when a context is finished. The termination of a context is determined by an explicit `End_of_Context` call, a `Switch_Context` call, or when no rule is eligible for firing in the current cycle. A call to `End_of_Context` simply indicates the end of the current context. The `Switch_Context` call specifies a new context to activate which also terminates the context that makes the call. The RTS maintains a set of *active contexts*. During each cycle, a search for instantiation(s) is only initiated for a rule whose context is active.

This simple technique results in significant reduction in the excess work done by the RTS due to parallelism. Without the context mechanism, the search of instantiations for those rules whose contexts are not active may waste a significant portion of the computation resource. For Venus and Venus/DA, however, the ideal mechanism for similar purpose should probably exploit the modularity of the language. This is one of our future direction for this research.

8.5 The Venus/DA Translator and Compiler

As discussed earlier, the "right" way to implement the Venus/DA language is the traditional parser and code generator approach. This will constitute the Venus/DA compiler presented in Figure 8.2. We select the translation approach depicted in Figure 8.3 since our primary goal is to understand the effectiveness of the decomposition abstraction approach; besides, the platform and software environment we used are out of date. In the future rewrite of the entire system, we will certainly take the parser and code generator approach. In this section, we describe the Venus to Venus/DA translator and discuss how to implement the actual compiler.

The translator is a set of Perl scripts that handles any Venus program consisting of a single main module. In other words, the modular feature of the Venus language has not been incorporated into the DA system yet. The interplay between parallelism and modularity, the relationship between context mechanism and modularity are good topics for future research. We, however, concentrate on our implementation goal which is to demonstrate and evaluate the effectiveness of the DA approach.

The translator takes input from the C++ source code generated by the Venus compiler and a *configuration file*. The configuration file is a simple table consists of entries of rule information, one entry per rule. Each entry is composed of five fields:

- rule name,
- context id,
- rule type (REGULAR, ALL, or DISJOINT),
- the rule's CID, and
- the ACindex or DCindex (for REGULAR rule, this field is not used)

As discussed earlier, all information above can be obtained at compile-time, through parsing, code generation, as well as interference analysis. The configuration file is actually an extension to the Venus rule configuration process. The process records important feature of rules on run-time data structures. We add information related to parallel execution to the run-time data structures so that the Venus/DA RTS can use them.

The main part of the translator is to partition the rules among LE's such that the join code and the execution code of the same rule are always handled by the same LE. Other parts of the translator are primarily to augment the code for parallel execution. Most of the parallel execution related code are separated from the C++ source code such that the same parallel code can be used on different rule programs. Rest of the translator simply copies the input source code line by line into output file, possibly with some substitutions applied. The translator also breaks large input source into separate compilation units for parallel compilation on the Sequent.

For static load balancing with C&C, selected rules from the Venus source code are copied and constrained by hand. The resulting Venus source is then compiled by the Venus compiler to generate the C++ source code for the input of the translator. It is certainly possible to build a subsystem to do the job. We decided not to do so since this is not our primary concern. Similar issues have been studies elsewhere [35, 36].

8.6 Chapter Summary

This chapter presents in details how we implement the Venus/DA run-time system and the constructs for decomposition abstraction. We demonstrate how to convert a sequential rule language (Venus in our case) into a parallel rule language with DA constructs (Venus/DA). The core of the implementation technologies is the LEAPS/DA inference algorithm that employs multiple LEAPS engines for SPMD style parallel matching and rule execution. We must emphasize that the techniques developed here can be easily adopted to implement other DA language systems. By developing the system on Sequent Symmetry, however, we avoid an important issue that is almost certain to have significant impact on other systems, namely locality issue. On distributed memory machines, for example, we not only need to partition the rules but also the working memory objects to different processing nodes. The communication cost would certainly be much higher. These issues are among our primary directions for future research.

Chapter 9

Experimentation and Performance Results

For performance evaluation, we have conducted a variety of experiment on the three bench mark programs. The experimentation plan is designed with the following objectives:

- To evaluate the effectiveness of the DA mechanisms.
- To discover the strength and weakness of the DA mechanisms.
- To understand the behavior of the Venus/DA RTS. In particular, we want to test how well the system scale with respect to computation resources and problem size.
- To find out the reasons for the success or failure of the DA approach.

In this chapter, we document our experimentation on the Venus/DA RTS and analyze the performance results. We note that, since the purpose of this implementation is to build a prototype with the aim of future rewrite on modern platform and software environment, we emphasize on the variety of the experiments rather than the number of test programs we run.

9.1 The Benchmark Programs

We use the same MANNERS and WALTZ programs as we did in Chapter 7. The LIFE program is dropped because of the decision of not to implement aggregate operations in the current prototype. It is replaced by a much larger program called ARP which is a route planning program employing the A* algorithm. We list the programs in Table 9.1 for reference.

9.1.1 From Venus to Venus/DA

The following steps are taken to prepare a Venus program for our experimentation:

Program	No. Rules	Description
MANNERS	8	A combinatorial search program for seat assignment.
WALTZ	33	A constraint satisfaction program using Waltz's algorithm for scene labeling [158].
ARP	111	A route planning program using A* algorithm.

Table 9.1: Benchmark programs used in the experiments.

1. **Nondeterministic safe transformation.**

   Transform the Venus program into a nondeterministic safe program. This is usually done by strengthening rule antecedents such that each rule matches exactly the state it is designed to fire. Sometimes it is necessary to replace a less specific rule with more specific rules, or add some new rules. The transformed program must run correctly without relying on any explicit or implicit conflict resolution strategies such as priority, recency, specificity, or rule order.

2. **Copy-and-constrain parallel rules if necessary.**

   All parallel rules (i.e. rules containing set selection conditions, ALL combinator, or DISJOINT combinator) are candidates for C&C.

3. **Replace stage changing rules with the DA context mechanism whenever possible.**

   In most cases, this should reduce the number of rules in the program.

4. **Compilation using the Venus Compiler.**

   The nondeterministic safe and C&C version of the Venus program is compiled using the Venus compiler. This can be done on any platform where the Venus compiler is available.

5. **Construct the configuration file.**

   As discussed in last chapter (Section 8.5), this is currently done by hand.

6. **Venus to Venus/DA Translation.**

   Translate the C++ code generated by the Venus compiler into Venus/DA C++ code. Simply run the translator.

7. **Compile the Venus/DA C++ code.**

   This should be done with C++ compiler on the target machine.

Surprisingly enough, the most time consuming step is the first step that transform the original Venus code into a nondeterministic safe program. The reason is that all three programs (actually most sequential rule programs) rely on explicit or implicit conflict resolution strategies for their correct execution. Some rules can be retained without change while some other rules may require the recognition of the exact state for those rules to fire. We also need to prevent some rules from firing earlier than then they should. Note however that these problems occur only because the original programs were written to take advantage of the built-in conflict resolution strategies of Venus. If we were to start from scratch with the Venus/DA language, things should be much better.

We summarize the changes made to the test programs with emphasis placed on how various sources of parallelism in the programs are expressed using DA mechanisms.

MANNERS

Compare to the other two programs, MANNERS is a relatively easy one. It consists of 8 rules and only one of them, namely the `make_path` rule, is a parallel rule. Since all instantiations of the rule can be fired in parallel, we simply transform it into an ALL rule.

WALTZ

One type of rules that are most difficult to do the nondeterministic safe transformation are the rules that are designed to fire only when no other rules can fire in the same context. In many cases, these rules stay satisfiable throughout the execution of the context but not fired because of some implicit conflict resolution strategies until the end of the context. Most of the context changing rules are this type of rules. It is particularly difficult to characterize exactly the states that these rules should fire such that they don't fire otherwise. In many cases, we not only need to transform the rules in questions, we may also need to change all other rules in the same context as well. A practical (but not necessarily elegant) technique is to add boolean attributes to data objects that are processed in the context such that they are checked when processed. The context changing rules can be transformed by adding the test that all such

attributes have been successfully checked (i.e. all processing has been done successfully).

We highlight the transformation done on the WALTZ program:

- The `reverse_edge` rule (or all its copies) is transformed into an ALL rule since the rule is intended to duplicate all line segments.
- All junction making rules are transformed into DISJOINT rules since each rule matches disjoint set of edges to form a junction. These rules are also pair-wise compatible because of the functional dependency between the junctions and the edges.
- The `match_edge` rule (or all its copies) is transformed into an ALL rule since all pairs of opposite edges (such that one is labeled and the other is not) must be matched.
- All labeling rules are transformed into DISJOINT rules since each rule match a junction and the set of edges associated with it for labeling. For the same reason as the junction making rules, these rules are pair-wise compatible.
- The rule that label the remaining unlabeled edges as boundaries is a perfect ALL rule. However, in the original Waltz program, the rule labels an edge and also print its label. The output will be unreadable if multiple instantiations of this rule are fired in parallel. We therefore split the labeling and printing into two rules. The rule that labels the remaining edges is transformed into an ALL rule.
- All other rules remain the same (i.e. stay as regular rules).

ARP

The central part of the ARP program is a route planner that employs the A* algorithm [112] to search for an optimal route between the start and the finish points. The target for parallelization is also the rules for the route planner. The well-known A* algorithm is basically a best-first search algorithm that keeps on expanding the current best node until a solution is found. Apparently, there are two simple strategies to parallelize the algorithm:

1. We can parallelize the calculation for the best node but still expand only the single best node.

2. In addition to the calculation, we can also expand more than one node, say n best nodes, in parallel.

The first one can be implemented simply by using the set selection conditions, ALL, or DISJOINT combinators to transform the cost calculation rules. The second one is less obvious since it needs to select n best nodes for expanding which is not as straightforward to specify as the former one. However, there is an alternative strategy similar to the second one that can be implemented directly. We can simply expand all best nodes instead of just one. In other words, all nodes with the same best f value (i.e. value of the heuristic function) are expanded. If there is only one such node, then only that node is expanded. If there exists more than one such node, all of them are expanded. This can be implemented simply by transforming the node expanding rule into an ALL rule.

The biggest problem in parallelizing the ARP program is the set of agenda and task control rules. These rules are strictly sequential unless we completely change the control mechanism. The latter is not desirable either since we want to have fair performance comparison between the sequential and the parallel programs. The good news is that the sequential effect reduces when the problem size increases. We therefore chose not to parallelize the control rules.

9.2 Experimentation Methodology

All programs, sequential and parallel, are running in three stages — input, computation, and output. Since parallel I/O is out of the scope of this research, we compare the performance of the computation part for both sequential and parallel execution. Furthermore, all experiments are conducted under the following guidelines which are equally applicable to both sequential and parallel execution.

- To minimize the cache effect, all data are collected after the caches have stabilized.
- Use the nondeterministic safe version of the sequential programs for the sequential time base.
- Measure the elapsed time of the computation part (i.e. right after the input of the last data and immediately before the output of the first result).

- Measure the mean and variance of at least 10 runs for each data point.
- Measure each program on at least 4 data sets of increasing size.

The following sets of experiments are repeated on each program:

- **Speedup Experiment**

  With fixed problem size, increase the number of processors from 1 to 12 [1]. Measure the speedup by comparing the execution time against the sequential execution time. This will demonstrate the overall speedup of the Venus/DA system.

- **Scaling Experiment**

  Repeat the speedup experiment on all 4 data sets. The purpose is to understand how well the system scales with respect to problem size.

- **Processor Utilization Measurement**

  With fixed problem size, 12 processors, keep track of the processor utilization throughout the execution. This complements the speedup experiment since we don't usually achieve good speedup without high processor utilization.

On separate runs, we also perform the following measurement in order to understand the behavior of the DA system:

- **Barrier Synchronization Overhead Measurement**

  With fixed problem size, 12 processors, measure the time spent exclusively on the barrier, i.e. the time when all processors are at the barrier. We want to understand if the barrier synchronization constitutes a sequential bottleneck.

- **CPU Time Distribution Measurement**

  With fixed problem size, 1 to 12 processors, measure the percentage of time each LE spent on join operations, fire instantiations, stack operations, synchronization, and others. Also average the numbers over all

[1] This is the number of actually usable processors on our Sequent Symmetry

LE's to obtain the average percentage of time the system spent on each type of computation. Both the individual measurement and the average tell us about how much time the system is doing real work and how much of the execution time is spent on overhead that does not occur on sequential version.

- **Stack Measurement**

  Over 4 data sets, on 1 to 12 processors, measure the number of stack operations (i.e. push and pop) done on each LE and the total number of stack entries on all LE over time. The LEAPS stack is the most important data structure in both the original LEAPS algorithm and the LEAPS/DA algorithm. This measurement will tell how much more entries and stack operations are performed by the DA system. It is another account for the overhead due to parallelism.

- **Join Measurement**

  Over 4 data sets, on 1 to 12 processors, measure the total number of join operations performed on each LE. This is intended to show the effectiveness of our intelligent backtracking scheme. Compare the sum of these measurement with the sequential number will demonstrate home much saving is obtained by the scheme. The result should reflect the speedup obtained.

- **Overhead Ratio Measurement**

  Over 4 data sets, on 1 to 12 processors, measure the number of key operations performed by the parallel and sequential systems. By measuring the ratio of these numbers, we should have a good picture of how much extra work is done by the parallel system and whether the extra work increases with data size.

9.3 Performance Results and Analysis

We present and analyze various performance results on three benchmark programs. We also compare the performance with simulation results whenever possible.

#PE's	16	32	64	128	256
Seq	0.776	5.491	58.101	771.751	11356.580
1	2.517	11.190	89.246	761.020	10697.210
2	1.576	6.488	48.500	397.595	5513.295
3	1.376	5.098	34.568	280.415	3831.823
4	1.244	4.482	28.572	222.555	2982.035
5	1.206	4.104	24.590	186.270	2500.277
6	1.272	3.994	22.320	162.066	2128.965
7	1.248	3.856	20.276	145.835	1897.205
8	1.340	3.850	19.092	131.701	1702.175
9	1.442	3.925	18.258	121.070	1512.250
10	1.516	3.978	17.208	111.280	1360.665
11	1.552	4.200	16.678	102.068	1213.335
12	1.755	4.272	16.100	91.575	1035.840

Table 9.2: MANNERS execution time(seconds) on increasing problem size.

9.3.1 Overall Speedup and Scaling Results

MANNERS

Table 9.2 and Figure 9.1 show the execution time and overall speedup results of the MANNERS program on problem size of 16, 32, 64, 128, and 256 guests. The first row denoted by Seq is the sequential execution time on each problem set. The second row is the execution time of the parallel program running on a single processor. Each execution time is the mean of at least 10 runs.

This set of experiments is a good indication of the effectiveness and scalability of the DA mechanism. First of all, the system exhibits good speedup behavior. For the 256 guests problem, we achieve 11-fold speedup over 12 processors. We also have the desired behavior of scalable speedup, both in terms of number of processors and problem size. When the problem size gets larger, which usually means more available parallelism, the system is capable of exploiting those parallelism to achieve higher level of concurrency. We note that when problem size gets larger than 128 guests, the speedup trend seems

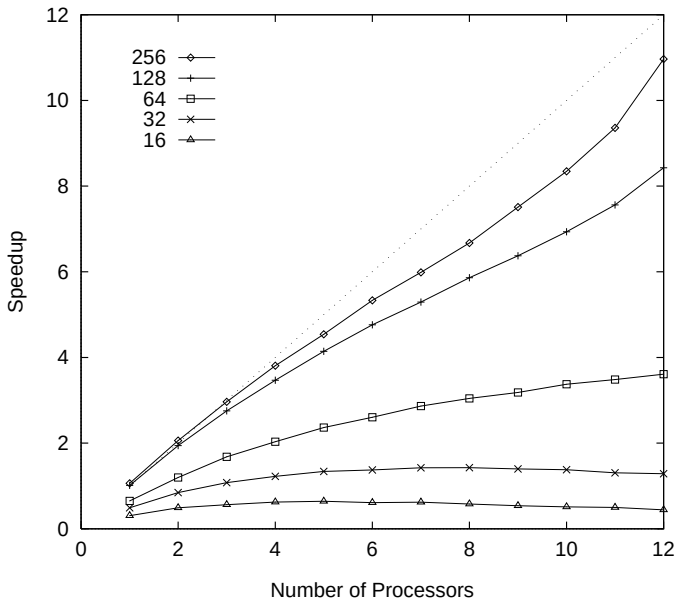

Figure 9.1: MANNERS overall speedup on different problem size.

to go up rather than down, suggesting that the system has the potential to achieve even higher speedup with more processors available.

WALTZ and ARP

To our surprise, we were unable to get comparable results on both WALTZ and ARP program. Since the reasons for failure are similar, we discuss them together.

Figure 9.2 is the results on WALTZ program. The problem size is the number of line segments in the input drawing to be labeled. We were only able to collect data for small problem sizes because the machine we worked on kept failing. Judging from the data we already have, there is no speedup what so ever. We face the similar situation on the ARP program as well which is depicted in Figure 9.3. The problem size in the ARP program is the number of possible points of the space that the route planner travels.

The reasons for the poor performance could be that the problem sizes are simply too small to allow the effect of parallel processing. When the problem size increases, we may have the similar results as the MANNERS program. However, detail analysis reveals several additional reasons that teach us important lessons about this research.

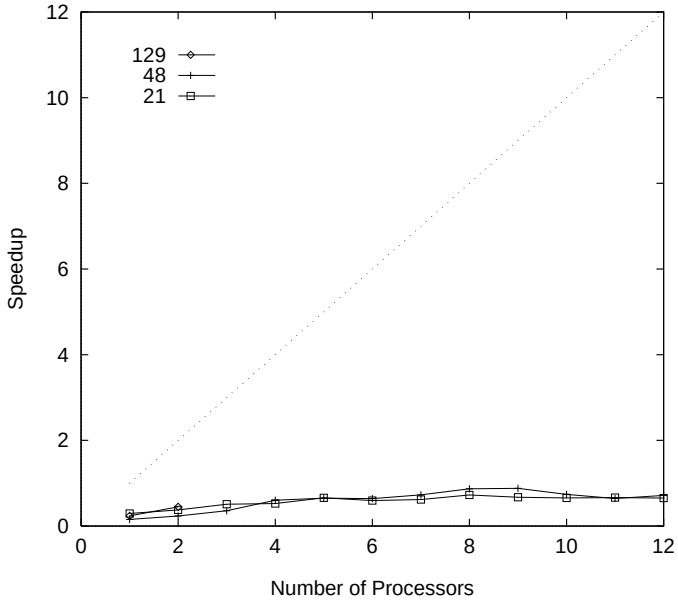

Figure 9.2: WALTZ overall speedup on different problem size.

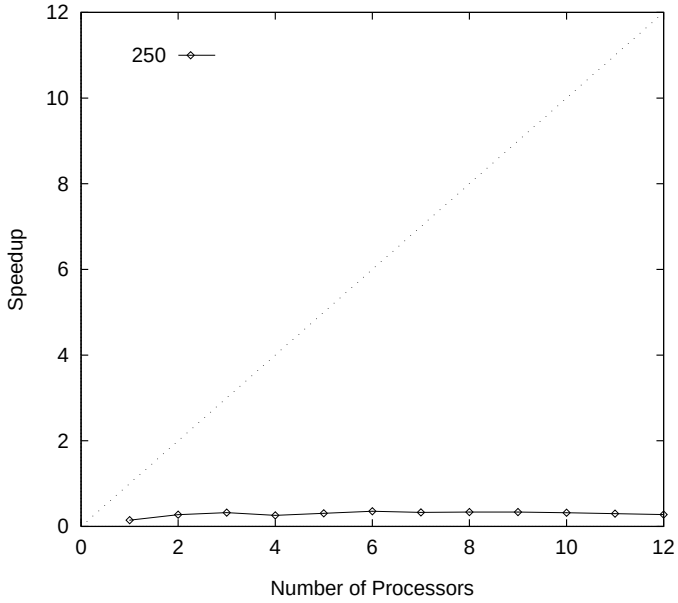

Figure 9.3: ARP overall speedup on different problem size.

- First and foremost, it is not that the DA approach fails to expose enough concurrency, but that there exists an unexpected search interference during the match phase in our implementation. This also explains why the effect doesn't show up in simulation since the simulation does not account for match cost. Take WALTZ for example, the program uses only three types of WME's. Since the LEAPS-based algorithm uses cursors to scan through working memory during the search for instantiations, all cursors are searching through the same three WME classes. What's worse is that in a critical phase, all active rules are accessing only a single class of WME's. Even though, from the decomposition specification, we know that these rules will eventually fire on disjoint sets of WME's and can fire in parallel without interference, they do interfere during the match. The asynchronous execution model makes the situation even worse since it is possible that some instantiation is removing a WME which happens to be scanning through by one or more concurrent search processes.

  We learn from this set of experiments that the nature of interference must be carefully studied, analyzed, characterized, and classified. Syntactic interference does not necessarily imply semantic interference, which is one of the main point of this thesis. On the other hand, semantic compatibility (i.e. non-interference) does not necessarily imply implementation compatibility. There is no free lunch in the world. As long as the concurrent processes have the potential of accessing the same set of data structures, they are still likely to interfere with each other.

  Nevertheless, this is not a problem that can't be solved. We have think of at least two possible solutions. The first one is to adopt a synchronous execution model such that the search and the firing proceed in two phases. That is, all parallel instantiations are found before firing any of them. We will certainly loss come efficiency but the interference discussed above is avoided since finding instantiations are read-only process. The second one is to use a delay update technique such that all updates are delayed until cycle synchronization point. This prevents the updates from interfering with the search. Some other techniques are certainly possible such as replication of WME's. The implementation techniques to fully exploit the available parallelism specified by decomposition abstraction are one of the immediate future work for this research.

- The second point is that the C&C technique has its limitation. The most significant drawback is that the copies increase the code size considerably. Large code size often results in heavy memory usage and poor

performance. The problem gets worse when the data size gets larger. This is one of the main reason why we can't run on large data sets for the WALTZ and ARP programs. A possible solution is to use parallel search instead of C&C. We already have ideas on how to do this which is part of our future work.

- Finally, the duplication of stack entries on multiple LEAPS engines is another important factor to explain the poor performance. This can be solved by better selection code and better partitioning of the rules.

As discussed earlier in this chapter, the ARP program has two factors that affect the performance considerably. The program has an agenda-task-subtask control mechanism which is strictly sequential. The program also expands one node at a time. Both of these constitute part of the reasons for the poor performance of the ARP program.

Since we can not obtain enough data from the execution of WALTZ and ARP for the problems discussed above, we measure various aspects of the MANNERS program to demonstrate the potential and to find possible solutions for the problems as well.

9.3.2 Processor Utilization Measurement

This set of measurement is to understand how well the RTS manages the computation resources. Because of the cycle execution model, we expect the utilization to be up and down frequently, especially when the concurrency is low. When the available concurrency is high, then the system should stay close to 100% utilization much longer than the synchronization point. We should see good performance in such case.

Figure 9.4 demonstrate the processor utilization graph of the system running with 12 processors on MANNERS program with 256 quests as input data size. The figure reflects the cycle execution of the run-time system. Note how the utilization graph corresponds closely to the concurrency profile of MANNERS, i.e. Figure 7.2. At the early stage of the execution, the available concurrency is low and so is the CPU utilization. This is evident by the dense area that begins the plot. When the available parallelism gradually increases toward the end, the utilization also increases since more and more parallel instantiations are available for execution. Since it does not take long to pass the period of low concurrency, we achieve good speedup on this benchmark program.

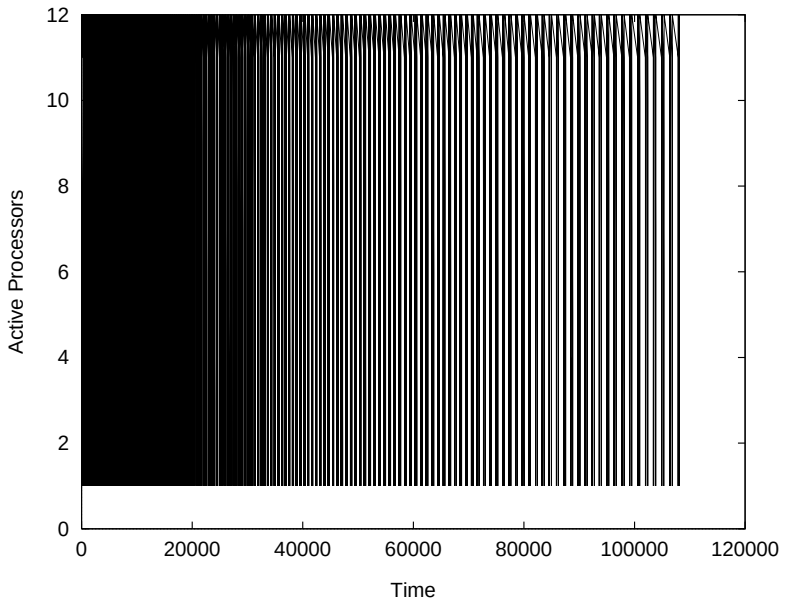

Figure 9.4: MANNERS256 processor utilization.

9.3.3 Behavior Measurement

We collect and present rest of the measurement in this section. The main purpose is to understand what exactly each LEAPS engine is working on during the course of execution. This will show us why we are getting good results or where the performance bottleneck is.

The following tables present the number and time statistics of the MANNERS program on different problem sets. In the tables, S stands for the sequential version and P stands for the parallel version. Performance figures on the same problem size are grouped into the same segment. For parallel version, we present the minimal, mean, maximal, and total numbers of all the LE's. Since the sum of execution time of all LE's is of little meaning, they are not listed in the time statistics table. Note that the execution time is a little higher than the speedup experiment presented in previous section because of the instrumentation added to collect these data. Also note that the LE's that run regular rules are in sleep state most of the time and therefore consume only a small portion of the computational resources. The mean values in the table that count the sleep time in do not represent the system behavior well. We therefore collect the mean values among those LE's that run parallel rules and listed as the ppmean.

In sequential execution, the number of stack entries popped is about the same and always smaller than the number of entries pushed. In parallel

Problem	#Push	#Pop	#Join	#Fire
S16	618	501	517	184
P16min	2	1	1	1
P16mean	46	93	47	8
P16max	75	358	98	18
P16total	886	1770	895	169
S32	2254	2022	2054	624
P32min	2	1	1	1
P32mean	169	392	187	31
P32max	269	1666	321	57
P32total	3229	7462	3564	593
S64	8604	8148	8212	2272
P64min	2	1	1	1
P64mean	703	1720	795	116
P64max	1102	7130	1297	198
P64total	13370	32686	15120	2209
S128	33404	32506	32634	8640
P128min	2	1	1	1
P128mean	2304	5850	2708	448
P128max	3609	25770	4337	737
P128total	43790	111157	51469	8513
S256	132410	130610	130866	33664
P256min	2	1	1	1
P256mean	9098	23482	10770	1758
P256max	14297	105905	17126	2838
P256total	172585	446167	204641	33409

Table 9.3: Number statistics of the MANNERS program.

Size	Join	Fire	Sync	Stack	Exec
S16	0.35	0.70	n/a	0.16	1.49
P16min	0.01	0.01	1.85	0.01	2.92
P16max	0.10	0.41	2.47	0.30	2.95
P16mean	0.03	0.08	2.08	0.09	2.91
P16ppmean	0.05	0.06	2.04	0.11	2.90
S32	3.67	2.23	n/a	0.64	7.60
P32min	0.01	0.01	3.52	0.01	5.57
P32max	0.50	0.70	5.10	0.74	5.64
P32mean	0.27	0.26	4.17	0.21	5.60
P32ppmean	0.41	0.29	3.87	0.27	5.60
S64	48.54	10.92	n/a	2.53	66.01
P64min	0.00	0.01	9.89	0.05	19.37
P64max	5.66	1.51	18.76	1.73	19.45
P64mean	3.47	0.91	12.88	0.71	19.40
P64ppmean	5.39	1.24	10.15	0.94	19.40
S128	714.59	58.94	n/a	11.69	805.62
P128min	0.01	0.01	32.27	0.06	101.69
P128max	56.20	4.81	101.19	6.04	101.76
P128mean	35.49	3.08	56.71	2.26	101.72
P128ppmean	55.56	4.48	33.23	2.62	101.72
S256	10980.69	379.30	n/a	48.86	11502.00
P256min	0.01	0.01	141.53	0.16	1088.55
P256max	876.49	26.58	1087.41	22.66	1088.63
P256mean	554.03	16.45	486.38	9.04	1088.59
P256ppmean	872.95	25.18	145.90	12.19	1088.58

Table 9.4: Time statistics of the MANNERS program.

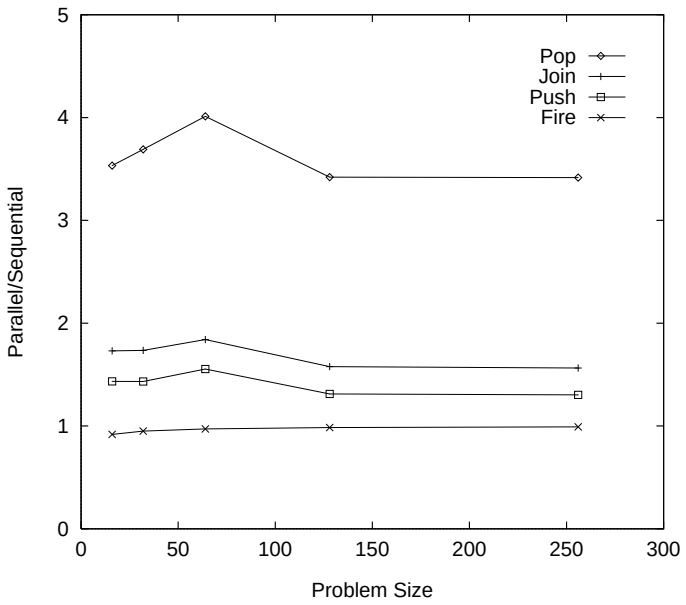

Figure 9.5: Parallel vs. sequential execution of MANNERS.

execution, however, we almost always have to do more popping than pushing. This is due to our asynchronous parallel execution model. At the start of each cycle, there is no way to know which rule will fire. All processors are therefore devoted to the search for instantiations. When some instantiations are found and fired, other concurrent search processes for incompatible rules have no reason to proceed. In the current implementation of the RTS, these LEAPS engines stop immediately and report at the barrier. The entries that have already been popped are restored in the stack adjustment process. Those extra search that are interrupted constitute the source of the extra pop's in the table. Nevertheless, the extra work due to parallelism is within a constant factor of the sequential execution work. The constant does not increase with the problem size, as depicted in Figure 9.5. This is a good indication of the scalability of our approach and implementation. Also note that the total number of rule firing of the parallel version is smaller than the sequential version because of the context mechanism employed which eliminates the context switching rules.

In both sequential and parallel executions, join time still constitutes a large portion of the total execution time. Good speedup results show that the RTS is spreading the load quite well. The existence of a perfect hash function for copy-and-constrain the only parallel rule in the program is also an important factor. We note that for smaller problem size, the synchronization time dominates all other items. This is because the concurrency is so low

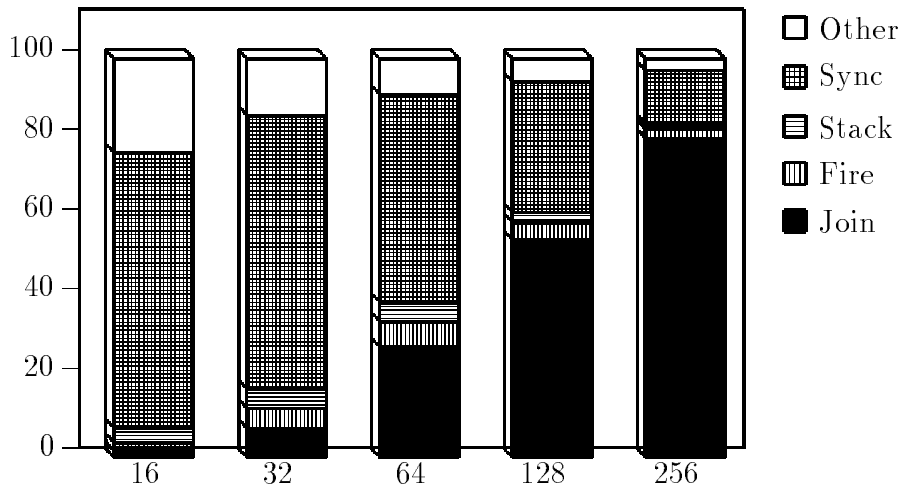

Figure 9.6: MANNERS CPU Time Distribution (Percentage).

that processors are synchronize with each other most of the time. But when problem size gets larger, we observe a significant shift of time distribution from synchronization to real work (i.e. join, fire, and stack operations). The speedup is also improved dramatically. In other words, when the concurrency is high enough, the RTS can indeed effectively exploit the parallelism to achieve good speedup. We demonstrate this shift of computational work load in Figure 9.6, which is the average percentage of time an LE spends on each type of work (i.e. join, fire, stack operations, synchronization, and the rest).

For a LEAPS-based implementation, one may think that the cost on the stack operations (i.e. push and pop) should dominate. This used to be the case in our early implementation. This is no longer the case because of the implementation of the context mechanism. The mechanism successfully focuses the search on only the active rules (i.e. rules belong to the current active contexts).

9.4 Remark

The numbers on MANNERS experiments are so good that it is hard to believe the system failed on the other two programs. However, we understand why it is the case and know how we may solve the problem. In general, this prototype implementation still demonstrates the potential of our approach.

The results presented in this chapter are both encouraging and disappointing. They are encouraging because decomposition abstraction can indeed reveal significantly higher degree of concurrency than pure syntactic approaches. They are disappointing because the implementation interference offset the available parallelism expressed by our mechanisms. However, the later problem can be solved by better software environment and implementation. A rewrite of the entire system should give us much better performance than what we have presented in this chapter.

Chapter 10

Conclusions and Future Work

In this research, we have identified and demonstrated that application semantics constitute the most critical source of parallelism in production systems. We established the decomposition abstraction approach as the foundation toward the organization and specification of semantic level parallelism in production systems. The set of decomposition abstraction mechanisms have been shown to be both effective in expressing application parallelism and easy to use. The use of functional dependency in the derivation of parallelism suggests the potential of a new direction which employs semantic analysis on data relationship specifications to extract application parallelism that may otherwise difficult to identify or specify. Both simulation and implementation results show that the combination of explicit specification and semantic analysis has the great potential of achieving the goal of massive and scalable speedup. Better implementation environment and techniques are necessary to fully exploit the parallelism expressed through decomposition abstraction.

The approaches and techniques developed in this research have applications in other areas besides parallelization of production rule systems. The decomposition abstraction is the missing layer which needs to be superimposed upon the familiar procedural and data abstractions to achieve truly portable parallel programming. We believe that it will be a necessary ingredient for data/knowledge based systems demanding high performance on large bodies of data and knowledge.

10.1 Future Work

Based on the experiences we gained from this research, further research in the following directions are promising both in terms of expressive power of the decomposition abstraction mechanisms and the performance of the implementation.

- *Aggregate operators in both antecedent and consequent.*

Aggregate operators should be allowed in both antecedent and consequent. This will provide the expressive power of specifying aggregate constraints in the antecedent.

- *Modularity instead of flat contexts.*

  The context mechanism should be generalized to introduce modularity into the decomposition abstraction process. The interplay between modularity and parallelism should be an exciting topic for further investigation.

- *Parallel search vs. copy and constrain.*

  Copy and constrain is effective but also has the problem of increasing code and image size. The subtlety of selecting rules and attributes for making the copies is another drawback of applying this technique. To facilitate parallel match in the LEAPS-based execution environment, a parallel search algorithm may be better.

- *Static partitioning and dynamic scheduling for load balancing.*

  Load balancing is critical to any parallel system. Effective exploitation of the available parallelism expressed by decomposition abstraction still require careful management of granularity and scheduling strategy. Statically partitioning of rules and dynamically scheduling of parallel executable instantiations based on the decomposition specification is a must addressed issue in any follow up research.

- *Functional dependency theory.*

  The notion of functional dependency introduced in this research is just a beginning. It captures a very familiar type of decomposition which is, in most cases, easy to specify. Other notions of dependency that characterize different types of decomposition are certainly possible. We expect a through investigation alone this line of research to establish a unify dependency theory that links the relationships between data objects to parallel decomposition.

- *Fine-grained parallelism in the object base.*

  We did not explore the potential parallelism of allowing multiple threads of execution within an object and other forms of type-specific concurrency. When application objects are large and complex as required by modern database applications, this level of parallelism may have significant impact on the system performance.

- *Integration of syntactic and semantic based techniques.*

  The semantic specification provided by decomposition abstraction should improve the CREL transformation and clustering results. The concurrency within each cluster should also increase. The integration of syntactic and semantic based techniques has the great potential of achieving better results with less help from the programmer.

- *Implementation strategies for distributed memory machines.*

  We avoid most of the locality issues with implementation on the Sequent Symmetry shared memory multiprocessor. When moved onto distributed memory machines and message passing paradigm, the implementation strategies are expected to be considerably different. It is a good research direction to see how semantic decomposition can be mapped onto distributed machine for efficient processing.

- *Software engineering issues.*

  The programming implication of decomposition abstraction is worthy of much attention. Decomposition abstraction is a natural successor of the familiar notions of procedural, control, and data abstractions. A good software engineering process that integrates all the abstraction mechanisms would be a significant contribution to the parallel programming research.

- *Fundamental issues.*

  Last, but certainly not the least, are a few fundamental issues that need to be looked into to have a better understanding of the nature of parallel processing both in production systems and in general.

  First of all, while syntactic non-interference may be overly conservative in the pursue of parallelism, semantic compatibility does not give us concurrency for free. For example, when syntactically interfering rules can be determined to be semantically compatible by our mechanisms, in the search for instantiations, we may still have the potential of interference on what ever data structures we search. Unless we replicate the data objects, we will always have the problem of accessing the same data object by multiple processes and the potential need for expensive locking. The replication is not necessary a good solution either because of the need to maintain consistency. A fundamental research is required to understand

the nature of interference. A classification of the degree of interference and the implication on implementation should provide a good foundation of efficient processing of shared data objects.

For this research, we also have the issue of adopting the LEAPS-based inference algorithm for the rule processing engines. A fundamental revision of the LEAPS/DA algorithm and an optimized selection mechanism are required to reduce the stack maintenance cost. In particular, the new algorithm should result in disjoint partitioning of the LEAPS stack rather than duplication of the stack entries as in the current implementation.

Finally, an effort should be made to classify and extract the core ingredients of decomposition abstraction mechanisms in all explicit parallel programming languages. Theoretical study on the expressive power of these mechanisms must be conducted. The results will bring us closer to, if not right on, the goal of easy to use and architectural independent parallel programming.

Appendix A

A Simple Course Scheduling System

A.1 Class Definitions

```
class  Course  {
    string   name,
    boolean  registrants_counted,
    int      registrants,
    boolean  scheduled,
    string   special_equip,
    string   instructor,
    string   classroom,
    string   time,
    boolean  printed
}
```

A.2 Functional Dependency Declarations

```
{ Department } --> { Classroom }
{ Department } --> { Course }
{ Department } --> { Student }
```

A.3 Context Declarations

```
SENIOR    |-  SPECIAL
POPULAR   |-  GET_INFO, SPECIAL
REGULAR   |-  GET_INFO, SPECIAL, SENIOR, POPULAR
PRINT     |-  SPECIAL, SENIOR, POPULAR, REGULAR
```

A.4 Rule Definitions

```
rule  Count_Registrants  in_context  GET_INFO  {
    ALL ( ( c : Course :: c.registrants_counted == NO ),
```

```
          [ s : Student :: c.name <| s.take* ] )
-->
    c.registrants = Count(s*),
    c.registrants_counted = Yes
}

rule  Schedule_Special  in_context  SPECIAL  {
    ( t : Time ),
    DISJOINT ( ( c : Course :: c.scheduled == NO
                      && c.special_equip != NULL ),
               ( i : Instructor :: c.name <| i.teaches*
                      && i.assigned < 3 ),
               ( r : Classroom :: t.time <| r.slots*
                      && c.special_equip <| r.equip* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.empty_slots* = r.empty_slots* - t.time
}

rule  Schedule_Senior  in_context  SENIOR  {
    ( t : Time ),
    DISJOINT ( ( c : Course :: c.scheduled == NO
                      && c.registrants < LIMIT ),
               ( i : Instructor :: i.is_senior == YES
                      && c.name <| i.teaches*
                      && i.assigned < 1 ),
               ( r : Classroom :: t.time <| r.slots* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time
}

rule  Schedule_Popular  in_context  POPULAR  {
    ( t : Time ),
```

```
    DISJOINT ( ( c : Course :: c.scheduled == NO
                      && c.registrants > THRESHOLD ),
               ( i : Instructor :: i.is_senior == NO
                      && c.name <| i.teaches*
                      && i.assigned < 3 ),
               ( r : Classroom :: t.time <| r.slots* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time
}

rule  Schedule_Regular  in_context  REGULAR  {
    ( t : Time ),
    DISJOINT ( ( c : Course :: c.scheduled == NO ),
               ( i : Instructor :: i.is_senior == NO
                      && c.name <| i.teaches*
                      && i.assigned < 3 ),
               ( r : Classroom :: t.time <| r.slots* ) )
-->
    c.instructor = i.name,  c.classroom = r.number,
    c.time = t.time,  c.scheduled = YES,
    i.assigned = i.assigned + 1,
    r.slots* = r.slots* - t.time
}

rule  Print_Result  in_context  PRINT  {
    ( c : Course :: c.scheduled == YES && c.printed == NO )
-->
    print_schedule(c),
    c.printed == YES
}
```

BIBLIOGRAPHY

[1] A. Acharya and M. Tambe. Production systems on message passing computers: Simulation results and analysis. In *Proc. IEEE Intl. Conf. on Parallel Processing Vol. II*, pages 246–254, 1989.

[2] G. Amdahl. Validity of the single-processor to achieving large-scale computer capabilities. In *AFIPS Conf Proc., Vol. 30*, pages 483–485, 1967.

[3] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] T. Anderson, E. Lazowska, and H. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Trans. on Computers*, 18(8):713–732, August 1989.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[6] Francois Bancihon and Setrag Khoshafian. A calculus for complex objects. In *Proc. 15th ACM Symp. on Principles of Database Systems*, pages 53–59, 1986.

[7] F. Bancilhon. Naive evaluation of recursively defined relations. Technical Report DB-004-85, Microelectronics and Computer Technology Corporation, 1985.

[8] F. Bancilhon, D. Maier, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 1–15, 1986.

[9] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[10] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. Electronic Computers*, 15:757–762, 1966.

[11] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[12] Brian N. Bershad. *The PRESTO Users Manual.* Department of Computer Science, University of Washington, Seattle, Washington 98195, 1991.

[13] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software - Practice and Experience*, 18(8):713–732, August 1988.

[14] Elisa Bertino, Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Trans. on Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[15] G. E. Blelloch. CIS: A massively concurrent rule-based system. In *Proc. 5th National Conference on Artificial Intelligence*, pages 735–741, 1986.

[16] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In *Proc. 17th Intl. Conf. on Very Large Data Bases*, pages 287–295, 1991.

[17] A. Brogi and P. Ciancarini. The concurrent logic language Shared Prolog. *ACM Trans on Programming Languages and Systems*, 13(1):99–123, 1991.

[18] James C. Browne and et. al. A new approach to modularity in rule-based programming. In *ICTAI'94: IEEE Intl. Conf. on Tools for Artificial Intelligence*, 1994.

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

[20] B. G. Buchanan and E. H. Shortliffe. *Rule-Based Expert Systems.* Addison-Wesley, Reading, MA, 1984.

[21] Special section on next-generation database systems. *Comm ACM*, 34(10), October 1991.

[22] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[23] Jean-Pierre Cheiney, G. Kiernan, and C. de Maindreville. A database rule language compiler supporting parallelism. Reports de Recherche No 1397, Institut National de Recherche en Informatique et en Automatique, 1991.

[24] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proc. 15th Intl. Conf. on Very Large Data Bases, Amsterdam*, pages 195–203, August 1989.

[25] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):76–90, March 1990.

[26] T. A. Cooper and N. Wogrin. *Rule Based Programming with OPS5.* Morgan Kaufmann, Palo Alto, 1988.

[27] F. Darema-Rogers, V.A. Norton, and G.F. Pfister. Using a single-program-multiple-data computation model for parallel execution of scientific applications. Research Report RC 11552, IBM T.J. Watson Research Center, Yorktown Heights, 1985.

[28] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In *Advances in Object-Oriented Database Systems: 2nd International Workshop on Object-oriented Database Systems*, pages 129–143, September 1988.

[29] U. Dayal et al. The HiPAC project: Combining active databases and timing constraints. *SIGMOD RECORD*, 17(1):51–70, March 1988.

[30] Christophe de Maindreville and E. Simon. Modelling production rules and query processing strategies with production compilation network. Research report, INRIA, November 1987.

[31] Christophe de Maindreville and E. Simon. A predicate transition net for evaluating queries against rules in a DBMS. Research Report 604, INRIA, February 1987.

[32] Christophe de Maindreville and Eric Simon. A production rule based approach to deductive databases. In *Intl. Conf. on Data Engineering*, pages 234–241, 1988.

[33] Lois M.L. Delcambre and James N. Etheredge. The relational production language: A production language for relational databases. In Larry Kerschberg, editor, *Expert Database Systems, Proc. 2nd Intl. Conf. on Expert Database Systems*, pages 333–351. Benjamin/Cummings, Redwood City, CA, 1988.

[34] Hasanat M. Dewan, Salvatore J. Stolfo, , Mauricio Hernández, and Jae-Jun Hwang. Predictive dynamic load balancing of parallel and distributed rule and query processing. In *Proceedings of the 1994 ACM SIGMOD International Confernece on Management of Data*, pages 277–288, 1994. Also appears in SIGMOD RECORD, 23(2):277–288, June 1994.

[35] Hasanat M. Dewan and Salvatore J. Stolfo. System reorganization and load balancing of parallel database rule processing. In *Proc. of 7th International Symposium on Methodologies for Intelligence Systems (ISMIS-93)*, Trondheim, Norway, June 1993.

[36] Hasanat M. Dewan, Salvatore J. Stolfo, and Leland Woodbury. Scalable parallel and distributed expert database systems with predictive load balancing. *Journal of Parallel and Distributed Computing*, 22(3):506–522, September 1994.

[37] R.O. Duda, P.E. Hart, K. Konolige, and R. Reboh. A computer-based consultant for mineral exploration. Technical Report Tech. Report; Final Report, SRI Project 6415, SRI International, September 1979.

[38] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. *Task Scheduling in Parallel and Distributed Systems.* PTR Prentice Hall, Prentice-Hall, Inc., 1994.

[39] Electronics Research Laboratory, University of California at Berkeley. *The POSTGRES Reference Manual, Version 2.1*, February 1991.

[40] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Comm ACM*, 19(11):624–632, November 1976.

[41] John E. Faust and Henry M. Levy. The performance of an object-oriented threads package. In *ECOOP/OOPSLA '90*, pages 278–288, 1990.

[42] C.L. Forgy. Note on production systems and Illiac IV. Technical Report Tech. Report CMU-CS-80-130, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1979.

[43] C.L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pittsburg, PA, 1979.

[44] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[45] C.L. Forgy, Anoop Gupta, A. Newell, and R. Wedig. Initial assessment of architectures for production systems. In *Proc. 4th National Conference on Artificial Intelligence (AAAI-84)*, pages 116–120, Austin, TX, August 1984.

[46] David Gadbois and Daniel P. Miranker. Discovering procedural execution of rule-based programs. In *Proc. National Conference on Artificial Intelligence*, 1994.

[47] H. Gallaire, J. Minker, and J.M. Nicolas. Logic and database: a deductive approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[48] S. Gallant. Connectionist expert systems. *Comm ACM*, 31:152–169, February 1988.

[49] J-L Gaudiot and A. Sohn. Data-driven parallel production systems. *IEEE Trans. on Software Engineering*, 16(3):281–293, March 1990.

[50] N.H. Gehani and W.D. Roome. Concurrent C. *Software - Practice and Experience*, 16(9):821–844, 1986.

[51] Douglas N. Gordin and Alexander J. Pasik. Set-oriented constructs for rule-based systems. In *Proc. 7th Conference on Artificial Intelligence Applications*, pages 76–80, Miami Beach, FL, 1991.

[52] Douglas N. Gordin and Alexander J. Pasik. Set-oriented constructs: From Rete rule bases to database systems. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 60–67, 1991.

[53] Anoop Gupta. Implementing OPS5 production systems on DADO. Technical Report CMU-CS-84-115, Department of Computer Science, Carnegie Mellon University, Pittsburgh, December 1983.

[54] Anoop Gupta. Measurement on production systems. Technical report, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1984.

[55] Anoop Gupta. *Parallelism in Production Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, March 1986.

[56] Anoop Gupta, Charles Forgy, and Allen Newell. High-speed implementation of rule-based systems. *ACM Trans on Computer Systems*, 7(2):119–146, May 1989.

[57] Anoop Gupta, M. Tambe, D. Kalp, C.L. Forgy, and A. Newell. A parallel implementation of OPS5 on the encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2), 1989.

[58] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[59] L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. ACM SIGMOD*, pages 377–388, 1989.

[60] Robert H. Jr. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans on Programming Languages and Systems*, 7(4):501–538, October 1985.

[61] Wilson Harvey, Dirk Kalp, Milind Tambe, David McKeown, and Aleen Newell. The effectiveness of task-level parallelism for production systems. *Journal of Parallel and Distributed Computing*, 13(4):395–411, December 1991.

[62] F.D. Highland and C.T. Iwaskiw. Knowledge base compilation. In *Proc. 11th International Joint Conference on Artificial Intelligence IJCAI-89*, 1989.

[63] D. Hillis and G. Steele. Data parallel algorithms. *Comm ACM*, 29:1170–1183, 1986.

[64] S. Hiranandani, K. Kennedy, C. Koebel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[65] C.A.R. Hoare. Monitors: An operating system structuring concept. *Comm ACM*, 17(10):549–557, October 1974.

[66] M. Hsu, R. Ladin, and D. McCarthy. An execution model for active database management systems. In *Proc. 3rd Intl. Conf. on Data and Knowledge Bases*, pages 171–179, June 1988.

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

[68] T. Ishida. Methods and effectiveness of parallel rule firing. In *Proc. IEEE 6th Conference on Artificial Intelligence Applications*, pages 116–122, 1990.

[69] T. Ishida. Parallel firing of production system programs. *IEEE Trans. on Knowledge and Data Engineering*, 3(1):11–17, March 1991.

[70] T. Ishida and Salvatore J. Stolfo. Toward the parallel execution of rules in production system programs. In *Proc. IEEE Intl. Conf. on Parallel Processing*, pages 568–574, 1985.

[71] C. Kellogg, A. O'Hare, and L. Travis. Optimizing the rule-data interface in a KMS. In *Proc. 12th Intl. Conf. on Very Large Data Bases, Tokyo, Japan*, pages 42–51, August 1986.

[72] M.A. Kelly and R.E. Seviora. A multiprocessor architecture for production system matching. In *Proc. National Conference on Artificial Intelligence (AAAI-87), Vol I*, pages 36–41, 1987.

[73] M.A. Kelly and R.E. Seviora. An evaluation of DRete on CUPID for OPS5 matching. In *Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, August 1989.

[74] G. Kiernan and C. de Maindreville. Compiling a rule database language into a C/SQL application. In *Intl. Conf. on Data Engineering*, pages 388–395, April 1991.

[75] G. Kiernan and C. de Maindreville. A rule language compiler for SQL database servers. *Data and Knowledge Engineering*, 7(2):89–113, December 1991.

[76] G. Kiernan, C. de Maindreville, and E. Simon. Making deductive database a practical technology: A step forward. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 237–246, 1990.

[77] D. Klappholtz, X. Kong, and A. D. Kallis. Refined Fortran: An update. In *Proc. of Supercomputing '89*, Santa Clara, CA, November 1989.

[78] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, January 1988.

[79] Chin-Ming Kuo. *Parallel Execution of Production Systems.* PhD thesis, Department of Computer Sciences, University of Texas at Austin, May 1991.

[80] Chin-Ming Kuo, Daniel P. Miranker, and James C. Browne. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13(4):424–441, December 1991.

[81] S. Kuo. *An Asynchronous Message-Driven Parallel Production System.* PhD thesis, Department of Electrical Engineering-Systems, University of Southern California, August 1991.

[82] S. Kuo, D. Moldovan, and S. Cha. Control in production systems with multiple rule firings. In *Proc. IEEE Intl. Conf. on Parallel Processing, 1990, Vol. II*, pages 243–246, 1990.

[83] S. Kuo, D. Moldovan, and S. Cha. MCMR: A multiple rule firing production system model. In *5th International Parallel Processing Symposium*, pages 256–259, 1991.

[84] Steve Kuo and Dan Moldovan. Implementation of multiple rule firing production systems on hypercube. *Journal of Parallel and Distributed Computing*, 13(4):383–394, December 1991.

[85] Steve Kuo and Dan Moldovan. The state of the art in parallel production systems. *Journal of Parallel and Distributed Computing*, 15(1):1–26, May 1992.

[86] J.E. Laird, A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[87] Ho Soo Lee and Marshall I. Schor. Match algorithms for generalized Rete networks. *Artificial Intelligence*, 54(3):249–274, April 1992.

[88] B. Lindsay and L. Haas. Extensibility in the Starburst experimental database system. In A. Blaser, editor, *Database Systems of the 90s*, pages 217–248. Springer-Verlag, 1990.

[89] B. Lindsay, J. McPherson, and H. Pirahesh. A data management extension architecture. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 220–226, 1987.

[90] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[91] Guy M. Lohman, Bruce Lindsay, Hamid Pirahesh, and K. Bernhard Schiefer. Extensions to Starburst: Objects, types, functions, and rules. *Comm ACM*, 34(10):94–109, October 1991.

[92] David Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

[93] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 215–224, 1989.

[94] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[95] D.M. McKeown, W.A. Harvey, and J. McDermott. Rule based interpretation of aerial imagery. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 7(5):570–585, 1985.

[96] Daniel P. Miranker. TREAT: a better match algorithm for AI production systems. In *National Conference on Artificial Intelligence*, pages 42–47, 1987. Also appears as Tech. Rep. AI TR87-58, Dept. of Computer Science, Univ. of Texas at Austin, 1987.

[97] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* PhD thesis, Department of Computer Science, Columbia University, 1987. Also published by Morgan-Kaufmann, 1990.

[98] Daniel P. Miranker. Special issue on the parallel execution of rule systems: Guest editor's introduction. *Journal of Parallel and Distributed Computing*, 13(4):345–347, December 1991.

[99] Daniel P. Miranker and David A. Brant. An algorithmic basis for integrating production systems and large databases. In *Proc. 6th Intl. Conf. on Data Engineering*, pages 353–360, February 1990. The LEAPS algorithm.

[100] Daniel P. Miranker, David A. Brant, B.J. Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *National Conference on Artificial Intelligence*, pages 685–692, July 1990.

[101] Daniel P. Miranker, C. Kuo, and J. Browne. Parallelizing compilation of rule-based programs. In *Proc. IEEE Intl. Conf. on Parallel Processing, Vol. II*, pages 247–251, 1990.

[102] Daniel P. Miranker, Chin-Ming Kuo, and James C. Browne. Parallel transformations for a concurrent rule execution language. Technical Report TR-89-30, Department of Computer Science, University of Texas at Austin, October 1989.

[103] Daniel P. Miranker and B.J. Lofaso. The organization and performance of a TREAT based production system compiler. *IEEE Trans. on Knowledge and Data Engineering*, 3(1):3–10, March 1991.

[104] Daniel P. Miranker, B.J. Lofaso, G. Farmer, A. Chandra, and David A. Brant. On a TREAT based production system compiler. In *Proc. 10th Proc. Intl. Conf. on Expert Systems*, pages 617–630, Avignon, France, June 1990.

[105] D. Moldovan. A model for parallel processing of production systems. In *Proc. IEEE Intl. Conf. on Systems, Man and Cybernetics*, pages 568–573, 1986.

[106] D. Moldovan. RUBIC: A multiprocessor for rule-based systems. *IEEE Trans. on Systems, Man, and Cybernetics*, 19(4):699–706, 1989.

[107] David A. Mundie and David A. Fisher. Parallel processing in Ada. *IEEE Computer*, 19(8):20–25, August 1986.

[108] S. Naqvi and S. Tsur. *A Language for Data and Knowledge Bases.* W.H. Freeman Publ., 1989.

[109] Daniel E. Neiman. *Design and Control of Parallel Rule-Firing Production Systems.* PhD thesis, University of Massachusetts at Amherst, September 1992.

[110] Daniel E. Neiman. Parallel OPS5 user's manual and technical report. Technical Report COINS TR 92-28, Computer and Information Sciences Department, University of Massachuseetts, April 1992.

[111] Daniel E. Neiman. Issues in the design and control of parallel rule-firing production systems. *Journal of Parallel and Distributed Computing*, 23(3):338–363, December 1994.

[112] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, CA, 1980.

[113] K. Oflazer. *Partitioning in Parallel Processing of Production Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1987. Also appears as Tech. Rep. CMU-CS-87-114, March 1987.

[114] K. Oflazer. Highly parallel execution of production systems: A model, algorithms and architecture. *New Generation Computing*, 10(3):287–313, 1992.

[115] David A. Padua, D. J. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Computer*, 29(9):763–776, September 1980.

[116] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm ACM*, 29(12):1184–1201, December 1986.

[117] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[118] Alexander J. Pasik. *A Methodology for Programming Production Systems and Its Implications on Parallelism.* PhD thesis, Columbia University, 1989.

[119] Alexander J. Pasik. A source-to-source transformation for increasing rule-based system parallelism. *IEEE Trans. on Knowledge and Data Engineering*, 4(4):336–343, August 1992.

[120] L. Raschid, T. Sellis, and C. Lin. Exploiting concurrency in a DBMS implementation for production systems. In *Proc. Intl. Symp. Databases in Parallel and Distributed Systems*, pages 34–45, 1988.

[121] *Proceedings of RIDE'94: Workshop on Research Issues in Data Engineering/Active Database Systems*. IEEE Computer Society Press, February 1994.

[122] James G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. Technical Report 89-5, Department of Computer Science, Tufts Univ., Medford, MA, October 1989.

[123] James G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13(4):348–365, December 1991.

[124] J.G. Schmolze. An asynchronous parallel production system with distributed facts and rules. In *Proc. AAAI-88 Workshop on Parallel Algorithms for Maching Intelligence and Pattern Recognition*, St. Paul, Minn., August 1988.

[125] J.G. Schmolze and S. Goel. A parallel asynchronous distributed production system. In *Proc. AAAI–90*, pages 65–71, 1990.

[126] M.I. Schor, T.P. Daly, H.S. Lee, and B.R. Tibbitts. Advances in Rete pattern matching. In *AAAI-86*, pages 226–232, 1986. YES/RETE.

[127] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. 17th VLDB*, pages 469–478, September 1991.

[128] F. Schreiner and G. Zimmermann. PESA I — a parallel architecture for production systems. In *Proc. IEEE Intl. Conf. on Parallel Processing*, pages 166–169, 1987.

[129] P.M. Schwarz, W. Chang, J.C. Freytag, G.M. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst database system. In *International Workshop on Object-Oriented Database Systems*, pages 85–92, 1986.

[130] Timos Sellis, Chih-Chen Lin, and Louiqa Raschid. Implementing large production systems in a DBMS environment: Concepts and algorithms. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 404–412, 1988.

[131] Timos Sellis, Chih-Chen Lin, and Louiqa Raschid. Data intensive production systems: The DIPS approach. *SIGMOD RECORD*, 18(3):52–58, November 1989.

[132] Eric Simon and Christophe de Maindreville. Deciding whether a production rule is relational computable. In *Proc. Intl. Conf. on Database Theory*, pages 205–222, 1988.

[133] A. Sohn and J-L Gaudiot. Connectionist production systems in local and hierarchical representation. In N. Bourbakis, editor, *Applications of Learning and Planning Methods*, pages 165–180. World Scientific Publishing Co., Teaneck, NJ, 1990.

[134] A. Sohn and J-L Gaudiot. Connectionist production systems in local representation. In *Proc. IEEE Intl. Joint Conf. on Neural Networks Vol II*, pages 199–202, Washington, DC, January 1990.

[135] A. Sohn and J-L Gaudiot. A macro actor/token implementation of production systems on a data-flow multiprocessor. In *Proc. 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, August 1991.

[136] A. Sohn and J-L Gaudiot. A survey on the parallel distributed processing of production systems. *International Journal on Artificial Intelligence Tools*, 1(2):279–331, June 1992.

[137] Salvatore J. Stolfo. Five parallel algorithms for production system execution on the DADO machine. In *National Conference on Artificial Intelligence*, pages 300–307, 1984.

[138] Salvatore J. Stolfo. Initial performance of the DADO2 prototype. *IEEE Computer*, 20(1):75–83, January 1987.

[139] Salvatore J. Stolfo, Hasanat M. Dewan, and Ouri Wolfson. The PARULEL parallel rule language. In *Proc. IEEE Intl. Conf. on Parallel Processing*, volume II, pages 36–45, 1991.

[140] Salvatore J. Stolfo and David P. Miranker. The DADO production system machine. *Journal of Parallel and Distributed Computing*, 3:269–296, 1986.

[141] Salvatore J. Stolfo, David P. Miranker, and R. Mills. A simple processing scheme to extract and load balance implicit parallelism in the concurrent match of production rules. In *Proc. AFIPS Symp. on Fifth Generation Computing*, 1985.

[142] Salvatore J. Stolfo and D.E. Shaw. DADO: A tree-structured machine architecture for production systems. In *Proc. 2nd National Conference on Artificial Intelligence (AAAI-82)*, 1982.

[143] Salvatore J. Stolfo, Ouri Wolfson, Philip K. Chan, Hasanat M. Dewan, Leland Woodbury, Jason S. Glazier, and David A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4):366–382, December 1991.

[144] J.M. Stone. Nested parallelism in a parallel fortran environment. Technical Report Research Report RC 11506, IBM T.J. Watson Research Center, Yorktown Heights, 1985.

[145] M. Stonebraker et al. The implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[146] M. Stonebraker, E. Hanson, and Chin-Heng Hong. The design of the POSTGRES rules system. In *Intl. Conf. on Data Engineering*, pages 356–374, 1987.

[147] M. Stonebraker, E. N. Hanson, and S. Potamianos. The POSTGRES rules system. *IEEE Trans. on Software Engineering*, 14(7):897–907, July 1988.

[148] M. Stonebraker, E. N. Hanson, and S. Potamianos. A commentary on the POSTGRES rules system. *SIGMOD RECORD*, 18(3):5–11, September 1989.

[149] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching, and views in data base systems. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 281–290, 1990.

[150] M. Stonebraker and Greg Kemnitz. The POSTGRES next-generation database management system. *Comm ACM*, 34(10):78–92, October 1991.

[151] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proc. 1986 ACM-SIGMOD Intl. Conf. on Management of Data, Washington, D.C.*, pages 340–355, June 1986.

[152] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.

[153] D.S. Touretzky and G.E. Hinton. Symbols among the neurons: Details of a connectionist inference architecture. In *Proc. International Joint Conference on Artificial Intelligence*, pages 238–243, August 1985.

[154] S. Tsur and C. Zaniolo. LDL: a logic based data language. In *Proc. 12th Intl. Conf. on Very Large Data Bases, Tokyo, Japan*, pages 33–41, August 1986.

[155] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 and 2. Computer Science Press, 1989.

[156] Michael van Biema, Daniel P. Miranker, and Salvatore J. Stolfo. The do-loop considered harmful in production system programming. In Larry Kerschberg, editor, *Expert Database Systems*, pages 177–190. The Benjamin/Cummings Publishing Company, Inc., 1986.

[157] Larry Wall and Randal L. Schwartz. *Programming perl*. O'Reilly & Associates, Inc., 1991.

[158] D. L. Waltz. Understanding line drawings of scenes with shadows. In P Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, New York, NY, 1975.

[159] Jennifer Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 275–285, September 1991.

[160] Jennifer Widom and Sheldon J. Finkelstein. A syntax and semantics for set-oriented production rules in relational database systems. *SIGMOD RECORD*, 18(3):36–45, September 1989.

[161] Jennifer Widom and Sheldon J. Finkelstein. Set-oriented production rules in relational database systems. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 259–270, 1990.

[162] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Mass., 1989.

[163] W.A. Woods. Knowledge base retrieval. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 179–195. Springer-Verlag, New York, 1986.

[164] Shiow-yang Wu and James C. Browne. Explicit parallel structuring for rule based programming. In *Proc. 7th International Parallel Processing Symposium*, pages 479–488, 1993.

[165] C. Zaniolo. The representation and deductive retrieval of complex objects. In *Proc. 11th Intl. Conf. on Very Large Data Bases, Stockholm*, pages 458–459, 1985.

[166] C. Zaniolo. Safety and compilation of non-recursive horn-clauses. In *Proc. First Intl. conf. on Expert Database Systems*, pages 237–252, 1986.

[167] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1991.

VITA

Shiow-yang Wu was born in Hualien, Taiwan, the Republic of China, on June 1, 1962, the son of Shoei-yun Wu and Chuen-huey Chang. After completing his secondary education at Hualien High School in 1980, he entered National Chiao Tung University with majoring in Computer Engineering in Hsinchu, Taiwan. He received a Bachelor of Sciences degree from National Chiao Tung University in June 1984. In September 1984, he entered the master program of Computer Engineering in National Chiao Tung University. He received a master of sciences degree in June 1986. From 1986 to 1988, he served as the associate engineer in the Institute of Information Industry in Taipei, Taiwan. In August of 1988, he entered the doctoral program in Computer Sciences at University of Texas at Austin.

He has been married to Hsin-yi Chen since 1988. They are the parents of Alexander, born in March 26, 1992 and Douglas, born in November 16, 1994.

Permanent address: F 5 No. 5 Lane 97 Yenchi St.
Taipei, Taiwan, R.O.C.

This dissertation was typeset with LaTeX[1] by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The and revised by Tsan-sheng Hsu.