Parallel Execution of Production Systems

Chin-Ming Kuo

Department of Computer Sciences
The University of Texas at Austin

TR-92-42 November 1992

PARALLEL EXECUTION OF PRODUCTION SYSTEMS

by

CHIN-MING KUO, 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

May, 1991

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Browne, for the past 6 years of guidance. This thesis will not be complete without his encouragement and confidence in me. Special thanks to my co-advisor, Dr. Miranker for providing the critical comments during various stages of this research. I would also like to thank the members of my committee, Dr. Mok, Dr. Kumar, Dr. Malek, and Dr. Werth for their suggestions and comments. Last but not least, I would like to thank my wife, Ing-Chin, for her patient, confidence, and supports in every aspects of my study in UT.

CHIN-MING KUO

The University of Texas at Austin

May, 1991

PARALLEL EXECUTION OF PRODUCTION SYSTEMS

Publication No._____

Chin-Ming Kuo, Ph.D.
The University of Texas at Austin, 1991

Supervising Professor: James C. Browne and Daniel P. Miranker

The production system or rule-based system paradigm is a widely used form of building expert systems or artificial intelligence applications involving knowledge representation and knowledge base search. As the complexity and size of expert system applications expand, parallelizing production systems becomes an attractive approach in attempts to speedup the executions.

Previous attempts at parallel structuring of rule-based programs have failed to achieve desired levels of parallelism. This research is a comprehensive approach to the parallelization of rule-based programs. The structure of the productions systems is examined and static analysis techniques are developed. Several sources of run-time parallelism are developed to restructure the cases where the dynamic behaviors of the systems can not be detected by the compile-time analysis.

We propose a new production system language (CREL), more suitable for parallel implementation, to allow asynchronous execution. Static dependency analysis and dependency graph transformations are developed to explore potential parallelism at compile-time. Through compile-time dependency analysis and

transformations, rules that are not closely related are partitioned into clusters for asynchronous executions. At run-time, each cluster acts essentially as an independent rule system, with its own match-select-act cycles. Medium-grained parallelism is explored inside each cluster at three different levels – match, run-time consistency checking, and conflict resolution. A simple run-time management scheme is used to balance the system load.

Currently the CREL system is implemented on a Sequent's Symmetry shared-memory system. Despite fair amounts of overhead in synchronization, message passing, management of the complex run-time structures, and garbage collection, etc., the CREL system achieves an order of magnitude of speedups on a set of OPS5 benchmarks when compared with a sequential implementation based on the best known match algorithm. The performance gain increases significantly as the sequential constraints embedded in the programs were removed. We anticipate significant results for CREL application programs designed specifically for parallel execution.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Production Systems play an important role in building expert systems and artificial intelligence applications involving knowledge representation and state space search[41],[6],[56]. The current execution speed of production systems is insufficient to support extensive developments of real applications. As the use of rule-based system applications expands into new domains, there is also a growing need for a faster execution model of production system programs.

The introductory chapter defines production systems, identifies the sources of parallelism in such systems, and describes related issues. We then give a summary of our approaches, which are detailed in the chapters that follow. We also give, in the last two section, a preview of the results of this research, and the general directions for future research.

1.1 Production Systems : OPS5 as an Example

Before proceeding to the discussions of the OPS5 syntax, we would like to mention a few words regarding terminology. Although the context of this research is not specifically limited to the language of the OPS5, we will to use special terms from the OPS5 convention for consistency. In the following sections, special terminology from the OPS5 language is presented in boldface and accompanied by its description. After the introduction, we will assume it is clear that terms such as a **WME** represent a Working Memory Element, or a memory cell as a general term.

OPS5[15] is chosen as our core language because of its popularity and available applications. An OPS5 production system program is composed of a set of production rules called **rules**, a set of data elements called **Working Memory Elements (WME)** and an underlying execution mechanism (interpreter) governing the execution of the program.

WMEs are essentially the same as variables in conventional programming languages. WMEs are grouped by **class** predicate types, the first attribute field in a WME. Each class contains a fixed number of additional attributes to represent various characteristics of a complex object in the system.

Each production rule consists of two parts: a left-hand-side (**LHS**) and a right-hand-side (**RHS**). A LHS contains a list of **condition elements** (**CEs**), with each CE term consisting of a list of **test conditions**. The RHS contains a set of **actions** that perform updates to the working memory. The production rules represent long-term knowledge of a system to control the state transitions, while the WMEs store short-term "transient" knowledge of the system. From conventional programming languages' point of view, each production rule is an **if–then** statement where the LHS and RHS correspond to the **if** and **then** parts, respectively.

The test conditions in a LHS CE consist of a number of boolean operations on variables. A variable is always *free* in its first occurrence in a rule. All subsequent occurrences of the same variable in the same rule are always bound. Each LHS CE can be positively or negatively referenced in conjunctive normal forms. Negative CEs means "there exists no WME satisfying the -CE conditions". The example given below illustrates the use -CEs. Variable bindings are achieved by specifying the same variable in different CE terms. The entire LHS is also called a **condition list**.

In the example rule given below, we have two CEs in the LHS, one positive

and one negative. The variable bindings are done explicitly in atr3 of ClassB WME, and implicitly between atr1 of classA and atr4 of ClassB.

An action in a RHS contains operators such as **modify**, **make**, and **remove**, to perform updates to the working memory. All actions in a production rule's RHS are implicitly ANDed so that all actions must be executed when the instantiation is selected for firing. Variable binding between different attributes is achieved through specifying the same variable, as the $< x >$ in the following example. Details regarding the OPS5 syntax and semantics can be found in [15], [6]. An OPS5 rule and its interpretation is given below as an example:

```
(P Pi
 (classA âtr1 <x>        âtr2 <y>)
-(classB âtr3 (<z>=<y>) âtr4 <x>)
 -->
 (modify 1 âtr1= <x> + <y>      ))

Given two class A and B with attributes atr1 to 4. Rule
Pi matches all WMEs of classA, M, and there exists no
Class B element N so that atr1 of M equals atr4 of N
and atr2 of M equals atr3 of N. If such M exists, modify it.
```

According to the history of the development of production systems[65], the original execution semantics of a production system is as follows:
All rules are active at any time. Execution is carried out by repetitively looping through the **recognize-act** cycle. During each execution cycle, each rule performs tests on its LHS against the working memory. The "conflict set" is then formed by collecting all the instantiations[1]. During the conflict resolution phase, the most promising instantiation from the conflict set is selected as the instantiation to fire. At the act phase, actions from the RHS of the selected instantiation are executed and

[1] An instantiation is the collection of WMEs that satisfy the LHS match conditions.

the updates are reflected back to the working memory. The execution cycle repeats until the conflict set is empty or an explicit **halt** statement is encountered.

The type of computation performed in each execution cycle can be categorized as the following:

- **Match**

  In the match phase, each rule evaluates its LHS test conditions on the working memory. The match work in production systems resembles the query processing in relational database systems (equivalence between the relational algebra in Relational Database theory and the LHS condition list is given in Chapter 3).

  The computational cost of the match operation can be high because the arbitrary forms in variable bindings among CEs may result in cross products among a large number of WMEs.

  From the standpoint of state-space searches, since backtracking is not allowed in production systems, match phase is important because it insures all the possible search paths in the huge space will always satisfy the test conditions and always lead to correct solutions.

- **Conflict Resolution**

  In the conflict resolution phase, instantiations in the conflict set are selected for firing. Since the resolution strategy has significant effects on the performances of the system, various conflict resolution [2] strategies were suggested[42]. In OPS5 for example, strategies such as **specificity** and **recency** are designed as heuristics for sequential execution.

[2]We shall use the terms "selection" and "conflict resolution" interchangeably.

In OPS5, a time-tag is associated with every WME to reflect the *recency* of the WME, i.e., how recently the WME was updated. The time-tag of an instantiation is defined as the maximum time-tag of all WMEs that constitute the instantiation. The OPS5 *recency* strategy uses such time-tags as the criteria of conflict resolution.

Specificity, on the other hand, is an integer value associated with every instantiation to measure the complexity of the test conditions of its LHS. *Specificity* strategy means to select the instantiation from the conflict set with the highest number of test conditions in its LHS. Both *recency* and *specificity* perform well as search heuristics on sequential OPS5 implementations.

- **Act**

  The RHS actions of the selected instantiation are executed and changes are reflected to the global working memory. Typical actions in a RHS include creation of a new WME, changing attribute values of WMEs, and removing WMEs.

Up to now, we have only used OPS5 as an example to describe the syntax and semantics of a generic production system language. There are many strategies one can use in the three phases above to make the execution model tailored to specific environments.

1.2 Parallelism in Production Systems

Production Systems, in their generic form, appear to have a high degree of parallelism since all rules in the system are active and perform the **recognize-act** cycle simultaneously. Regardless of the conflict resolution strategies and the execution models, the sources of parallelism in production systems can be categorized

by the different phases in the execution cycle as follows:

1. **Match Parallelism**

   Given a LHS of a rule, the test conditions can be represented as a discrimination network where updates to WMEs are represented as dataflow tokens flowing through the network. Regardless of the degree of the state-saving property of the match algorithm, different granularity [3] will result in different degrees of parallelism. One extreme is to treat, for instance, the entire match network as a single computation unit. The other extreme is to distribute every test node in the network across physical processors to achieve more match parallelism. Chapter 4 discusses in details the issues related to parallel match.

2. **Rule Parallelism**

   In the conflict resolution phase, if the semantics of the production system allows parallel firing, multiple instantiations from either the same rule or different rules in the conflict set satisfying certain constraints can be fired in parallel, resulting in rule-level parallelism, or production parallelism. Since different conflict resolution strategies may produce different execution paths for a program, the design of the rule-level parallelism model require changes of definitions of both the language and its execution model.

3. **Action Parallelism**

   After instantiations are selected, their actions can be executed in parallel. Again, certain restrictions must be enforced to insure the integrity of the working memory state and correctness of the execution.

---

[3] By granularity, we mean a unit of the match work. See Chapter 2 for more details.

Other potential sources of parallelism in production system exections include pipelining between the execution stages [22] and sharing of the match network, among others. Due to the complexity of the problem as a whole, we will concentrate on improving the above items in our research. In addition, an important characteristic of production systems is that the execution cycle time[4] is small compared to the overhead of communications and synchronization among the parallel tasks, it is not necessarily beneficial to exploit every source of parallelism.

1.3 Summary of the CREL Approaches

So far, we have presented backgrounds and definitions of production systems. We also listed various sources of parallelism and their corresponding issues. In this summary, we briefly describe the new approaches and rationales with respect to the language, the execution model, the static analysis, and the run-time aspects of parallel execution of production systems.

The status of parallelizing production system research can best be described by the following statements[22] in explaining the limited speedup obtained in the PSM project, centered on the Rete based parallel match:

1. For each execution cycle, the number of WMEs changed per cycle is small in terms of the percentage of the total number of WMEs.

2. Even for such a small number of WMEs being affected per firing, the number of rules affected is again small. The multiple effects of points 1 and 2 do not make parallelizing matching on sequential OPS5 a promising approach.

[4]The match, conflict resolution, and act cycle.

3. There is a large variation in the processing requirement (mainly match phase) among the active rules. Since there must be a global synchronization point for each cycle, all active processors must wait for the match phase of the 'hottest' rule to complete the current cycle, thus degrading the overall processor utilization.

Although other projects, which will be described in the next chapter, tried to reduce the effect of point (3) by using the intra-rule parallelism to speed up all active rule matchings, the overall gain is limited by the fact of match time occupying only on the average of 50% instead of 90% of the total execution time. There has been no attempt to improve points (1) and (2).

To correct these three points, we propose the following approaches: we first change the semantics of OPS5 to allow more parallelism and asynchronous execution. Static dependency analysis is then applied to the rule systems. Dependency graph transformations are applied to the dependency graphs in order to optimize the static analysis. Sources of run-time parallelism are identified and various techniques of function partitioning are applied to the clusters in the systems to exploit run-time parallelism and balance the system loads.

With the new semantics of CREL defined, one would be able to design parallel production systems with well-structured blocks of rules representing methods to solve subgoals of a problem. This will relieve the effects of point (1) and (2). Parallel rule firing can also reduce the effects of (1) and (2) by increasing the number of WME changes per unit time. Effective rule decomposition in graph transformations as well as load balancing techniques in run-time can reduce the effect of match time variance for point (3).

It is our belief that to achieve further improvements in speeding up production system execution, new methods must be designed together with existing

suitable algorithms for various stages in the execution. Only when the combination of best algorithms/techniques in various stages of the execution is applied can the overall benefits be multiplied by individual improvements.

1.4 Summary of the CREL Results

The technical innovations of the CREL systems can be divided into the categories of the language and execution model, the static analysis and transformations, run-time management, and a parallel match based on the Treat algorithm.

The CREL language and its execution model eliminate the recency resolution strategy and allow asynchronous execution and multiple rule firings. The static analysis and transformations identifies, at compile time, clusters of rules that can be executed independently and asynchronously. The CREL parallel match algorithm achieves match parallelism at the rule level while avoiding the need of fine-grained locks on the WMEs and related structures. The CREL run-time checking algorithm performs consistency checking among instantiations at run-time to allow multiple firing of these instantiations at the same time while maintaining the correctness of the execution. The CREL run-time management system insures the partitioning of work and balances the overall system loads, while maintaining minimum amounts of overhead.

We present a set of performance measurements on the actual CREL systems and perform analysis on these statistics. The observations made in Chapter 7 can be reiterated as follows. We first observe that because of the improvements in the compilation techniques of the sequential match algorithm, the focus of one's attempt to parallelize production systems has been changed. Specifically, match no longer is the primary target for parallelization.

We then observe that the CREL's static analysis and clustering technique

produce limited results. This can be attributed primarily to the use of control variables in the benchmark systems. Other factors include the declarative nature of production systems, where rules are written to cover the entire system through free variables. As a result, the information obtained at compile time has limited effectiveness. The static clustering technique, however, establishes the basis of the CREL execution model and leads to the elimination of the OPS5 recency strategy.

The most significant improvement in the CREL system results from the run-time checking to allow multiple rule firing (MRF) in one cycle. The effectiveness of MRF is significant in the original benchmark programs, and is further enhanced in the cases of replicated systems. We observe that through MRF, the cycles of a system can be significant reduced. Since the execution cycles introduce synchronization overhead in every parallel system, the focus of attention in parallelizing production systems should be to reduce the total number of execution cycles, instead of reducing the cycle time.

Last but not least, is the observation that hashing is very effective in every aspect of the system. The results from the replicated systems described earlier, which is a form of hashing, support such an observation.

1.5 Future Directions

In our view, further research on parallel production systems can be categorized as follows. Because of the shifts of focus in parallel production systems, as observed in this research, efforts should be made toward the investigations of developing new match algorithms and data structures that are more appropriate for parallel or distributed processing. Because of significance of the effects from the use of control variables, efforts should be made toward development of and experiments a systematic method for CREL programming, both in terms of expressing parallelism

and the issue of correctness. Because of the proven benefits of MRF, efforts should be made toward more efficient algorithms for the run-time checking process. Because of the limited information available to the static analysis, efforts should be made to provide a higher level descriptions to assist the static analysis process. Because of the overhead in the current implementation, especially the join code sizes, efforts should be made toward the development of compilation techniques that do not cause near exponential growth of the executable image sizes. Last but not least, special attention should be focused on techniques to fully utilize the hashing concept, both in the area of reduction of match complexities, and in the area of providing more concurrency for the CREL system.

Chapter 2

Related Works

This chapter discussed some previous work and the key issues in parallelizing production systems. We also discuss the strengths and weakness of each research project and conclude with a brief description of the organization of the chapters that follow.

2.1 Previous Works

In this section, we described the history of the development of production systems. Early work on production systems concentrated on refining conflict resolution strategies [42]. *Recency* was suggested as a heuristic for efficient state space search and proved to be a good strategy for sequential implementations. The OPS5 language was then designed to standardize the language used for production systems and the construction of rule-based expert systems. Efficient sequential match algorithms then became the primary research focus. Recently, parallel implementations became feasible because of the advances in VLSI technology and parallel processing research. Examples are the DADO project at Columbia University [62] and the PSM project at Carnegie-Mellon University [24]. The former concentrates in parallelizing match work, specifically the RETE algorithm. The latter focuses on effective implementations of various AI applications, including production systems, on DADO's special tree structured architecture. Although both projects are excellent research efforts, the end results from both projects failed to achieve the anticipated

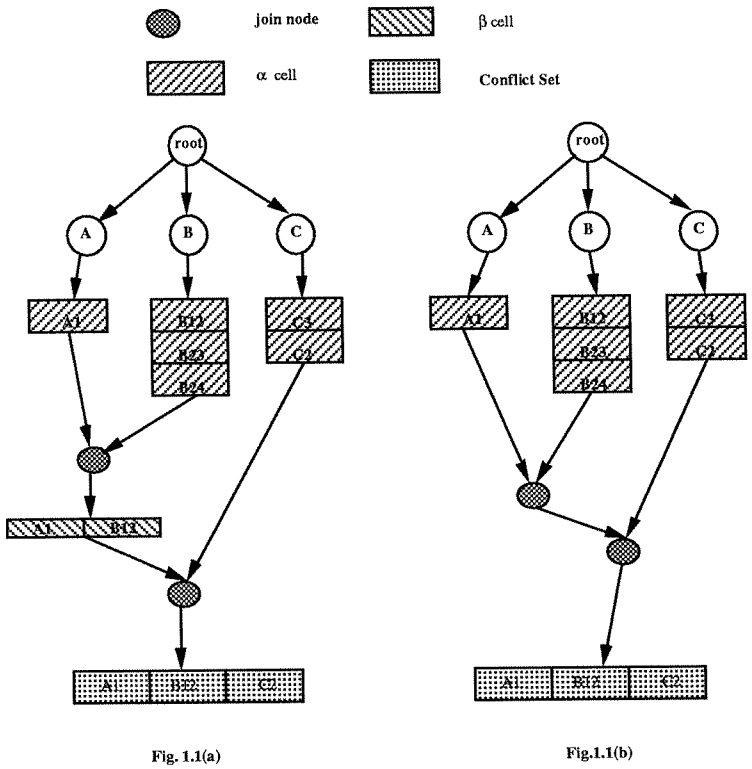

Figure 2.1: Examples of the RETE and the TREAT networks.

speedups.

2.2 Related Research Categorized by the Key Issues

According to the sources of parallelism, the key issues and the corresponding strategies in improving the performance of production system execution can be categorized as the low-level match, production system partitioning and distribution, and parallel firing and synchronization [47]. These topics are elaborated separately as follows.

2.2.1 Low-Level Match

In general, the percentage of the working memory affected by the firing of one rule is small, so one should restrict to as small a set of rules as possible to perform match work in the next execution cycle. The RETE match algorithm[16] was invented first to restrict the set of rules necessary to perform match in each cycle, and secondly to reduce the overhead of match work by storing intermediate match states of previous cycles in a RETE network so the matching can be done incrementally. Fig 1.1(a) gives an example of a RETE network.

The TREAT algorithm[46] is another incremental match technique that takes advantages of the match networks. The key idea of TREAT comes from the observation that the majority of the nodes have duplicate information stored in a RETE network, thus redundant work is performed every time an update token flows through the network.

Under sequential implementations, although the worst case time complexity of TREAT is the same as RETE, TREAT outperforms RETE in both space and average time complexities. We will later discuss in detail the tradeoff between these two match algorithms in our parallel implementation. Fig. 1.1(b) gives an example of a TREAT network for the purpose of comparison.

2.2.2 Partitioning and Distribution

Orthogonal to the match parallelism (also known as intra-rule level parallelism) is the rule-level parallelism. Oflazer[50] first studied the aspect of production system partitioning as a means of increasing parallelism. The basic idea is that since the percentage of working memory affected by a rule firing is small, a production system can be partitioned into disjoint sets so that rules from the same set can be distributed to different physical processors for efficient parallel match work. Although

multiple rule firings is not allowed in this approach, it is still classified as exploiting the rule-level parallelism.

One of the drawbacks with Oflazer's approach is that if the semantics of the language remains the same, i.e., the system still support only single rule firing. Therefore, at any instance, the set of "interesting" rules in an execution state is small compared to the number of rules in the entire system. [1] Another potential problem with this approach is the conflicts of interests in distributing the partitions. On one hand, rules in the same partition tend to become active at the same time; thus, distributing them onto separate physical processor elements(PEs) will reduce the overall match time. On the other hand, rules in the same partition tend to make references/updates to the same classes of WMEs; consequently, allocating them in the same PE will reduce the communication overhead.

2.2.3 Parallel Firing and Synchronization

Most of the existing work in parallelizing production systems concentrates on the match phase[23],[24], [21], [61], [56]. Presumably this is attributable to the assumption that the match work takes up to 90% of the entire computation[25].

However, recent results on the performance of a highly optimized TREAT based OPS5 compiler invalidates this long standing assumption[40]. The *ops5c* compiler improves the execution speed of production systems in an order of magnitude compared to compiled RETE based implementation. Detail Analysis further revealed that the average match time is reduced to less than or equal to 50%.

An immediate implication of this observation is, regardless of parallel or sequential implementations, the maximum speedup one can achieve is 2 if the

[1] average less than 30 rules vs. a full system of 300 to 3000 rules.

improvement is solely from the match work. Thus parallel firing, among other sources, must be incorporated into the system to achieve substantial improvements.

Ishida first studied the potential of allowing multiple instantiations to fire in each cycle [29]. He suggested a method similar to the standard compile-time dependent analysis by first constructing interference graphs and traversing the graphs to locate cycles of dependency. All instantiations in the cycles can not be fired at the same times. Schlmoze presented a formal model to solve this problem with a similar approach in [58], i.e., to build the interference graph and finding cycles. The unique feature of this approach is that he developed a set of algorithms with different run-time costs and degrees of optimality. Experiments were performed on a set of benchmark programs against the set of the algorithms to study the combined aspects of run-time checking and run-time overhead on overall system performance.

These related works on multiple rule firing failed to recognize the relationships between the language semantics, the execution model, and the run-time overhead of multiple rule firing. For example, associated with the parallel firings, issues of synchronization among simultaneously fired rules need to be addressed to insure the correctness of the execution.

2.2.4 Discussions

We need to point out that the issues in the above categorization are not independent subjects. Instead, they are closely related. The design and implementation of a match algorithm, for example, has profound effects on the granularity of the match parallelism as well as on the design of the rule and action level parallelism. To successfully build a parallel environment for production systems, in additional to the match parallelism, we have to incorporate the rule-level parallelism to obtain additional speedup, address the problem of global execution cycle synchronization,

and take into consideration the combined effects of these sources of parallelism on the overall system performance.

2.3 Other Related Works

The logic programming paradigm[10], [11] resembles production systems in many ways. Both systems use pattern matching/variable bindings to test a **if** guarded condition, and apply changes to the global state by the **then** action statements.

A fundamental difference between the two paradigms, however, is the difference between forward chaining and backward chaining systems. Specifically, logic programs perform resolutions on a set of facts. The state of the system remains constant in the resolution process. In other words, logic programs describe only the static states of the systems. Production systems, on the other hands, compute results by matching and then updates to the global system state. The state of the system, therefore, is changing during the execution.

Since the facts represented by a logic program are static and the actions and the execution of the logic program are to infer additional facts from existing ones by a resolution mechanism, the sequence of free variable bindings in logic program executions does not change the global state of the system. Therefore, the synchronization problem is kept minimum. On the other hand, for production systems, it is the "update" actions such as **make**, and **modify**, that make parallel execution difficult because one needs to be concerned with the sequencing problem.

There is also various work on Equational programming in the design of parallel programs [8], conforming time constraints [44] or static prediction of time bounds of real-time decision systems[5]. Production systems resemble these systems in that the fundamental execution models are the same. The existence of

pattern matching variables in LHSs and potentially unbounded numbers of WMEs in production systems, however, make the efficient evaluation of LHSs the focus of attention. We shall concentrate on the design of parallel execution techniques specific to production systems, without worrying about the real time aspects.

2.4 Organization of the Thesis

After the definitions of production systems and brief descriptions of the related work, Chapter 3 discusses the language issue and the execution model. Chapter 4, then presents the static analysis, transformations of the dependency relations, and the clustering techniques. Chapter 5 deals with the issues in run-time parallelism and management. Chapter 6 gives a global picture of the integrated CREL system and presents some pseudo codes to give a better illustration of the actual CREL implementation. Other related implementation issues are also discussed. Chapter 7 presents the results of the CREL performance measurements and observations based on these results. Chapter 8 gives the concluding remarks and directions for future research.

Chapter 3

The CREL and its Execution Models

There are three major aspects in parallelizing production systems: the language, its execution model, and an effective run-time management strategy. In order to explicitly or implicitly express parallelism, we need a suitable production system language. To best express the behavior of the execution, an execution model is needed. Lastly, effective run-time management strategies should be applied to the system in order to maximize utilization of the available resources. The first two aspects, the language and the execution model, are the primary topics of this chapter. Run-time parallelism and process management issues will be discussed in Chapter 5 and 6.

3.1 Unified Computation Model

Taking the approach unique to those of the previous efforts to parallelize production systems as described in Chapter 1, we consider the production system model in terms of the unified computation model[4]. Given a rule-based program $\mathbf{P}$, with global state variable $\mathbf{V}$ representing the state of the system, we classify a rule, P_i, in $\mathbf{P}$ as a schedulable computation unit (SCU). The SCU consists of two basic functions: a match module (the match condition list LHS_i), and a action module (the action list RHS_i). The execution of the system is such that each SCU (or rule), P_i, synchronously evaluates its match function on the current working memory, as in $LHS_i(V)$. A set of WMEs satisfies the LHS_i constitutes an "instantiation" for rule

P_i. After the match work for all rules complete, instantiations of all rules form the conflict set. In the conflict resolution stage, the prominent one is selected from the conflict set for execution. Assuming instantiation I of rule P_i is selected, the updates from executing the RHS actions of $RHS_i(I)$, are reflected to the working memory. The execution cycle repeats until the conflict set becomes empty or the system is explicitly halted.

There are many dependency relations between SCUs in a production system. In the model described above, synchronization between all SCUs during execution cycles is a type of control dependency. Actions from one rule resulting in a change of the match conditions of another rule is a type of data dependency. The conflict resolution process is yet another type of dependency relation, where simple edges between rules are unable to model its behavior.

In the following sections, production systems can be modeled through stepwise abstraction, from the language, through the execution model, and to the run-time management. The justification of the abstraction is "separation of concerns", meaning a production system language should follow the conventional definitions of production systems from the system designers' viewpoints. The programmers should not be concerned with the underline execution model. Likewise, the execution model should best reflect an effective execution behavior and the semantics of the language, without concerns about the physical execution environments. The abstraction then simplifies the dependency analysis, and minimizes the conceptual difference between the conventional production system languages such as the OPS5 and the CREL due to the changes in the language semantics.

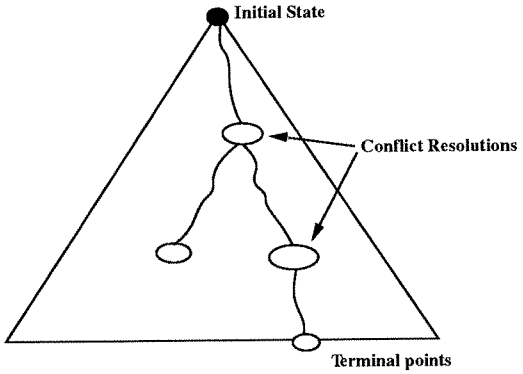

Figure 3.1: Execution paths of a production system.

3.2 CREL

If we consider the execution of a production system as a state-space search, with the search alternatives being the selection alternatives in each conflict resolution, as in Figure 3.1, a specific conflict resolution strategy will dictate the search path. Strategies such as *recency* and *specificity* in the OPS5, for example, uniquely determine the execution path for a given production system program in a depth first search based on time-tags.

In order to allow multiple rule firings in each cycle, the first question one would ask is : "which set of instantiations can be fired in parallel to achieve the same result as in sequential execution?" An immediate answer is : a set of instantiations working on disjoint parts of the working memory space can be fired in parallel without disturbing one another[50].

For compile-time analysis, this means tests should be performed on all WME classes appearing in the RHS and LHS of every rule against that of all other rules to partition the entire system into disjoint sets. Experiments on typical OPS5 benchmark programs show that this technique are impractical based on the following

facts. In production system programming, data items are modeled by WME classes and rules responsible for processing the same class of data items tend to become active at the same time, thus partitioning of rules based on the data dependency tends to results in strongly connected graphs.

Although the partitioning approach resulted in little improvement for parallel rule firing. We extend the basic idea of the partitioning rules to the CREL system. The combined results of the rule partitioning and the changes to the semantics – relax the *recency* strategy in the conflict resolution process – can provide significant parallelism to the system. Removing the *recency* strategy introduces nondeterminism in the systems. To insure a correct execution model, we define that a parallel execution of a production system is correct if and only if it is serializable[64]. A parallel execution is serializable if there exists a sequential execution path resulting in the exact same final state.

Given a starting state of a production system, a sequential execution can be uniquely characterized as a sequence of rule firings. Since the nondeterminism only exists in the conflict resolution stages, meaning match and actions are deterministic, the sequential execution path can also be uniquely represented by the sequence of rule selections during the conflict resolution stages. The same reasoning applies to the cases of parallel execution, except there are multiple instantiations selected for firing per cycle.

In order to simplify the concept of serializability, we define serializability in terms of a single cycle in Definition 1. To facilitate the definitions and discussion, some notations are defined as follows:
Given a production system program $\mathbf{P}$ with N rules, $\mathbf{P}= \{P_1, P_2, \ldots P_N\}$, we define a parallel firing E_j , of $\mathbf{P}$ in cycle $\mathbf{j}$, as the set of instantiations selected for firing in cycle $\mathbf{j}$:

$$E_j = \{I_{j_1} \parallel I_{j_2} \parallel \ldots \parallel I_{j_m}\} \tag{3.1}$$

where m is the number of instantiations in E_j, I_{j_k} s an instantiation of rule P_{j_k}. For each j,k, we further define C_{j_k} as the set of instantiations from rule P_{j_k} in cycle j, the conflict set is $\cup_{\forall k} C_{j_k}$.

Definition 1 Serializability

E_j *is serializable if and only if there exists a serialized execution path of* E_j, E'_j, *such that* E'_j *produces the exact same result as* E_j. □

Notice first that the subscript j of E'_j does not represent the cycle count of the serial execution since there is no one-one correspondence between the cycle counts of parallel and serial executions. Secondly, one should be aware of the difference in representing E_j and E'_j since E_j is a set and E'_j is a sequence of E_j.

With the definition of serializability given above, it can be shown that an execution path E of $\mathcal{N}$ cycles where $E = \{E_1 \rightarrow E_2 \ldots \rightarrow E_{\mathcal{N}}\}$ is serializable if and only if $\forall j \in [1..\mathcal{N}]$, E_j is serializable. This is because both parallel and sequential execution models are globally synchronized.

Notice that in Definition 1 on serializability, a particular execution path is represented by a set of instantiations. Because of the lack of backtracking in production systems, consistency checking (of whether a parallel execution path is serializable) should be performed on *all* combinations of instantiations in the current conflict set to insure serializability of the selected parallel execution path. Such a consistency checking can be a potential bottleneck when the size of conflict set and the number of rules of a system is large.

Therefore, we perform a static analysis on all the rules in a system before run-time to identify the sets of rules that can be fired in parallel without invalidating

the serializability requirement. Following Definition 1, the objective of such static analysis is to check for a given set of rules, a parallel execution of any combinations of all instantiations from the conflict set should be serializable. This can be done by inspecting the syntactic structures among rules interacting with each other. We will discuss in detail in later sections.

Before we continue the subject of static analysis techniques, we shall give a clear definition of the changes we make to the OPS5 languages and the effects of these changes. As mentioned earlier, Oflazer's approach on partitioning production system programs does not produce plausible results. In addition to the lack of block structure in production system languages, the limited improvement of parallelism in all previous approaches is also attributable to the unique sequential execution path dictated by the OPS5 conflict resolution strategies. To increase the degree of parallelism, we modify the OPS5 language and define the CREL as follows:

- Syntax

  The CREL syntax is exactly the same as the OPS5, including syntax definitions, WME class declarations, RHS, LHS, etc.

- Conflict Resolution Strategy

  Rather than using the *recency* and *specificity* from the OPS5, the CREL selection strategies are designed for more parallelism:

  Specificity

  Given instantiations from two rules P_i and P_j, an instantiation of P_i has higher priority for being selected than those of P_j if and only if the number of tests in LHS_i is greater than that of LHS_j. Since *specificity* is powerful in constructing synchronization points in production system programs, it is included in the CREL selection strategies.

Nondeterminism

In addition to *specificity*, selection among remaining instantiations in the conflict set is nondeterministic under the fairness assumption that all satisfied instantiations will eventually be selected for firing in finite time.

Since there are multiple execution paths for a CREL program, we need to define the correctness of CREL programs.

Definition 2 Correctness of CREL programs

A CREL program is **correct** *if and only if all legal serial execution paths reach correct terminal states.* □

Notice that in terms of language itself, other than the conflict resolution strategy, the execution behavior of a CREL program does not change from the OPS5. In other words, restrictions such as global execution cycles and single rule firing per cycle still exist.

From the programmer's point of view, in constructing a correct CREL program and to increase the degree of parallelism, one should not rely on *recency* and should insure all possible execution paths always lead to a correct terminal state.

3.3 The CREL Execution Models

In this section, we gradually relax the execution constraints in the CREL execution model to allow multiple rule firings and no global synchronization, while maintaining the correctness of the execution by insuring serializability as defined earlier.

With the CREL formally defined, again we ask the question : "Under a multiple rule firing strategy, which sets of rules can be fired in parallel without

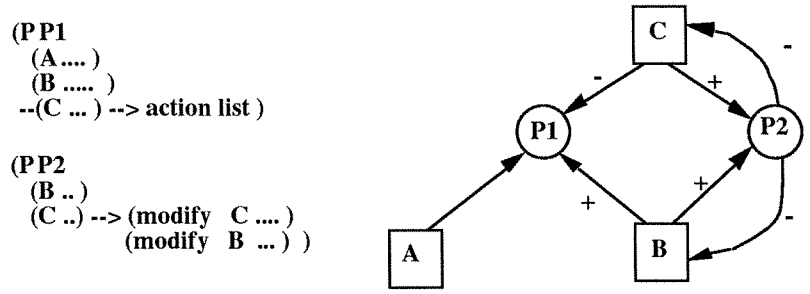

Figure 3.2: A bipartite data dependency graph example.

violating the correctness of the execution?" To answer such a question at compile-time, we need to identify the sets of rules in which parallel firing does not guarantee serializability. We define such rule sets as the mutual exclusion sets.

3.3.1 Bipartite Data Dependency Graph

A mutual exclusion dependency relation occurs when there are conflicts in accessing the same data between rules. In order to express the interactions among rules and WMEs, we adopted the bipartite data dependency graph representation originated by Ishida[30] as follows: Given a production system program, the mutual exclusion dependency graph G_m is defined as $G_m = (V, E)$ where

Nodes V can be classified into rule-nodes and pattern-equivalent WMEs nodes (WME-nodes) as:

Rules

Rules correspond to rule-nodes, represented by "circle"(◯) symbols in graph G_m.

Pattern-equivalent WMEs

A "pattern-equivalent WMEs" is a set of "similar" WMEs that is refer-

enced or updated by production rules. Pattern-equivalent WMEs nodes are represented by "square" (□) symbols in graph G_m.

Edges E represent the types of data dependency relations between pattern-equivalent WMEs and rules. Edges can be further classified into "referenced" and "changed" types of edges as:

(+/–) referenced

A type of edges from WME-nodes to rule-nodes. A **referenced** edge is drawn from a WME-node, O_i, to a rule , P_i, if O_i appears in the LHS of rule P_i. The edge carries a positive(negative) sign if O_i appears in P_i's positive(negative) CEs.

(+/–) changed

A type of edges from rule-nodes to WME-nodes. A **changed** edge is drawn from a rule, P_i, to a pattern-equivalent WME set, O_i, if O_i appears in the the RHS of P_i. The edge carries a positive sign if O_i appears in P_i's **make** action elements. The edge carries a negative sign if O_i appears in P_i's **modify** or **remove** actions.

Figure 3.2 gives an example of the bipartite data dependency graph, where rule P_1 has a positive reference to the WME-node W_1 (class A) and the WME-node W_2 (class B). P_1 also has a negative reference to WME-node W_3 (class C). Rule P_2, on the other hand, has a positive reference and a negative change to W_2 and a positive reference to W_3. The bipartite data dependency graph G_m in Figure 2.2 tells us that at compile-time, without knowing in advance the values bound to the free variables in rule P_1 and P_2, the firings of P_2 may interfere the match work of P_1 because of W_2.

3.3.2 Mutual Exclusion Set

A mutual exclusion set is defined to be a set of rules in a production system that cannot be executed in parallel. In other words, parallel firing of all rules in the same mutual exclusion set cannot be serialized. In order to compute the mutual exclusion sets, we need to identify the conditions prohibiting the serializability requirements.

Recalling Definition 1 on serializability on two rules, P_1 and P_2 for example, the firing sequence [1] of $\{P_1, P_2\}$ is not a valid serial execution of parallel execution $\{P_1 \parallel P_2\}$, if the firing of one rule, say P_2, invalidates some instantiations of the other rule, in this case, P_1. Because of the duality of the problem, we will only analyze the cases of rule P_2 interferes with rule P_1.

Assume at time t=0, the parts of the conflict set at t=0 originated from rule P_1 and P_2 are C_0^1 and C_0^2, respectively. The following two cases are the potential situations where firing of one rule invalidates instantiations from the other rules, using the bipartite data dependency graph representation:

(A) $P_1 \overset{+}{\leftarrow} W_1 \overset{-}{\leftarrow} P_2$

Rule P_2 is deleting or modifying an WME-node (W_1), which is also positively referenced by rule P_1. In other words, the firing of P_2 *may* delete some entries of W_1, which may constitute parts of the current conflict set of P_1.

(B) $P_1 \overset{-}{\leftarrow} W_1 \overset{+}{\leftarrow} P_2$

Rule P_2 is making an WME-node (W_1), which is also negatively referenced by rule P_1. In other words, the firing of P_2 creates some entries of W_1, which may invalidate some instantiations of P_1 because of negative references.

[1] in terms of rules as opposed to instantiations

Definition 3 Interference

Cases (A) and (B) described above are defined as two types of interference between two rules, where the firing of one rule, P_2*, interferes the match work of the other,* P_1*.* □

Given a pair of rules, P_1 and P_2, only a single incident of the two cases above will not constitute a serializability problem, since for parallel execution $\{P_1 \parallel P_2\}$, a valid serial execution such as $\{P_1, P_2\}$ still exists while $\{P_2, P_1\}$ is not a valid one.

The two types of interference, therefore, impose a total ordering on the serial execution sequence of the rules involved in the interference. If interference, such as ($P_1 \overset{+}{\leftarrow} W_1 \overset{-}{\leftarrow} P_2$), exists between two rules, P_1 and P_2, any valid serial execution should follow the total ordering that P_1 proceeds P_2. Notice that such ordering is transitive.

Algorithm DP-ANALYSIS1, can be used to calculate the mutual exclusion sets in a production system:

Algorithm 1 DP-ANALYSIS1

Given a bipartite data dependency graph G_m *of a production system, traverse* G_m *starting from any rule-node. A mutual exclusion set is formed if and only if*

1. *the mutual exclusion set forms a minimum cycle in* G_m*, and*
2. *there are conflicts in the serial ordering of all rules in the cycle because of interferences.* □

Figure 3.3 gives examples of opposite cases of cycles where rules within clusters are mutually exclusive, while the cycle between P_1, P_2, P_5, and P_4 does not constitute a mutual exclusion set.

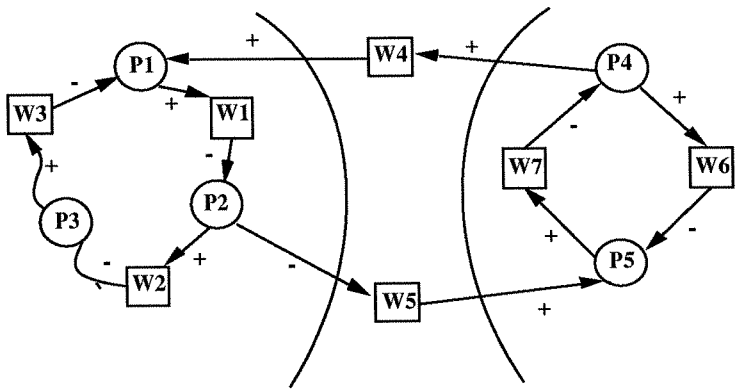

Figure 3.3: Example of Mutual Exclusions.

Theorem 1 *Parallel firing of all rules in a mutual exclusion set is not serializable without run-time checking.*[2]

Proof: Given a mutual exclusion set $\{P_1, \ldots, P_N\}$, we first prove by the case where N, the size of the mutual exclusion set, is 2. For N=2, assuming the two conflicting instantiations of interferences are ($P_1 \stackrel{-}{\leftarrow} W_1 \stackrel{+}{\leftarrow} P_2$) and ($P_2 \stackrel{-}{\leftarrow} W_2 \stackrel{+}{\leftarrow} P_1$), we prove that for all possible instantiations from the conflict set, without run-time checking, there exists no valid serial execution of $\{P_1, P_2\}$ such that P_1 proceeds P_2 and P_2 proceeds P_1, simultaneously.

Assume at time t=0 the conflict set is $CS_0 = CS_0^1 \cup CS_0^2$, where CS_0^1 and CS_0^2 are the subsets of conflict set originated from rule P_1 and P_2, respectively. To insure serializability without run-time checking, a parallel firing of any pair of instantiations, $\{I_1, I_2\}$, where $I_1 \in CS_0^1, I_2 \in CS_0^2$, should be serializable.

Let A_1 and A_2 be the RHSs of rule P_1 and P_2 respectively, and let conflict set CS_i represent the conflict set at cycle **i**. The effects of individual firings of I_1,I_2

[2]Run-time checking means the checking of interference/serializability at run-time. See Chapter 5 for more details.

on the conflict set are

$$\triangle cs^2 = A_1(I_1) \text{ and}$$

$$\triangle cs^1 = A_2(I_2)$$

where $\triangle cs^1$ and $\triangle cs^2$ are the updates to the conflict set from the firings of other rules. Since interferences exist between P_1 and P_2 in both directions, without run-time checking, interference ($P_1 \stackrel{-}{\leftarrow} W_1 \stackrel{+}{\leftarrow} P_2$) implies $\triangle cs^1 \neq \emptyset$ and ($P_2 \stackrel{-}{\leftarrow} W_2 \stackrel{+}{\leftarrow} P_1$) implies $\triangle cs^2 \neq \emptyset$. The new conflict set, after parallel firing of $\{I_1, I_2\}$, is

$$CS_1 = (CS_0^1 - \triangle cs^1) \cup (CS_0^2 - \triangle cs^2)$$

For serial firing of $\{I_1, I_2\}$, on the other hand, the changes to the conflict set are [3]

$$CS_0 \xrightarrow{I_1} [CS_1 = CS_0 - \triangle cs^2] \xrightarrow{I_2} [CS_2 = CS_1 - \triangle cs^{1x}]$$

Without run-time checking, it can be shown that $\triangle cs^2 \neq \emptyset$ because of the interferences from P_1 to P_2, thus a particular instantiation I_2 may be removed from CS_1 before I_2 even being selected for firing. The same analogy can be applied to the cases of $\{I_2, I_1\}$, thus concludes the proof of N=2.

In general case where the mutual exclusion set is $\{P_1, \ldots, P_N\}$ and the cycle in dependency is

$$P_1 \stackrel{-}{\leftarrow} W_1 \stackrel{+}{\leftarrow} P_2 \stackrel{-}{\leftarrow} W_2 \ldots \stackrel{+}{\leftarrow} P_N$$

[3] Notice the cycle count index.

the same analogy can be made such that any serialized execution of instantiations from all rules in $\{P_1, \ldots, P_N\}$ will invalidate some of these instantiations along the serial execution because of the cycle of interferences. □

Theorem 1 establishes the basis for parallel rule firings of production system programs. To correctly execute multiple rule firing, we first compute the mutual exclusion sets using Algorithm DP-ANALYSIS1, and then repeat the execution cycle where selections from the conflict set satisfy the mutual exclusion constraints, i.e., selection of multiple rules from the same mutual exclusion set should not form a cycle with conflicting interferences.

Theorem 2 *A correct CREL program is guaranteed to reach a correct terminal state under the execution scheme described above.*

Proof: From Theorem 1, we know all parallel rule firings of instantiations from the same mutual exclusion set, conforming to the mutual exclusion constraints, are serializable. For the cases of parallel firings of rules across multiple mutual exclusion sets, since there exists no cycle of interferences between rules from different mutual exclusion sets, any arbitrary interleaving of these rules is a valid serial execution. Thus, any parallel execution confirming the above execution scheme is serializable.

From Definition 2, any serial path in a correct CREL program is guaranteed to reach a correct terminal state, thus the serial execution corresponding to the parallel execution is also assured to reach a correct terminal state. □

3.3.3 Discussion

So far we have established the foundation of a parallel execution model for production systems. Before we proceed, some aspects of the current execution

model, such as trivial cycling and programming methods for constructing correct CREL programs, need to be discussed.

Trivial cycling refers to the cases where there are multiple firings of the same instantiation. In the OPS5, trivial cycling is avoided by always deleting the instantiation from the conflict set after its firing. In the CREL parallel rule firing, the same technique can be used without modification. Specifically, each rule contributes instantiations to the conflict set after its match phase. After the conflict resolution phase, multiple instantiations are selected for firing. All instantiations fired need to be deleted from the conflict set to avoid trivial cycling.

Notice that the changes of semantics we made to the OPS5 are confined to the conflict resolution strategies, therefore all the existing algorithms, such as the TREAT and RETE match algorithms, conflict set support, and no trivial cycling, still function properly in our model.

The changes in conflict resoluton strategies in the CREL, however, have greater implications on the programming methods. Generally, constructing a correct nondeterministic program is more difficult than a deterministic one. Not relying on *recency* is the first guideline in writing correct CREL programs. Techniques developed in UNITY[8] can also be adopted. In sections that follow, we will give two examples from existing the OPS5 programs to show that the amount of efforts involved to perform program conversions from the OPS5 to the CREL form depends heavily on the applications and the programming disciplines of the application programmers.

3.3.4 Global vs. Local Synchronization

If we look carefully at the the process of finding a valid serial execution path for a given parallel execution, we see that there exists no sequencing constraint

on rules that do not interfere with one another. To clearly explain the idea, we use the following example as an illustration. Given a production system program with five rules (as in Figure 3.3) where there are two mutual exclusion sets, $M_1 = \{P_1, P_2, P_3\}$ and $M_2 = \{P_4, P_5\}$. During each conflict resolution phase, assuming the conflict set always contains all entries from all five rules, the eligible selections are the combinations of the selections from M_1 and M_2, such as $\{P_1 \parallel P_2 \parallel P_4\}$ and $\{P_2 \parallel P_3 \parallel P_5\}$.

If one of the rules, P_2 for example, has a complex LHS match condition that takes up to 90% of the match time among all five rules in one cycle, the overall system utilization will be seriously degraded by the need for a global synchronization. On the other hand, since there is no sequencing constraint on rules from different mutual exclusion sets, a parallel execution path of $\{P_1 \parallel P_2 \parallel P_4\}$ followed by arbitrary number of firings of P_4, is still serializable, as though there were no synchronization requirement between the executions of M_1 and M_2. Such an observation leads to the following theorem that breaks the global synchronization requirement found in conventional production systems.

Theorem 3 *A parallel execution which observes mutual exclusion constraints as defined in Theorem 2 and has asynchronous execution cycles among mutual exclusion sets always reaches a correct terminal state if the given CREL program is correct.*

Proof: The correctness part can be derived from Definition 2, so we only need to prove that such parallel execution is always serializable. Without loss of generality[4], we assume the system contains two mutual exclusion sets, M_1 and M_2, as illustrated in Figure 3.4. The mutual exclusion constraints exist between the (P_i P_j) and (P_k

[4]This will also be true for cases of more than two mutual exclusion sets.

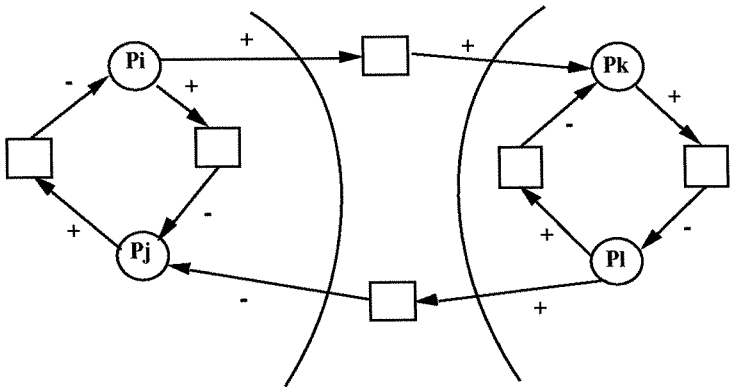

Figure 3.4: The example used in Theorem 2.3.

P_l) pairs. Any parallel execution without the global synchronization requirement between M_1 and M_2 can be expressed as a regular expression

$$E = (P_i \mid P_j)^* \parallel (P_k \mid P_l)^*$$

where the execution path, E, can contain arbitrary occurrences of one single instantiation from each set of M_1 and M_2. It can be shown that E is serializable since there is no constraint between rules from different mutual exclusion sets, therefore the firing frequencies between M_1 and M_2 have no effect on the serializability condition. □

Definition 4 Cluster

A cluster is defined as a set of rules where global synchronization is needed to insure serializability. □

From the proof of Theorem 3, it can be shown that a cluster is the transitive closure of mutual exclusion sets where synchronization is necessary for all rules in the closure. The transitivity of the cluster relation guarantees that the sets of rules resulting from clustering are disjoint, which is crucial to our run-time strategy.

The implications of Theorem 3 are that global synchronization among execution cycles is no longer required to insure the correctness of the execution model, and that the effect of large match-time variances on overall system utilization can be reduced to a minimum. Such observations can also help the scheduling/mapping problem by condensing the computation graph into clusters of mutual exclusion sets where synchronization boundaries meet with rule partition boundaries. The run-time management issues such as this, and other changes necessary for the asynchronous execution model, will be discussed in detail in Chapter 5 and 6.

3.4 Examples

Before proceeding to the next chapter on dependency graph transformations, we need to discuss the aspects of CREL programming due to the changes of its semantics. Although such changes are designed with the objective of keeping the OPS5 applications compatible with the CREL environments, removing *recency* and the introduction of nondeterminism in the CREL require additional work from the programmers. Since production systems do not support backtracking, one needs to insure that at any cycle, any instantiation selected for firing should always lead to a correct termination point.

The lack of flow control constructs in production system languages results in the programmers' tendency to reply on *recency*. The following two OPS5 examples are used to illustrate the opposite ends of the compatibility issue between the CREL and the OPS5. In general, we can use the invariants (as in UNITY programs) to prove the correctness of CREL programs. These examples only serve as demonstrations. A systematic proof mechanism for the correctness of CREL programs is beyond the scope of this research.

3.4.1 Example1:Tourney

Tourney[5] is an OPS5 program for tournament scheduling among 16 players(no.1 to no.16). The goal is to generate a schedule so that each player has the opportunity to play with each other player exactly once. **Tourney's** approach is first to generate all possible combinations (instead of permutations) of 4-tuples per game among all 16 players. Then use various test conditions to select, from all possible 4-tuples, the ones without conflicts with existing facts (such as no.1 already played with no.10). All the candidates and facts are recorded by ways of WMEs. The algorithm relies heavily on the *recency* strategy because of the " $\geq$" comparisons in various CEs.

To construct a CREL version of the **Tourney** program involves essentially rewriting the entire code. This is a typical example of the cases where *recency* is handy for writing efficient OPS5 programs. The reason is that any solution to the **Tourney** problem is actually a selection of 4-tuples among 16 players instead of a permutation of the 4-tuples. Thus during the stages of generating possible candidates, OPS5 **Tourney** uses ordering of the player numbers to insure only one set of the selections is generated instead of the full permutations of the same set of players[6]. Once the ordering is established in the temporary WMEs (class "foursome"), subsequent rules[7] take advantages of the *recency* strategy to test the conditions of the schedule accordingly.

From the viewpoint of state space search, using the *recency* strategy and enforcing total ordering of the player numbers in a candidate set restrict **Tourney**'s search paths to a small set of branches among the potentially large search trees. On

[5]See 4.3. Because of space constraints, actual code can be obtained from author upon request.
[6]Rules "north-pick-one" and "make-Candidate-make-Candidate"
[7]Rules "make-choice-do-it" and "make-candidate-make-candidate"

the other hand, a correct yet nondeterministic CREL program needs to search the entire problem state space to generate correct solutions.

3.4.2 Example2: Life

We now give a segment of an OPS5 program as an example opposite to the **Tourney** one. The **Life** OPS5 program[8] simulates the existence of bacteria in a rectangular grid of cells. After an initial pattern of living cells is input, the life/death of each cell is determined by the FACTS of LIFE, documented in the **Life** OPS5 code. To make our case short, the set of rules to compute the number of neighbors of a cell [9] is selected as an example to show that they do not rely on the *recency* strategy. By converting these three rules into UNITY-like program code and presenting the invariant, we can prove that they are indeed correct under the CREL semantics.

To present this example, we first begin with brief descriptions of the data structures used in the algorithm.

cell[5x5]: A 5x5(x,y range [1..5]) grid to hold the following information:

cell[i, j] = (↑ x ↑ y ↑ alive ↑ NB# ↑ gen# ↑ flag)
where x, y = grid coordinates,
alive = (alive, dead),
NB# = number of NBs,
gen# = generation number,
flag = being processed or not (1, 0).

NB[7x7]: A temporary array with index range [0..6] to count the number of neighbors.

The algorithm works by going through all cells alive and not already being processed. For all of those cells, rule "make-NB" generates 8 of the NBs adjacent

[8] See 4.3
[9] Rules "make-NB", "compute-NB", and "cleanup-NB"

to the cell being considered. After these NBs are generated, rule "compute-NB" goes through all WMEs of class "NB" and match the coordinates against the cell being considered. If there is a match, the NB WME is deleted and the NB# field in the current cell is incremented. The last rule in the set, "cleanup-NB", deletes all unnecessary NB WMEs to make ready for the next stage of problem solving.

To accommodate the inability of UNITY to dynamically allocate additional memory (i.e. creating WMEs), we modify some of the code by using variable counts to keep track of the NB WMEs created. This is one of the important areas that needs further work if we are going to develop a general proof methodology for the CREL because it is not always the case that the states of the WMEs can be represented by a simple counter.

We now gives a UNITY-like codes for the three OPS5 rules mentioned above, in Figure 3.5. The relations between the formulas given above and the original **Life** OPS5 code can be comprehended with close examination. It can also be shown that the following invariant always holds regardless of the execution sequence of the formulas P1,P2, and P3.

$$
\begin{aligned}
&\text{CONST}(x, y) = \\
&+ \textstyle\sum_{F(p,q)} (\text{cell}[p, q].\text{flag} \times (\text{cell}[p, q].\text{alive} = 1)) \\
&+ \text{cell}[x, y].\text{NB\#} \\
&+ \text{NB}[x, y]\,, \forall x, y \in [1..5]
\end{aligned}
$$

where $F(p,q) = \{(p,q) | \forall p, q \in [1..5], ((p,q) \neq (x,y)) \wedge (|(p,q) - (x,y)| \leq 1)\ \}$
and CONST(x, y) = Total number of neighbors of cell[x,y].

Thus concludes our discussion on the CREL semantics and issues in its various execution models.

P1: (corresponds to OPS5 rule "Make-NB")

$$
\begin{array}{lll}
\forall x, y & \in [1..5] & \\
& \text{IF} & (\text{cell}[x, y].\text{alive} = \text{alive}) \wedge \\
& & (\text{cell}[x, y].\text{flag} = 1) \\
& \Longrightarrow & \\
& & (\text{cell}[x, y].\text{flag} = 0) \\
& \| & (\text{NB}[x, y+1] = \text{NB}[x, y+1] + 1) \\
& \| & (\text{NB}[x, y-1] = \text{NB}[x, y-1] + 1) \\
& \| & (\text{NB}[x+1, y] = \text{NB}[x+1, y] + 1) \\
& \| & (\text{NB}[x+1, y+1] = \text{NB}[x+1, y+1] + 1) \\
& \| & (\text{NB}[x+1, y-1] = \text{NB}[x+1, y-1] + 1) \\
& \| & (\text{NB}[x-1, y] = \text{NB}[x-1, y] + 1) \\
& \| & (\text{NB}[x-1, y+1] = \text{NB}[x-1, y+1] + 1) \\
& \| & (\text{NB}[x-1, y-1] = \text{NB}[x-1, y-1] + 1)
\end{array}
$$

P2: (corresponds to OPS5 rule "calculate-NB")

$$
\begin{array}{lll}
\forall x, y & \in [1..5] & \\
& \text{IF} & (\text{NB}[x, y] > 0) \\
& \Longrightarrow & \\
& & (\text{NB}[x, y] = \text{NB}[x, y] - 1) \\
& \| & (\text{cell}[x, y].\text{NB\#} = \text{cell}[x, y].\text{NB\#} + 1)
\end{array}
$$

P3: (corresponds to OPS5 rule "cleanup-NB")

$$
\begin{array}{lll}
\forall x, y & \in [0..6] & \\
& \text{IF} & \text{TRUE} \\
& \Longrightarrow & \\
& & (\text{NB}[x, 0] = 0) \\
& \| & (\text{NB}[0, y] = 0) \\
& \| & (\text{NB}[x, 6] = 0) \\
& \| & (\text{NB}[6, y] = 0)
\end{array}
$$

Figure 3.5: Unity-like code for the Tourney program.

Chapter 4

Dependency Graph Analysis and Transformations

Given the definitions of CREL and its parallel execution model, the next step is to extract as precisely as possible the run-time characteristics of production system programs to perform effective compilation and task scheduling. The focus of this chapter is on the compile-time dependency analysis and on the transformations of the CREL computation graphs.

There are many works on the static analysis of computation graphs for conventional languages for various parallel environments[27, 17, 35]. Although the basic methods of static analysis are the same, the problem of mapping computation graphs of production system programs onto resource graphs differs from that of the conventional languages. The lack of language constructs, both in terms of control and data constructs in production system languages, makes our static analysis difficult. Moreover, pattern matching operators with free variables and arbitrary expressions in the LHS CEs in CREL makes the static analysis intractable. In addition to the static dependencies among rules, production system run-time behaviors rely heavily on the states of the WMEs. The results presented later in this chapter further support the above statement.

Facing difficulties in the static analysis, we develop two different approaches to increase the degree of parallelism as measured by the compile-time analysis. The first approach takes advantages of some common practices in production system programming, such as embedding control flow in the data flow model,

and knowledge of the relationships between CEs with free variables in order to restrict interferences between rules. The effects of these transformation techniques is to reduce the densities of the dependency graphs, which implies more parallelism. The second approach is to increase rule level parallelism by optimizing rules with complex LHSs. Techniques commonly used in parallelizing block structure languages, such as loop unfolding, and techniques used in database query optimization, such as horizontal and vertical partitioning, are examined and tailored to fit in our context. Again, the effects of such transformations are to further reduce graph densities and consequently increase parallelism.

The goal of this chapter is to have an optimized computation graph from static analysis with sufficient degrees of parallelism. Further explorations of run-time parallelism will complement the static analysis.

4.1 Dependency Analysis

4.1.1 Control Dependency

For a production system language lacking control structures such as CREL, mutual exclusion relationships do little to reveal the true dependency constraints in a computation graph. Because production systems are programmed such that WME classes represent real world objects, and rules are designed to test various states of the objects, the dependency graphs from mutual exclusion analysis are most likely to be strongly connected clusters. Run-time mutual exclusion analysis is applicable, but since the average cycle time of a typical production system program being small[25], we would like to reduce the complexity of the run-time analysis by reducing the sizes of the rule sets by the clustering technique.

In writing production system programs, a common strategy called "secret-message" [55] is used to emulate the block structures in conventional languages.

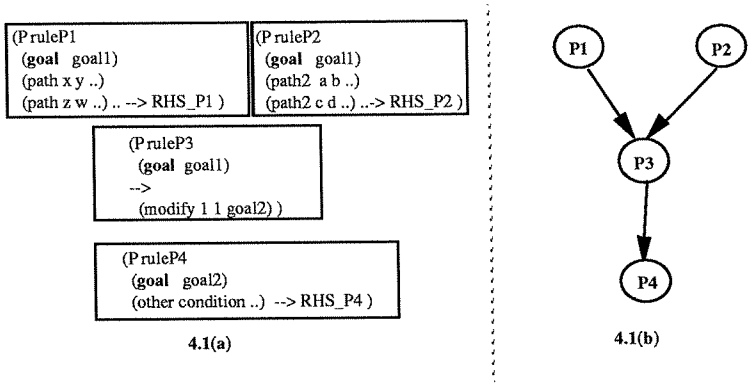

Figure 4.1: Control variable and the precedence relations.

Such a strategy uses a designated class (usually called **goal** elements) to constrain the range of active rules to a certain set so that different stages of problem solving are partitioned into different set of rules. The rules responsible for transitions between different stages make use of the "specificity" resolution strategy to impose priorities on rules. The example in Figure 4.1.a illustrates the concept.

In Figure 4.1, the control variable **goal** is used to limit the set of active rules in the current "method"[7] to rules P_1 and P_2. Such a strategy is beneficial both in terms of modular programming and our dependency analysis. Given a production system program, one can then construct a partial ordering on the sequences of firings among rules in a production system program by tracing through the values of the control variable such as **goal**, as in Figure 4.1.b.

Note that, although such partial ordering from the control variable(s) may have cycles due to the repetition of various stages in the problem solving, only the part of the information that talks about "rules that can never be fired at the same time" are useful in the compile-time analysis.

Define the precedence relation, " $\Rightarrow$", between rules as $P_i \Rightarrow P_j$ if, within a cycle, the firings of P_i always precede the firings of P_j due to "specificity". We say two rules P_i and P_j can be active at the same time if and only if there is no precedence

constraints between them. Rules P_1 and P_2 in Figure 4.1 can be active at the same time.

Notice that the relation " $\Rightarrow$" only imposes a partial ordering on rules. We further define a set of rules as simultaneously active to each other if and only if there exists no precedence constraints among any pair of rules in the set.

We can think of the mutual exclusion and control dependencies as a set of orthogonal relationships among rules in a CREL program graph. When both types of relationships are analyzed jointly , the resulting graph will be more realistic in representing the run-time behavior of the system. With this in mind, we give revised definitions of mutual exclusion and clusters in the next subsection.

Because the use and intent of control variables is governed strictly by the desire of programmers, and the the bounds and sequences of such control variables can not be estimated precisely during compile-time, either inputs from sample trace files or assistance from the programmer at compile-time should be provided. In our current analysis, the **goal** element is provided by the programmer. Eventually we hope to somewhat automate such goal element selection processes.

4.1.2 Mutual Exclusion Revisited

The purpose of defining mutual exclusion and clustering is to identify, at compile-time, the smallest set of rules which are mutually exclusive and thus need to be fully synchronized during the execution cycle. The basic idea again is serializability. Our previous definitions of mutual exclusion and cluster assume all rules are active simultaneously, which may not be true when the control dependency is introduced. Therefore, we revise these definitions as follows:(called "control variable smart")

- Mutual Exclusion

  A set of rules are mutually exclusive if and only if they are all active and there exists a minimum cycle of interferences in the bipartite dependency graph.

- Cluster

  A cluster is composed entirely of a set of rules that are mutually exclusive.

Our analysis on the existing benchmark programs in [25] shows that, by combining control dependency and mutual exclusion dependency, the technique of "control variable smart' can improve significantly the quality of the dependency analysis. Table 4.1 gives a comparison of the analysis with and without the use of control variables.

The modification needed to include "control variable smart" to calculate both the mutual exclusion sets and clusters, is straightforward, as given below:

Algorithm 2 DP-ANALYSIS2

Given a bipartite data dependency graph G_m *of a production system defined the same way as in Algorithm 1, delete all edges between two rules* P_i, P_j *where* P_i *and* P_j *cannot be active at the same time. Apply Algorithm 1 on the resulting bipartite dependency graph.* □

It can easily be shown that the parallel asynchronous execution model proposed in Chapter 3 is still correct if the new definitions are used since the basis of "interference" between two rules P_i and P_j is such that the firing of one rule "may" violate the conflict set of the other rule. If P_i and P_j cannot be active at any time, such dependency could not exist.

4.1.3 Data Dependency

Data dependency is defined to be the relationship where firing of a rule P_i affects the match conditions of some other rule P_j. In our approach such relationship is embedded in mutual exclusion analysis. Portion of the data dependency is also used for program flow control, which is used to analyze the control flow. Lastly, the amount of data interaction can be associated with edges in the computation graphs for task assignment.

4.2 Dependency Graph Transformations

Even with the introduction of control variables, results of experiments on existing benchmark production system programs provide only moderate encouragement in terms of the degree of parallelism. After careful study of these benchmark systems, the following facts are observed. Forward chaining production systems differ from other expert systems paradigms regarding the state space search techniques. Production systems use subgoals to link segments of search paths into a global path leading to the final solutions. To avoid backtracking in a production system, intermediate potential search paths are recorded by creations of temporary WMEs. A specific set of rules then use various condition tests in their LHSs to insure correct search paths are selected.

The implications of such techniques for the behavior of production system executions are that there will be many potential goals or potential subgoals generated in the form of temporary WMEs and that there will also be rules with complicated CEs to select correct solutions among those candidates. For the rules responsible for generating potential-goals WMEs, the total count of firings per execution will be large, with medium cycle time. However, for rules responsible for goal screening, the cycle time will be large while the total number of firings will be small due to the

complexity of the RHSs and the large numbers of WME candidates. We can think of such strategies as expanding and contracting subtrees in state space searches. We name the former type of rules "expanding rules" and the latter ones "contracting rules".

As mentioned in Chapter 1, in order to achieve substantial speedup in production system execution, parallel rule firings must be adopted. Since evaluating LHS match still takes up a major part of the computation costs, one should constantly be conscious of the effects on match performance in designing parallel firing algorithms.

Ideally, during the execution, the system should be smart enough to identify these two types of rules and adopt different execution strategies. For expanding rules, instantiations should be fired in parallel and independently whenever possible, while for contracting rules, efforts should be made to reduce the match time as well as the frequency of evaluating LHS match.

Although currently these two types of rules cannot be identified from our bipartite data dependency graphs, with some assistance from programmers the following decomposition technique generally can improve the degree of parallelism and optimize LHS match at the same time. Before we continue with the graph transformations, it's important to identify the equivalence between production system LHS evaluations and query processing in relational databases.

4.2.1 Representing CREL LHS by Relational Algebra

The equivalence between LHS evaluation and relational query processing can be shown as follows:

1. Classes are relations, and WMEs are tuples.

2. A CE in a LHS is a *select* query.

3. Multiple CEs in a LHS are equivalent to a *join* operation over multiple relations. A LHS in a rule

```
(P P1
   (A .. <x> .. )
   (B .. <y> .. ) --> RHS_P1 )
```

   is equivalent to a *join* query on relations A and B, $\left(A \bowtie_{f(x,y)} B\right)$, where $f(x, y)$ is the function to bind variables in A and B.

4. Multiple CEs in a LHS with negative CEs are equivalent to a join followed by a set difference operation. A LHS in a rule

```
(P P2
   (A .. <x> ..)
  -(B ... <y> <x> ..) --> RHS_P2 )
```

   is equivalent to $\left(A - \prod_A (A \bowtie_{f(x,y)} B)\right)$, where $\bowtie$ and $\prod$ are *join* and *projection* operators respectively. The above query reads: "find all instantiations of A which satisfies conditions in A while there exists no B instantiation such that $f(x, y)$ is true."

Once the equivalence has been established, techniques in query optimization can be adopted to optimize the matching task in production system execution. Two basic query optimization techniques are:[67, 45]

1. Horizontal Partition

   Partition relations in a query into disjoint subsets and evaluate the partitioned queries independently. The union of all results from these partitioned queries becomes the result of the original query. That is, if A can be partitioned into disjoint subsets $\{A_i, i = [1..N]\}$, then

$$(A \bowtie_{f(x,y)} B) = \cup_{\forall i}(A_i \bowtie_{f(x,y)} B)$$

2. Vertical Partition

   By join operator's associativity, a complex query can be decomposed into sub-queries where the results are again joined into the final result, such as the following

$$A \bowtie_{f1} B \bowtie_{f2} C \bowtie_{f3} D = (A \bowtie_{f1} B) \bowtie_{f2} (C \bowtie_{f3} D)$$

Stolfo first applied the technique of horizontal partition to the production systems[54], with the intention to reduce match time of "hot spot" rules with complicated RHSs. The same technique can be applied to the dependency graph as a form of graph transformation. The results are surprisingly good in reducing the LHS match complexity and reducing dependency graph connectivity, while the implementation overhead can be kept to a minimum. In order to store intermediate join results, however, the vertical partition technique introduces additional declarations of classes and requires multiple execution cycles to compute the original match computations. Such overhead cannot be justified without statistics from a real parallel implementation, so this approach will be addressed in the later stage of this research.

4.2.2 Propagating Constants

One of the improvements one can make to the dependency graph is to provide better classification of the data units used in the bipartite dependency graph. Reducing granularity of the objects implies reducing the density of the graph. Since all "squares"(□) in the graph represent some forms of selections from WM classes, certain assertions known at compile-time can be used to prevent interference between two rules. One approach is to take into consideration all constants in the LHSs. I.e., CEs with the same class name but with an attribute of different constant bindings

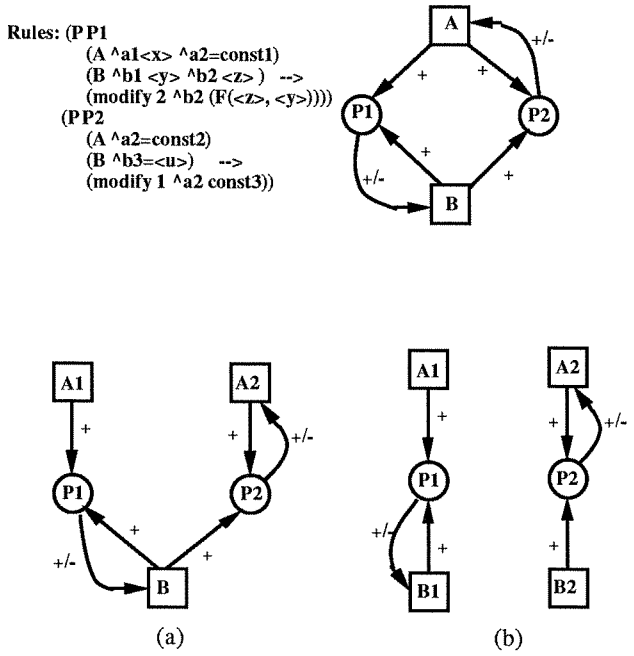

Figure 4.2: Examples of refinement by constant (a) and detecting disjoint attribute sets (b).

should be treated as different "□" nodes. The top part of the dependency graphs in Figure 4.2 gives an example with the entire "class" as the unit of objects. Figure 4.2.a shows the improvement by propagating constants in attribute a2 of class A.

Another improvement one can make is to identify cases, where CEs with the same classes but with disjoint references of attributes, should be treated as different nodes in the dependency graph. Figure 4.2.b illustrates the result of applying such a technique on the original bipartite data dependency graph to further reduce graph connectivity.

4.2.3 Horizontal Partitioning With Constrained Copies

The constrained copies technique shares the basic idea with the "tuple substitution" technique used in Ingres query optimization algorithms[67] by instan-

tiating potential variables into disjoint hash buckets, thus replicating the query into independent sub-queries. As an example consider a rule P_i with two CEs:

```
(P Pi
   (classA âtr1 <x>     âtr2 K1 )
   (classB âtr3  K2     âtr4 <x>)
   -->
   (modify 1 âtr1= (compute <x> + <y> ) ))
```

Assume $|x| = \{x_1, x_2, \ldots, x_m\}$, $x_i \cap x_j = \emptyset$ and K1, K2 are constants [1]. We can copy rule P_i into m rules, $P_{11}, P_{12}, \ldots, P_{1m}$ with variable x bound to x_i in P_i. The resulting set of rules will have exactly equivalent effect with the original rule P.

It can easily be shown that performing constrained copying on rules in a production system will not decrease the degree of parallelism from the mutual exclusion analysis. Constrained copying of any free variable will optimize the LHS match time as in query optimization, and there are many variable selection algorithms in query optimization[67, 59]. To increase parallelism in the dependency graph, however, not only the selection of variable for constrained copying is important, but the same attribute copying should be propagated throughout all rules in a cluster. The examples in Figure 4.3 illustrate cases where some variables are not suitable for constrained copying.

Generally speaking, it is difficult to select the best attribute (variable) to perform constrained copying at compile-time, since there's no estimate of the bounds of all attributes with free variables, and the number of expanded rules depends heavily on the loads of the processors at run-time.

Figure 4.4 gives the resulting graph from performing constrained copying and propagation(CCP) on part of **Waltz** program.

[1]Notice that the partition of x can be comprised of hash values instead of constants, as long as the partitions are disjoint.

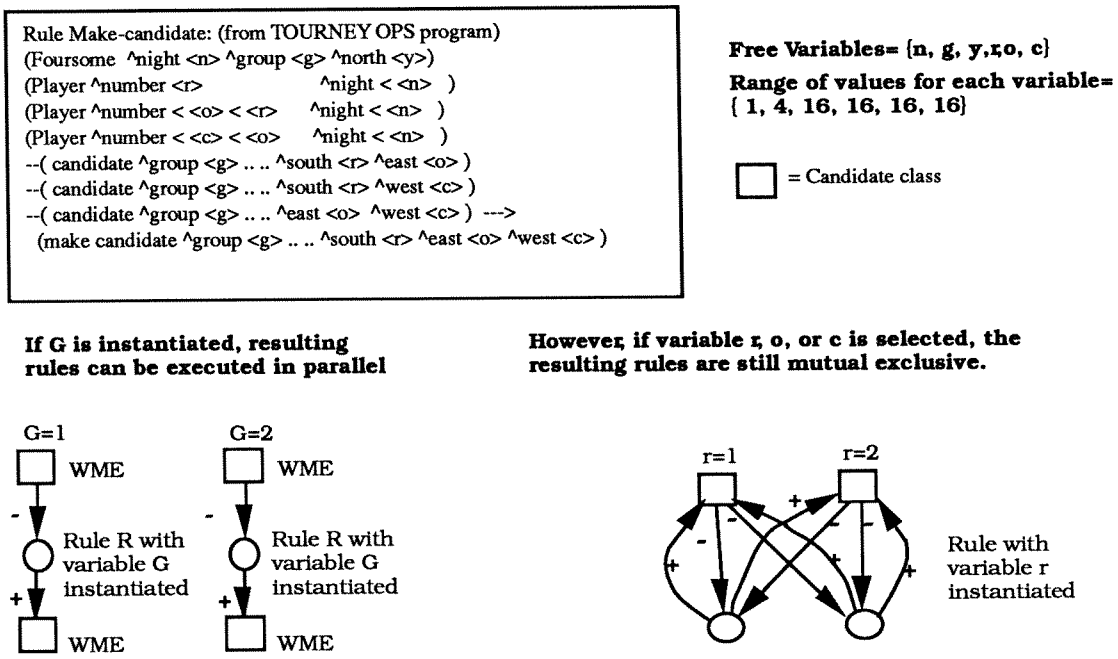

Figure 4.3: Examples of the importance of attribute selections.

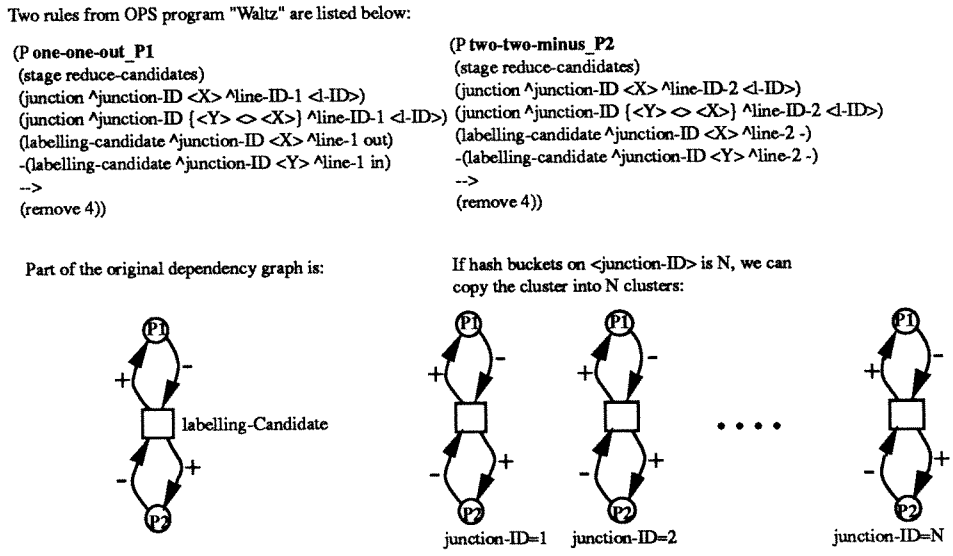

Figure 4.4: CCP algorithm and its example.

4.2.4 Multiple Rule Firing–Loop Unfolding

An important issue in compiling parallel production system programs is to unfold the "expanding" rules, i.e., to be able to fire multiple instantiations of the same rule in parallel. The example below illustrates a typical loop construct in OPS5 programs where all instantiations from rule P1 should be fired in parallel instead of looping through the match, select, and act cycles sequentially.

```
(P P1                              (P P2
 (goal remove-all-A)                 (goal remove-all-A-done
 (classA âtr1 <x> âtr2 K1)         -(classA âtr1 <x> âtr2 K1)
-->                                 -->
 (remove 2) )                        (modify 1 next-stage) )
```

By combining the idea of constrained copy and mutual exclusion analysis, the following algorithm is designed to detect the cases where we can "unfold" a loop, as in the above case. To fire multiple instantiations from the same rule in parallel, situations of interferences must be identified.

Algorithm 3 *Given a rule* P_i *of the form*

$$LHS_i(x1, x2, \ldots) \longrightarrow RHS_i(x1, x2, \ldots)$$

Let the conflict set of P_i *be* $CS_i(x1, x2, \ldots)$ *and* $\forall I \in CS_i$, $RHS(I)$ *be the RHS action of instantiation* I. *Rule* P_i *can have parallel firings of multiple instantiations if and only if* $\forall I \in CS_i$, $RHS(I)$ *does not invalidate any instantiations in* $(CS_i - I)$.

Proof: It can be shown that if there exists an I_m such that $RHS(I_m)$ invalidates instantiations in CS_i other than I_m, say I_n, the same interference exists from I_n to I_m, thus parallel firing of I_m and I_n is not serializable. □

Examples of loop bodies below illustrate various cases of whether looping rules can have all their instantiations fired in parallel. Rule P1 is a simple case where

parallel firings of multiple instantiations of P1 is serializable since the action of each instantiation will not invalidate other instantiations in the conflict set of P1. Rule P2 has only "make" as the RHS, so it is a perfect target for parallel firing of multiple instantiations. Rule P3, however, cannot have its loop unfolded since without run-time checking, the action of "modify"ing B by one instantiation may invalidate other instantiations in the conflict set.

```
(P P1                          (P P2
   (A âtr1 <x> ..)               (A âtr1 <x> ..)
   -->                            (B âtr2 {f2(<x>,<y>)})
   (modify A f1(<x>)))           -->
                                  (make A âtr1 f3(<x>,<y>)))

(P P3
   (A âtr1 <x> ..)
   (B âtr2 {f4(<x>,<y>)})
   -->
   (modify 2 f5(<x>,<y>)))
```

Notice the similarity between loop structures in production systems and conventional languages. Various techniques in unfolding Fortran loops[52, 1], such as "variable renaming" and "variable expanding" can be applied to production systems with modifications. This will also be part of our future work.

4.3 Static Analysis Results

Existing OPS5 benchmark programs studied in this section are

1. **JIG25:** Board game puzzle, total 5 rules.
2. **FACT:** Compute Factorial, total 6 rules.
3. **LIFE:** Simulate the birth/death life cycle of bacteria under certain rules by Conrad, total 14 rules.

4. **TOURNEY:** a Tournament scheduling program, 17 rules.

5. **TORUWALTZ:** Ishida's version of Waltz algorithm, total 32 rules.

6. **WALTZ:** The original version of Waltz algorithm, total 34 rules.

7. **RUBIK:** Rubik's game, total 66 rules.

8. **MAPPER:** A routing program to aid visitors traveling in New york City, total 235 rules.

9. **WEAVER:** A VLSI routing program, total 700 rules.

10. **JUDGE:** A Payload Communication program converted from CLIPS [20], total 245 rules.

These benchmark programs were studied to determine whether they are recency-free. So far we have convert FACT, LIFE, TORUWALTZ, WALTZ, as well as JUDGE into recency-free CREL forms. Empirical data shows that the conversion from recency-dependent OPS5 programs to recency-free CREL forms will introduce few additional dependencies among rules, thus statistics in Table 4.1 in general do reflect the static analysis of these benchmarks in recency-free forms.

4.3.1 Static Results

The current implementation takes into account all optimization techniques mentioned in this chapter, including constant propagation and "control variable smart". Constrained copying with propagation is selectively applied to LIFE and TORUWALTZ, represented as the last three entries in Table 4.1.

The result of the static clustering technique is represented by the total number of clusters in the system, and the maximum number of rules in a single cluster,

Compile–time Dependency Analysis					
Pgm No.	# of rules[2]	W/O Control Dep.		With Control Dep.	
		# of cls	max. # of rls/cl	# of cls	max. # of rls/cl
1	5	1	5	†[3]	–
2	5	5	1	†[4]	–
3	9	1	9	6	4
4	16	11	5	13	4
5	27	4	24	4	24
6	32	5	28	11	18
7	66	11	54	18	13
8	235	†[5]	–	–	–
9	700	†[6]	–	–	–
10	245	24	31	43	28
life-4	18[7]	3	16	14	2
toru3	63[8]	7	15	7	15
toru4	99[9]	7	24	7	24

Footnotes in Table 4.1 are:

4. Rules responsible for initialization of WM and I/O are not included in the analysis.
5. No Control Variable.
6. No Control Variable.
7. Currently Some external lisp functions are not implemented.
8. Currently special function **substr** in OPS5 is not implemented.
9. CCP with hash buckets of 4.
10. CCP with hash buckets of 4.
11. CCP with hash buckets of 4.

Table 4.1: Preliminary Static Dependency Analysis Results

to reflect the density of the dependency graph. In Table 4.1, the first column lists the benchmark program number defined earlier. The last three entries are special cases where techniques of CCP were applied to LIFE (life-4) and TORUWALTZ (toru3 and toru4). The second column lists the total number of rules in the system. The 3rd and 4th columns list the total number of clusters and the maximum cluster size, respectively. They are the dependency analysis results without tracing control variable. The 5th and 6th columns, on the other hand, represent cases where control variable is taken into consideration. Without control variable, for instance, the dependency analysis on RUBIK generates 11 clusters, with the largest cluster containing 54 rules. With control variable, the same RUBIK program can be partitioned into 18 clusters, with the largest cluster being only 13 rules.

Note that we did not present these preliminary results in terms of "speedup" or "degree of parallelism" because for one thing the computation costs of each rule are not included in the static analysis. Secondly, static analysis shows little information of the run-time activities of a system. What Table 4.1 does show is that, by combining various techniques discussed in this chapter, we can structure a ordinary CREL program into a more balanced one for efficient and asynchronous execution.

The static clustering analysis provides no gain of parallelism if no optimization is applied. This is due to the heavy usage of pattern matching variables in OPS5 programs as well as a programming style advocated by OPS5 primers that limits the number of different classes and defines many attributes per class. Since static analysis can not instantiate these free variables, the initial clustering result without optimization always contains a heavily connected cluster.

CCP always provides good results, provided there are eligible free variables to partition. Again this can be attributed to the heavy usage of free variables

in these systems. The performance of any static analysis technique depends heavily on the program and the dynamic results can only be determined at run-time. The RUBIK program, for example, has a very tightly coupled dependency graph. The RUBIK code is essentially implementing of depth-first search. Although the static numbers in Table 4.1 shows relative good results after the optimization, the trace run shows there are at most 2 clusters active concurrently and very few instantiations exist in the conflict set of each cluster at any given time.

4.3.2 Dynamic Results

Results from the static analysis do not reflect the number of clusters that are concurrently active at any given time. To observe the run-time activities, a CREL execution prototype system was implemented on the shared-memory Sequent Symmetry. This system provides minimal support for the run-time management with each cluster being statically assigned to a physical processor, and no multiple rule firing per cycle. The results give task-level parallelism as determined by the static clustering techniques.

Three benchmark programs that have been executed by the CREL system are the Life and two versions of the Toruwaltz programs. Concurrency profiles were plotted to study the run-time characteristics of these systems. The concurrency profiles were measured in terms of actual rule firings from each cluster. Figure 4.5 to 4.8 give the snapshots of the number of clusters active at any instant for the the three systems. The X-coordinate is the execution time of the system in milliseconds. The Y-coordinate indicates which active cluster is firing rules. Figure 4.5 represents the concurrency profile of LIFE with no optimizations. Not counting I/O, LIFE operates in two sequential phases, an enumerate stage and a data-elimination stage. Without optimization, the enumeration phase is represented by a single rule and is purely se-

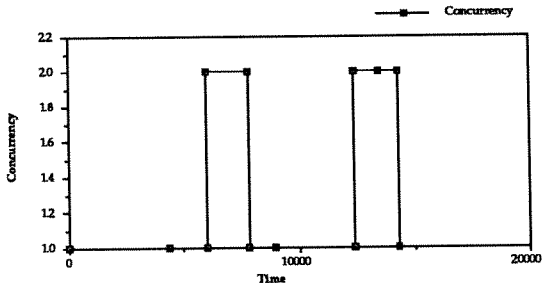

Figure 4.5: Concurrency profile of Life.

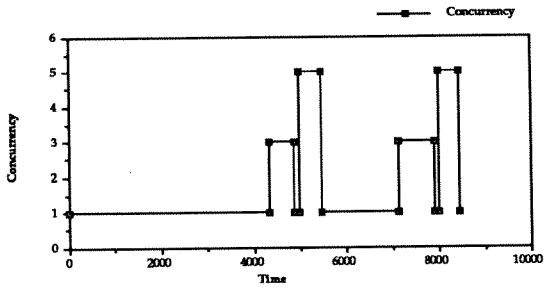

Figure 4.6: Concurrency profile of Life with optimizations.

quential. In the data-elimination stage, there can be up to 5 clusters active at the same time. Figure 4.6 is for Life with optimizations 1,2 and 3. Figure 4.7 is the concurrency profile, with cluster trace superimposed, of Toruwaltz with optimizations 1, 2, and 3. Toruwaltz also operates in two sequential stages, candidate enumeration and other candidate elimination. However, the second stage of Toruwaltz is connected into one big cluster which resulting in the strictly sequential execution. Figure 4.8 is the Toruwaltz system with all three optimizations plus applying CCP of size 4 on the cluster responsible for the second stage. We can observe the both the increased parallelism and the reduction of match time.

4.4 Conclusions

By identifying characteristics of production systems and applying decomposition techniques used in database query optimizations, we show the potential

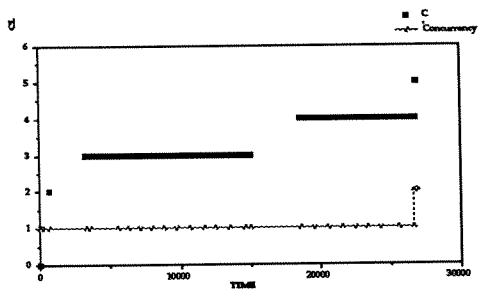

Figure 4.7: Concurrency profile of Toruwaltz.

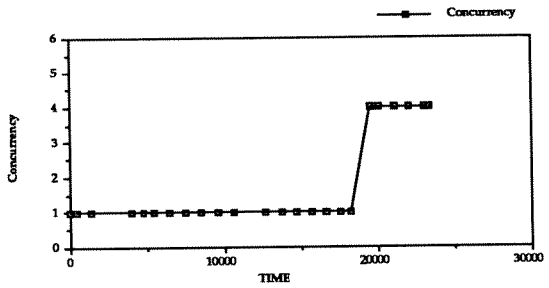

Figure 4.8: Concurrency profile of ToruWaltz with CCP of Size 4.

rule level parallelism in production systems by performing transformations on the dependency graphs. The merits of our decomposition technique are its simplicity, and that one can perform these decompositions at compile time. A well-balanced CREL computation graph can reduce the complexity of the run-time checking and management.

We also mention other compile-time techniques to reduce the interferences among rules. In all the techniques mentioned in this chapter, the restrictions of "compile-time" make analysis difficult, due to unbounded variables. Later we shall explore possibilities of run-time checking, assumptions on the ranges of attributes, and other means to predict run-time interferences more accurately.

Chapter 5

CREL Run-Time Parallelism

In Chapter 4, we discussed various aspects of the compile-time analysis of dependency relations and suggested transformations to improve parallelism exhibited in the dependency graphs. We also observed that run-time load balancing is necessary for the cases where static clustering fails to recognize the available parallelism. This chapter analyzes sources of run-time parallelism from a cluster's viewpoint. We first study the run-time characteristics of CREL systems. We then identify the criteria of an efficient CREL run-time system. Two potential sources of run-time parallelism are identified: one is to exploit parallel match, the other is to allow multiple rule firings per execution cycle. These two issues are addressed separately in the sections that follow. The conclusion section summarizes the run-time behaviors of CREL systems and leads to a summary of the global picture of the CREL run-time process management, which is the topic of the next chapter.

5.1 Characteristics of The CREL Run-Time System

When exploring the CREL run-time parallelism, implementation details pertaining to the CREL system need to be taken into considerations to avoid overhead and redundant work. To identify the bottlenecks in the CREL execution and to parallelize them, the basic code blocks as well as the key data structures of a

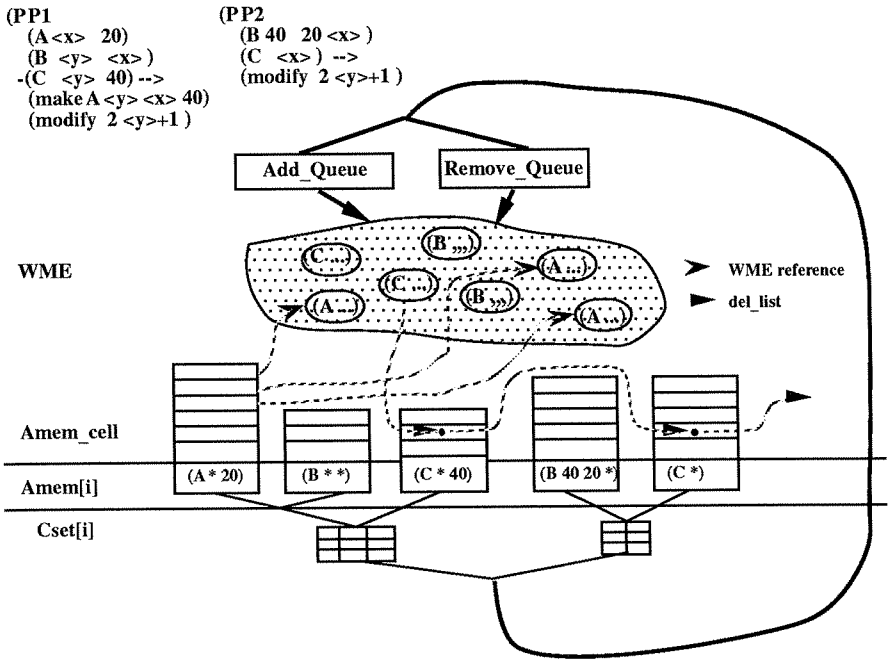

Figure 5.1: CREL key data structures.

sequential CREL execution cycle are reiterated here [1].

5.1.1 Data Structures

Three key data structures in a sequential CREL system (based on the TREAT algorithm) are:

[**WME**] WMEs are the core of a CREL system; they are dynamically allocated and globally shared to avoid the need of maintaining one copy for each cluster. A WME is a structure that contains the attribute values, and a linked list of back pointers (**del_list**). Each back pointer points to the alpha memory entries that reference it.

[1]Because of the CREL asynchronous execution model, we assume the context of discussion is within a cluster, unless stated otherwise.

[alpha memory list][2] Alpha memory list is a 2-dimensional linked list of pointers to WMEs and pointers to other alpha memory entries. The first list forms an alpha memory list for each CE in a rule. The second list is used to chain all the alpha memory entries that reference the same WME. The purpose of maintaining such a chain is for efficient delete operations on alpha memory entries when the referenced WME is deleted.[46].

[CSet] Each rule, i, has a conflict set, CSet[i]. CSet[i] is used to store the instantiations that successfully passed through the match network. An instantiation contains the addresses of the WMEs that constitute the instantiation.

[Remove_Queue], an array to hold the WME addresses(IDs) being marked as removed by the RHS actions in the current execution cycle. These IDs are kept in the Remove_Queue[] so that the actual WME removals can be delayed to the end of the current execution cycle.

[Exec_Queue] Exec_Queue[] is an array that holds the newly created WME IDs by the RHS actions in the current execution cycles. These IDs are kept in the Exec_Queue[] so that the match work of each new WME can be initiated at the beginning of the next execution cycle.

Additional data structures will be explained when we discuss the actual code blocks. With the key data structures explained above, we then proceed to discuss the main execution loop of a CREL system. Figure 5.1 illustrates a picture of these data structures and the relationships between them.

---

[2] Also known as AMem[] trees.

5.1.2 Main Loop

Under the CREL asynchronous execution model where clusters are the units of synchronization, each cluster behaves essentially the same as a sequential OPS5 system. The main loop in each cluster is given in Figure 5.2.

```
Match:{
        for each WME in Remove_Queue[] do {
          for each AMem_Entry in WME's del_list do {
                  update AMem[] link;
                  if AMem[] belongs to a −CE,
                    then join(WME, AMem_Entry);
                  free(AMem_Entry);
          }
          for i=0 to N do {
                  search WME in Cset[i], delete the entry if found;                    10
          }
        }                                   /* process Remove_Queue */
        for each WME in Exec_Queue[] do {
          for each AMem[i] involved with WME do { /* compiler generated */
                  allocate(AMem_Entry);
                  update AMem[i] link with AMem_Entry;
                  if (rule_active = TRUE)
                    then join(WME, AMem_Entry);
          }
        } }                 /* Match */                                              20
Select:{
        Firable = Conflict_resolve();
}                   /* Select */
Act:{
        for each rhs_action in RHS of Firable do {
          call rhs_action(); /* rhs_actions will update Remove_Queue[] and Exec_Queue[] */
}                   /* Act */
```

Figure 5.2: CREL Main_Loop

Code Main_Loop can be viewed as a cyclic data-flow program, where the Remove_Queue[] and the Exec_Queue[] are the reservoirs of the tokens that initiate the join.[3] The alpha memory lists and the join codes are the data-flow nodes and the

[3]Join and match will be used interchangeably.

corresponding processing routines, respectively. Tokens successful passed through the match network are stored in the conflict set for conflict resolution. After the "firable" is selected from all the Cset[], rhs_actions are executed and tokens are generated to reflect the updates to the WM space.

5.1.3 Criteria for Exploring Run-Time Parallelism

There are many potential sources of parallelism in the Main_Loop. The semantics of CREL system, however, requires rigid synchronization between the match, select, and act phases; one must be careful in exploring the sources of run-time parallelism so that excessive synchronization overhead can be avoided. Locking on globally shared data is a common source of overhead associated with parallelization of sequential programs. The criteria of an efficient CREL run-time system are recognized as:

1. Avoid redundant work in the parallel implementation. This is crucial to the parallel match. There is always a tradeoff between the extent one can pursue parallelizing match and introducing redundant effects.

2. Keep the total number of locks small. This is important for the CREL systems because of the complex data structures and, among others, the potentially large numbers of WMEs and alpha memory entries.

3. Keep the synchronization overhead small. This is important because of the highly dynamic behaviors and small cycle time of CREL systems.

4. Avoid fine-grained parallelism, for the same reasons stated in (3).

5. Design modular code for efficient task allocation. To reduce the overhead associated with task allocation and migration, CREL's task decomposition and

CREL Run-Time Profiles					
Waltz40		Waltz20		Tour	
Procs	%time	Procs	%time	Procs	%time
join_5	17.3	join_9	18.4	mcount	22.5
join_9	17.1	join_5	16.0	match_inst	8.8
find_inst	13.2	find_inst	12.8	write	7.9
join_12	11.4	join_12	11.1	join_44	6.3
join_38	3.9	join_38	3.7	join_58	5.8
FindNode	2.9	FindNode	2.8	join_42	5.2
join_65	2.6	join_65	2.5	join_43	4.5

Table 5.1: CREL Run-Time Profiles

code generation should be designed in such a way that the cost of loading both the code and data segments by a physical processor is kept small.

Table 5.1 illustrates run-time profiles of some CREL benchmarks. As one can observe from these two figures, the potential sources of parallelism to explore at run-time is the match and conflict resolution stages. When considering the above criteria, we choose to incorporate the token, intra-token, and conflict resolution parallelism in a CREL system. The next three sections discuss the each source of parallelism in detail.

5.2 Token Parallelism

The loops in Figure 5.2 that process the WME updates from both the Remove_Queue[] and the Exec_Queue[] are potential candidates for parallelism. Within the loops, each WME entry is examined for constant matching and appropriate join code is invoked if the matching is successful. Such parallelism can be explored by loop unfolding techniques commonly seen in numerical applications. Although we stated earlier that the match percentage in a typical rule-based system decreased

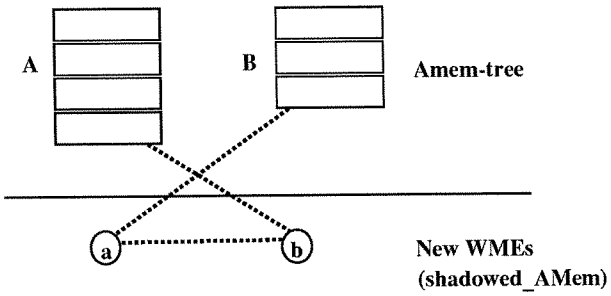

Figure 5.3: An example of token parallelism.

to be less than 50%, match work still plays a significant role in the entire execution.

The idea of token parallelism is to explore the WME token level. "Token parallelism" is so called because of the resemblance between dataflow programs and the loops in Figure 5.2, where WME tokens pass through alpha memory lists to invoke the corresponding join.

The joins associated with the newly created WME tokens are relatively independent. The simple example in Figure 5.3 illustrates the case at the beginning of a loop execution, where A, B represents the old alpha memory lists, and a, b are the newly created WMEs,respectively. If the loop that processes a and b is expanded, then the program will first invoke a join of $(a \bowtie B)$, followed by a update to alpha memory list, A, the second join, $(A + a) \bowtie b$, followed by a update to alpha memory list B. Formally, the work involved are in sequence given as follows(5.1):

$$(a \bowtie B),\ (A' = A + a),\ (b \bowtie A'),\ (B' = B + b) \tag{5.1}$$

Notice that in the above sequential joins, $(a \bowtie B)$ and $(b \bowtie A')$, are interleaved with update to A, $(A' = A + a)$, and update to B, $(B' = B + b)$. To allow both the joins of $(a \bowtie B)$ and $(A' \bowtie b)$ be executed in parallel, a read lock on B is required for the duration of $(a \bowtie B)$, and so is a read lock on A. Since the joins are always interleaved by updates to the alpha memory lists, these locks essentially force a sequence constraint on the parallel joins.

To achieve parallel joins without locking on the entire alpha memory lists, the alpha memory lists involved in a join must remain constant for the entire duration of that join. To keep the alpha memory lists constant, we introduce an auxiliary data structure called "shadowed-alpha-memory" to store the newly inserted alpha memory cells from the newly created WME entries. The structural definition of a shadowed-alpha-memory list is exactly the same as alpha memory lists. There is one shadowed-alpha-memory list for each CE. The loop that processes the WME updates in Exec_Queue[] is modified so that each alpha memory cell from a newly created WME entry is inserted into its corresponding shadowed-alpha-memory list instead of the original alpha memory list.

The auxiliary shadowed-alpha-memory lists establish a clear boundary between the "old" alpha memory lists and the newly created shadowed-alpha-memory lists. The joins of $(a \bowtie B)$ and $(A \bowtie b)$ can therefore be executed in parallel without lockings or interruptions from the updates to the alpha memory lists. We call this type of joins between a 'new' alpha memory cell (a) and an 'old' alpha memory list (B) "regular joins". From the established equivalence between match and join in relational algebra, the remaining join calculation, in additional to the regular joins, can be derived by computing the set difference between the sequential form of (5.1), and and the parallel form of $((a \bowtie B) + (A \bowtie b))$ as follows:

$$\begin{aligned}
&((a \bowtie B) + (A' \bowtie b)) - ((a \bowtie B) + (A \bowtie b)) \\
&= ((a \bowtie B) + ((A + a) \bowtie b)) - ((a \bowtie B) + (A \bowtie b)) \\
&= ((a \bowtie B) + (A \bowtie b) + (a \bowtie b)) - ((a \bowtie B) + (A \bowtie b)) \\
&= (a \bowtie b) \qquad (5.2)
\end{aligned}$$

In summary, the joins in our "token parallelism" approach consist of regular joins between every pair of a new alpha memory cell and an "old" alpha memory lists, and the join among the new alpha memory cells.

Things become a lot more complicated when the number of CEs in a rule is greater than 2. Expanding the following form, (5.3), analogous to (5.2), we observe that the "extra work" for a rule with 3 CEs amounts to

$$\begin{aligned}
&((a \bowtie B \bowtie C) + (A' \bowtie b \bowtie C) + (A' \bowtie B' \bowtie c)) \\
&\quad - \quad ((a \bowtie B \bowtie C) + (A \bowtie b \bowtie C) + (A \bowtie B \bowtie c)) \\
&\quad = \quad ((a \bowtie b \bowtie C) + (a \bowtie B \bowtie C) + (A \bowtie b \bowtie c) + (a \bowtie b \bowtie c)) \qquad (5.3)
\end{aligned}$$

In general, for a rule with N CEs, in additional to the regular joins, the worst case of the total number of joins, such as $(a \bowtie B \bowtie C)$, is $\bigcirc (2^N)$. Such a complexity makes it impractical to generate separate join codes for every join at compile-time. It is also infeasible to spawn all of them as separate work units at run-time.

To reach a compromise between the granularity of token parallelism and its overhead, we developed Algorithm **PJOIN1**. The goal of Algorithm **PJOIN1** is to allow parallel joins while avoiding locks on the entire alpha memory lists, in a simple manner without much overhead. The idea can best be illustrated by (5.4) with the first term in (5.4) being the match result from the previous cycle. The second, third, and forth terms in (5.4) can be viewed as the joins initiated by the mew WME cells of a_1, a_2, and a_3, respectively.

$$\begin{aligned}
&((a \bowtie B \bowtie C) + (A' \bowtie b \bowtie C) + (A' \bowtie B' \bowtie c)) \\
&\quad - \quad ((a \bowtie B \bowtie C) + (A \bowtie b \bowtie C) + (A \bowtie B \bowtie c)) \\
&\quad = \quad ((a \bowtie b \bowtie C) + (a \bowtie B \bowtie C) + (A \bowtie b \bowtie c) + (a \bowtie b \bowtie c)) \qquad (5.4)
\end{aligned}$$

One can observe from these three terms that there exist a fixed pattern in the join loop ranges in each join, and the pattern is dictated by the index of the new WME cell. In other words, Algorithm **PJOIN1** first insert each new WME cell into the

corresponding shadowed-alpha-memory lists. An index number is assigned to each CE according to its absolute position in the rule body. Algorithm **PJOIN1** then loops through each shadowed-alpha-memory entry (defined as the join seed) and invokes one join. Each join is a (N-1) depth nested loops with the domain of each join loop set according to the relative position of its index to the index of the join seed. Specifically, a join with seed of index n will loop through all the old alpha memory[i], $\forall i < n$, and loop through all the old alpha memory[i] *plus* the newly built shadowed-alpha-memory[i], $\forall i > n$. Figure 5.4 lists the pseudo-code of Algorithm **PJOIN1** and Theorem 4 proves the its correctness.

```
Given a rule of N CEs, assuming the old AMem[] index is [1..N].
The old AMem[1..N], and the newly created WME trees, shadowed−AMem[],
are A[1..N] and a[1..N], respectively. Both A[1..N] and a[1..N] are already built.

PJoin1():
Begin
  For all i = [1..N] do {
    For each cell x in a[i] do {
      add join work Join(i,x) to the Work Queue;
    }                                                                    10
  }
End

Join(i,x):                          /* i:idx, x: join seed */
Begin
  integer k;                        /* join index */

  For all k = [1..i−1] {
    join on A[k];
  }                                                                      20
  For all k = [k+1..N] {
    join on (A[k]+a[k]);
  }
End
```

Figure 5.4: Algorithm **PJOIN1**

Theorem 4 *Given the conditions of Algorithm* **PJOIN1** *listed in Figure 5.4, the algorithm computes the exact same result as the sequential TREAT algorithm with*

no redundant work.

Proof: Given the conditions of Figure 5.4, the total amount of work needs to be computed is

$$((A[1] + a[1]) \bowtie (A[2] + a[2]) \bowtie \ldots \bowtie (A[N] + a[N])) \tag{5.5}$$

while the total amount of work in **PJOIN1** is

$$\begin{aligned}
&(a[1] \bowtie (A[2] + a[2]) \bowtie (A[3] + a[3]) \ldots \bowtie (A[N] + a[N])) \\
&+ \quad (A[1] \bowtie a[2] \bowtie (A[3] + a[3]) \bowtie (A[4] + a[4]) \ldots \bowtie (A[N] + a[N])) \\
&+ \quad (A[1] \bowtie A[2] \bowtie a[3] \bowtie (A[4] + a[4]) \ldots \bowtie (A[N] + a[N])) \\
&+ \quad \ldots \\
&+ \quad \ldots \\
&+ \quad (a[1] \bowtie a[2] \bowtie a[3] \ldots \bowtie a[N])
\end{aligned} \tag{5.6}$$

Through algebraic simplifications, the difference between the total work need to be computed and the total work in **PJOIN1** is

$$(5.5) - (5.6) = (A[1] \bowtie A[2] \bowtie A[3] \ldots \bowtie A[N]) \tag{5.7}$$

Since (5.7) is the join result among all old alpha memory lists, they already exist in the conflict set and need not be recomputed. □

By the position of each CE in a rule body, Algorithm **PJOIN1** achieves the token parallelism without using locks and maintaining a simple join code generation scheme. With the presence of –CE, however, again things become more complicated. Adding a WME to a –CE may invalidate some of the instantiations currently in the conflict set. Therefore, in the sequential TREAT algorithm, when a WME is added to a –CE alpha memory list, the actions are first to treat the –CE as a +CE and perform

the join. The result from the join then is searched through the entire conflict set for deletion.

For the first part of the "adding WME cells to –CE" work, the join part, Algorithm **PJOIN1** remains correct in computing the join result without any modification. The real problem, however, lies in the deletion phase. Specifically, since the joins are done in parallel without synchronization, there may exist an instantiation from the join of a –CE that is ready for the deletion phase but not yet present in the conflict set. The case in Figure 5.5 is one such example. The instantiation (a_1, b_2, c_2), from the –CE join of $(a_1 \bowtie b_2 \bowtie (c_1 + c_2))$ with b_2 as the join seed, may not exist in the conflict set when it is searched for deletion. The reason being that the instantiation, (a_1, b_2, c_2), is supposed to be generated by the join of $(a_1 \bowtie b_1 \bowtie c_2)$ where c_2 is the join seed. Since there is no synchronization between these two parallel joins units, one can not guarantee the instantiation will be inserted into the conflict set before its deletion.

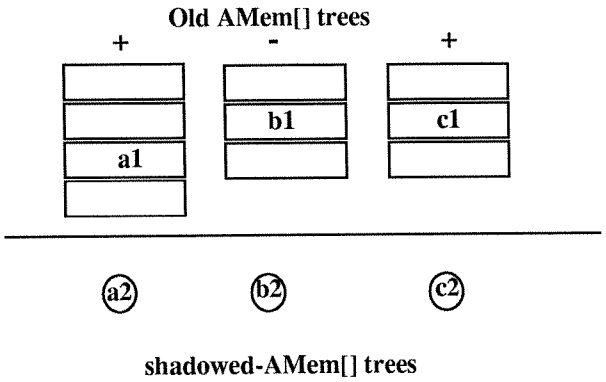

Figure 5.5: The Problem due to the presence of –CEs.

A simple solution to this problem due to the presence of –CEs, is to store all the join results seeded by –CEs in a temporary storage, wait for all the joins to complete, and then perform the deletion phase. Such an approach no doubt introduces additional time and space overhead and reduces the degree of parallelism.

```
Given a rule of N CEs, assuming the old AMem[] index is [1..N].
The old AMem[1..N], and the newly created WME trees, shadowed−AMem[],
are A[1..N] and a[1..N], respectively. Both A[1..N] and a[1..N] are built.
The polarity of the CE are stored in an array of Boolean, neg_flag[N].

PJoin2():
Begin
  integer pos_idx[1..N];                    /* store the positive index */
  integer j;
                                                                          10
  j = 0;
  For all i = [1..N] do {
          if (neg_flag[i] == TRUE ) do {
     pos_idx[i] = j;
     j = j + 1;
          } else
    pos_idx[i] = 0;                         /* for −CE */
  }
  For all i = [1..N] do {
    For each cell x in a[i] do {                                          20
      add join work Join(i,x) to the Work Queue;
    }
  }
End

Join(i,x):                                  /* i:idx, x: join seed */
Begin
   integer k;                               /* join index */
    if (neg_flag[i] == TRUE) then do {  /* −CE */
     For all k = [1..N], k < >i, neg_flag[k] == FALSE {                   30
      join on A[k];
          }
     For all k = [2..i−1], neg_flag[k] == TRUE {
      join on A[k];
     }
     For all k = [i+1..N], neg_flag[k] == TRUE {
       join on (A[k] + a[k]);
     }
    } else then {                           /* +CE */
     For all k = [1..i−1] {                                               40
       join on A[k];
     }
     For all k = [k+1..N] {
       join on (A[k]+a[k]);
     }
          }
End
```

Figure 5.6: Algorithm **PJOIN2**

Therefore, we modify Algorithm **PJOIN1** into Algorithm **PJOIN2** (in Figure 5.6), so that the special cases due to the presence of –CEs can be dealt without any temporary storage. The key ideas of the modification to the Algorithm **PJOIN1** are based on the following observations:

1. An instantiation only contains WMEs of +CEs. In the case of the example in Figure 5.5, the instantiation in question should be (a_1, c_2) instead of (a_1, b_2, c_2).

2. Any instantiation containing WME entries exclusively from the "old" alpha memory lists is guaranteed to be in the conflict set at the beginning of the current match phase.

3. To avoid the use of temporary storage to hold the join results seeded from –CEs, one needs to insure that every instantiation that is to be deleted from the conflict set due to the presence of -CEs guarantees to exist in the conflict set during the deletion phase.

By combining item 1, 2, and 3 above, we conclude that the only way to insure instantiations from –CE joins always exist in the conflict set during deletion is to rearrange the ranges of the join loop such that the join loop invoked by a –CE always deals exclusively with "old" +CE alpha memory lists. Algorithm **PJOIN2** in Figure 5.6 achieves such a goal by enforcing an order on all CEs so that all +CEs precede -CEs when considering the join loop ranges.

Lemma 1 *Given a rule of N CEs, the join order of the N CEs does not affect the correctness of Theorem 4.*

Proof: Since the join operator is commutative and associative, the proof of Theorem 4 always holds regardless of the join order. □

Theorem 5 *Given the conditions of Algorithm* **PJOIN2** *in Figure 5.6, the algorithm computes the exact same result as the sequential TREAT algorithm without synchronization among joins and without auxiliary storage for deletion of instantiations.*

Proof: The proof of completeness can be derived from the equivalence between CREL LHS and relational algebra, i.e., the match work for a rule $(A - B)$ can be expressed in terms of the following query form:

$$A - \Pi(A \bowtie B)$$

Since all the –CE joins are initially treated as +CE joins and Algorithm **PJOIN2** essentially reorders the join order of each join, with Theorem 4 and Lemma 1, Algorithm **PJOIN2** always computes the exact same join results as the sequential TREAT algorithm.

As for the proof of the second part of the theorem, because all the joins invoked by –CEs can only produce results exclusively from "old" +CE alpha memory lists, through the observations mentioned above, one can conclude that indeed every instantiation from the joins on –CE is guaranteed to be present during the deletion phase. □

Notice that we no longer claim, as in Theorem 4, that there is no redundant work done in Theorem 5. This is because there may exist a sequential match that will process a –CE join earlier than **PJOIN2**, thus reducing both the sizes of alpha memory lists and the conflict set.

Also notice we did not mention the match work invoked by removing a WME cell. The only case where a join is invoked while removing a WME cell is removing a WME of a –CE. The complications of deleting alpha memory entries, coupled by the requirement of maintaining the conflict set up to date, make it

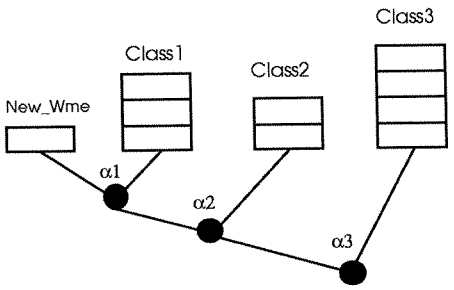

Figure 5.7: Join estimation and decomposition.

unattractive to develop a parallel implementation. Furthermore, since the percentage of match time spent in a system on deleting a –CE WME cell is small, we decide to keep the original sequential version in dealing with removing of WME cells.

After all the parallel joins complete, each shadowed-alpha-memory list is linked to its original alpha memory list. Therefore, the space overhead of Algorithm **PJOIN2** is small since the new alpha memory cells are only allocated once. By the use of shadowed-alpha-memory lists, rule-level match parallelism is achieved without locks on the alpha memory lists or WMEs and without any auxiliary storage. We shall point out, however, that the use of shadowed-alpha-memory structures introduces additional overhead in managing the run-time data structures, since organizing shadowed-alpha-memory lists has to be performed sequentially. Chapter 7 discusses in detail the issues of overhead from these sources of parallelism.

5.2.1 Intra-Token Parallelism and Join Decomposition

While token parallelism takes care of the joins initiated by each WME token, Table 5.1 shows that, for some systems, it is insufficient to obtain effective match parallelism by token parallelism alone. This section discusses issues related to the decomposition of a single join. We call the parallelism from such a decomposition "intra-token parallelism".

Assuming, at the beginning of an execution cycle, the join invoked by a newly WME on a rule with 4 CEs is given in Figure 5.7, where the current size of alpha memory list is N_i and the join selectivity is α_i, respectively. The sequential depth-first nested join loop will take $O(N_1 * \alpha_1 * N_2 * \alpha_2 * N_3)$ operations to complete, where an "operation" mainly contains pointer references to shared memory locations and comparison operations. The question one would ask is what is an optimal decomposition, if any, of such a join.

To model this problem and simplify the analysis, the following assumptions were made: assume that the cost the join is C_i, and the join can be arbitrarily partitioned into subtasks by ways of building hash index on some CE. To model the variance of the join cost, the join cost function is assumed to be a random variable with a normal distribution of mean C_i and standard deviation σ. There are two reasons to assume a normal distribution: the random variable can be equally divided into N smaller i.i.d(independent and identical distribution) random variables of mean C_i/N and variance σ/N, and a normal distribution satisfies the requirement of Kruskal and Weiss's conditions [36], whose analysis is the basis of our decomposition scheme.

Before we proceed, a brief summary of K&W's analysis and their results is given as follows: Assuming there are n independent tasks waiting to be scheduled, each with an i.i.d. distribution cost function of mean μ and variance σ. There are P processors available for use. Through the analysis of the constructed Markov model, K&W concluded that the best way to schedule the n units of work on P processors is to have a batch allocation with an allocation size being K=n/P. In other words, the best way to schedule these n units of work is to allocate K units per processors until the work is exhausted. They also concluded, under such allocation scheme, that the total execution time for the entire system is

$$T_{total} = K\mu + \sigma\sqrt{2K \log P} \tag{5.8}$$

Since the decomposed join subtasks are mutually independent, K&W's analysis can directly be applied to our case. In order to do so, we further assume, for the moment, that there are M PEs reserved exclusively for the cluster. The known parameters in our case are the total cost C_i, and the goal is to partition the join into n subtasks. Since T_{total} in (5.8) is optimal when K=n/M, the optimal join decomposition can be achieved by making K=1 and n=M, and the parallel join cost, T_{total} becomes

$$T_{total} = \frac{C_i}{P} + \sigma_i \sqrt{2 \log M} \tag{5.9}$$

The first term in (5.9) is the average cost to complete decomposed subtasks in parallel, assuming all M PEs are available. The second term in (5.9) reflects the imbalance of the task decompositions.

We know, therefore, if the above assumptions are indeed valid, then the optimal way to decompose a join is always to divide the join equally into M parts, where M is the number of available PEs. Unfortunately some of the assumptions above do not hold in a real CREL system – not only are there multiple sources of work pools (multiple clusters being active), but the activations of such clusters are also nondeterministic. In other words, whenever there is a join from cluster CL_i ready for decomposition, one can not quickly predict the exact number of available PEs. In addition, there are work requests from other active clusters that need to be taken into considerations.

One way to analyze such a dynamic system is to use a model of multiple arrivals with multiple service queues. In our system, however, the activation/reactivation process of clusters can not be simply modeled as random processes with some distribution functions. Therefore, we take a different approach to solve the problem. Assume there are two active clusters, CL_i, CL_j, each with its join cost

as C_i and C_j respectively. The number of available processors is M. We would like to compare the following two allocation methods:

[No boundary in M PEs]

Let CL_i and CL_j both have access to all M PEs. Since CL_i and CL_j are independent clusters, regardless of the execution sequence of the subtasks from CL_i and CL_j, the total time, T^{seq}_{total}, is

$$T^{seq}_{total} = T_i + T_j = \frac{C_i + C_j}{M} + (\sigma_i + \sigma_j)\sqrt{2\log M} \tag{5.10}$$

[Weighted Partition]

Partition M PEs into M_i and M_j, where M_i is the number of PEs assigned to C_i, in proportion to the costs of CL_i and CL_j. Assume $C_i = \beta C_j$ and $\sigma_1 = \beta\sigma_2$, $\beta > 1$. Then

$$M_i = \left(\frac{\beta}{1+\beta}\right) M, \quad M_j = \left(\frac{1}{1+\beta}\right) M \tag{5.11}$$

$$T_i = \frac{C_i}{M_i} + \sigma_i\sqrt{2\log M_i} = \frac{(1+\beta)\,C_j}{M} + \beta\sigma_j\sqrt{\frac{\left(2\log\frac{\beta M}{1+\beta}\right)}{\frac{\beta M}{1+\beta}}} \tag{5.12}$$

$$T_j = \frac{C_j}{M_j} + \sigma_j\sqrt{2\log M_j} = \frac{(1+\beta)\,C_j}{M} + \sigma_j\sqrt{\frac{\left(2\log\frac{M}{1+\beta}\right)}{\frac{M}{1+\beta}}} \tag{5.13}$$

Since $\beta > 1$, we know $T_i > T_j$. Thus the total time for the partitioned case, T^{par}_{total}, is

$$\begin{aligned} T^{par}_{total} &= \max(T_i, T_j) \\ &= T_i \\ &= \frac{C_i}{M_i} + \sigma_i\sqrt{2\log M_i} \end{aligned} \tag{5.14}$$

Comparing (5.10) and (5.14), we observe that the terms of steady state, $\left(\frac{(1+\beta)C_j}{M}\right)$, are the same. By eliminating the terms from steady states, we get

$$\begin{aligned}
\frac{T^{par}_{total} - (\text{steady state})}{T^{seq}_{total} - (\text{steady state})} &= \frac{(\beta+1)\,\sigma_j\sqrt{2\log M}}{\beta\sigma_j\sqrt{\frac{2\log\frac{\beta M}{1+\beta}}{\frac{\beta M}{1+\beta}}}} \\
&= \left(\frac{\beta+1}{\beta}\right)\left(\frac{\sqrt{2\log M}}{\sqrt{2\log\frac{\beta M}{1+\beta}}}\right)\left(\sqrt{\frac{\beta}{1+\beta}}\right)\left(\sqrt{M}\right) \\
&= \left(\sqrt{\frac{1+\beta}{\beta}}\right)\left(\frac{\sqrt{\log M}}{\sqrt{\log\frac{\beta M}{1+\beta}}}\right)\left(\sqrt{M}\right) \qquad (5.15)
\end{aligned}$$

(5.15) tells us that if M is large and β is moderate, the overhead due to imbalance in [Weighted Partition] approach is most likely greater than that in [No boundary]. In other words, it is better to assume all M PEs are available to both clusters, CL_i, CL_j, than to have fixed partitions on the M PEs into two disjoint sets, one for each cluster. Furthermore, maintaining run-time information such as M and the costs of C_i and C_j results in additional costs. We therefore choose to take the [No Boundary] approach.

To summarize, the analysis above tells us that in a CREL system with multiple clusters active and M PEs, when decomposing a potentially large join from one cluster, it is better to assume all M PEs are available and decompose the join into M subtasks, than to have a fixed partitions of PEs on the clusters. Notice that since optimality is reached independent of the actual values of $C_i$andC_j, the run-time manager can accomplish an optimal decomposition without the knowledge of the join costs of every cluster at any given time.

5.3 Conflict Resolution and Run-Time Checking

A major source of CREL parallelism that needs exploration is for each cluster to fire multiple instantiations from the conflict set in a cycle. The requirement

of single rule firing originates from OPS5; the CREL execution model does not restrict simultaneous firing of multiple instantiations in one cycle.

At the end of the match phase where the conflict set is fully enumerated, the goal of conflict resolution is to compute a maximum subset of instantiations from the conflict set so that they can be fired simultaneously without violating the CREL execution requirements. This section discusses the issues related to the conflict resolution. Specifically, given a conflict set, one needs to first check whether each pair of instantiations in the set can fire simultaneously. Secondly, given the results of the above checking, one needs to compute an optimal set of instantiations that can be fired in the current cycle.

5.3.1 The Problem

"Multiple rule firing" means, within each cluster, the system is cable of firing multiple instantiations within a cycle. The instantiations fired can be from the same rule or different rules. [4] Empirical studies show that, in a typical CREL system, a number of rules are responsible for expanding the problem search space by creating temporary WMEs. In such an expanding stage, the actions of an instantiation tend not to interfere with other instantiations in the conflict set. Therefore, without multiple rule firings in an expanding stage, a system spends most of the time maintaining the complex data structures, such as α memory lists and the conflict set, whereas most of these work can in fact be performed in parallel.

There are two aspects of the approach to allow multiple rule firings within a cluster. Given a pair of instantiations, the first aspect, defined as [RTC1] subproblem, is to check whether simultaneous firing of the pair of instantiations violates the CREL

[4] Issues related to multiple rule firings across clusters were discussed in chapter 4.

serializability requirement. In other words, we need to check whether the pair of instantiations interferes with each other. The second aspect, defined as [RTC2] subproblem, is to find out, given a conflict set and its interference relations, the "best" subset of instantiations [5] that is simultaneously firable.

To design a solution to both problems, RTC1 and RTC2, one should constantly be reminded of an important criteria: cost-effectiveness, since the cycle time of each cluster in a CREL system is small, and for every cycle and every cluster, the RTC checking needs to be performed on all pairs of instantiations in the conflict set. The following example illustrates the complexity of the RTC problem: assume the number of instantiations in the conflict set is 100, and there are 25 rules in a cluster, with a average of 5 CEs and 3 actions per rules, the total number of RTC checking per cycle, then, is on the order of 3000. In addition, after the checking is done, one still needs to compute the maximum subset from the conflict set. The cost of such computation varies depending on the interference relations, but undoubtedly will be high. Therefore, the dynamic nature of the problem prohibit us to perform exhaustive search of an optimal solution in every cycle.

5.3.2 Previous Work and Related Issues

Ishida first studied the potential of allowing multiple instantiations to fire in each cycle [29]. He suggested a method similar to the standard compile-time analysis by building an interference graph and then traverse the graph to find cycles. Schlmoze presented a formal model to solve this problem with an fundamentally similar approach[58]. Schlmoze's solution is to build interference graph and compute cycles, with a theoretical foundation from logic. Both approaches neglect the most

[5] The best candidate in terms of both optimality of the solution and the run-time overhead associated with computing such a solution.

important priority, reduction of run-time overhead.

5.3.3 Problem RTC1

From Section 5.3.1, RTC1 problem is formally defined as followed: Given the conflict set CS, and two instantiations I_m and I_n ($I_m, I_n \in CS$), RTC1 is to decide whether simultaneous firing of I_m and I_n guarantees serializability. Assume I_m and I_n are instantiations from rule P_i and P_j respectively. Without loss of generality, [6] we further assume P_i and P_j take the form of

$P_i \triangleq LHS_i \rightarrow RHS_i$, $LHS_i \triangleq (A_1) \wedge_{f_i} (\neg B_1)$,

$P_j \triangleq LHS_j \rightarrow RHS_j$, $LHS_j \triangleq (A_2) \wedge_{f_j} (\neg B_2)$, where

A_1, A_2, B_1, B_2 are CEs, and

f_i, f_j are join functions for rule P_i, P_j respectively.

Join function f_i defines the join operations between CEs in LHS_i. [7] RHS_i actions include **make** new WMEs, **remove** WMEs from the instantiation satisfying LHS_i, and **modify**, which is implemented as a (**remove, make**) pair.

The idea of the run-time consistency check is fundamentally the same as the CREL compile-time dependency analysis. In other words, simultaneous firing of I_m, I_n violates the CREL serializability requirement if and only if the actions of one instantiation violate(interfere with) the LHS conditions of the other instantiation. Therefore, for interferences to exist between I_m and I_n, at least one of the following two cases must be true:

1. The firing of $RHS_i(I_m)$ interferes with I_n, or

[6] See [15], [38] for complete definition of CREL and OPS5 syntax definition.

[7] See Section 4.2.1 for details of the equivalence between CREL LHSs and relational algebra expressions.

2. the firing of $RHS_j(I_n)$ interferes with I_m.

Because of the duality of the problem, we only need to analyze the case of "$RHS_i(I_m)$ interferes with I_n". Analogous to the strategy in CREL compile-time analysis, out of the four combinations of (+,–) CEs in a LHS and **make, remove** actions in a RHS, only the two following cases may cause interferences:

[RTC1.a] $RHS_i(I_m)$ removes a WME in I_n, or

[RTC1.b] $RHS_i(I_m)$ makes a new WME, which invalidates $LHS_j(I_n)$.

These notations and the case analysis can best be illustrated by using the example in Figure 5.8.

Assume there are two rules P_1, P_2 and the conflict set are:

```
(literalize A a1 a2)
(literalize B b1 b2)
(literalize C c1 c3)
(P P1                                   (P P2
  (A  ↑a1  <x>)                           (B  ↑b1  <m>)
  (B  ↑b1  <x>  ↑b2 <y>)                  (C  ↑c1  <n>)
 -(C  ↑c1  <y>)                           -->
  -->                                     (make C  ↑c1 ( <n>+1) )
  (make A  ↑a1  <x+1>)                    )
  (remove 2)                                                    10
)
```

$I_1 = [(A\ 1)(B\ 1\ 2)], (<x>=1, <y>=2),$
$I_2 = [(B\ 1\ 2)(C\ 1)], (<m>=1, <n>=1),$
$I_3 = [(B\ 1\ 3)(C\ 2)], (<m>=1, <n>=2),$ and
$I_4 = [(B\ 1\ 3)(C\ 1)], (<m>=1, <n>=1)$

Figure 5.8: An example of RTC1.

Since firing I_1 removes a WME (B 1 2) and (B 1 2) is in I_2, $RHS_1(I_1)$ satisfies the condition of [RTC1.a]. Firing I_2, on the other hand, creates a new WME,

(C 2), which invalidates $LHS_2(I_2)$ because of the binding of variable $< y >$. we conclude that $RHS_2(I_2)$ satisfies the condition of [RTC1.b].

If WME classes are represented by a universal relation [64] and LHSs are translated into relation algebra forms, instantiations in the conflict set then become the tuples of the join functions results. Furthermore, the RHS actions, *make* and *remove*, can be represented as set addition and subtraction operators respectively. Define RHS_i^+ as the set of newly created WMEs from the *make* actions of RHS_i, and define RHS_i^- as the set of WMEs being removed by the *remove* actions of RHS_i. The problem of [RTC1.a] then is translated to a set intersection problem as to decide whether

$$(RHS_i^-(I_m) \wedge I_n = \emptyset) \tag{5.16}$$

For the example in Figure 5.8,

$$(RHS_1^-(I_1) \wedge I_2) = (\{(B\ 1\ 2)\} \wedge \{(B\ 1\ 2)(C\ 1)\}) \neq \emptyset$$

$$\Rightarrow \{I_1, I_2\} \text{ can't be fired at the same time.}$$

A straight forward approach to solve [RTC1.b], however, involves join operations between I_n and the changed portion of WME space from RHS_i^+.[8] Both the complexity of the join operations and the overhead associated with building links and indices on the join data prohibit such an approach.

To avoid the join, we observe the bottleneck, as in the compile-time analysis, exists because we attack the interference problem by analyzing the entire conflict set instead of the individual instantiation. For problem RTC1.b, however,

[8]This is the foundation of all the incremental match algorithms, such as RETE and TREAT.

this is unnecessary since we only need to consider one instantiation pair, I_m, I_n, where all the attributes are fully instantiated. Thus, for [RTC1.b] to check if any of the newly created WME from $RHS_i^+(I_m)$ invalidates $LHS_j(I_n)$, we view each CE in $LHS_j(I_n)$ as a *fact* and check if a contradiction exists between all the *facts* from $LHS_j(I_n)$ and the newly created WMEs from $RHS_i^+(I_m)$. A contradiction exists if and only if there exists opposite facts, such as (A 1 2) and -(A 1 2). Thus problem [RTC1.b] becomes deciding if contradiction exists [9] in

$$RHS_i^+(I_m) \wedge LHS_j(I_n) \tag{5.17}$$

For the example in Figure 5.8,

$$\begin{aligned}(RHS_2^+(I_2) \wedge LHS_1(I_1)) &= (\{+(C\ 2)\} \wedge \{(A\ 1)(B\ 1\ 2) - (C\ 2)\}) \\ &= F \\ &\Rightarrow \{I_1, I_2\} \text{ can nott fire simultaneously.}\end{aligned}$$

Notice (5.16) is a set expression while (5.17) is a boolean expression. A uniform representation is needed to integrate (5.16) and (5.17). To convert (5.16) to a boolean expression, each WME in RHS_i^- is represented as a negative *fact* and each WME in instantiation I is treated as a *fact*. Putting (5.16) and (5.17) together, for a instantiation pair (I_m, I_n) (from rules P_i, P_j respectively), whether I_m interferes with I_n is equivalent to

$$\begin{aligned}(5.16) \wedge (5.17) &= (RHS_i^-(I_m) \wedge I_n) \wedge (RHS_i^+(I_m) \wedge LHS_j(I_n)) \\ &= RHS_i(I_m) \wedge LHS_j(I_n) \\ &\quad \text{since } I_m \in LHS_j(I_m) \text{ and } RHS_i^+ \wedge RHS_i^- = RHS_i\end{aligned}$$

[9] The expression is always false

Putting everything together, the run-time check function between an pair of instantiations, I_m, I_n, from rule P_i, P_j is defined as $RTC1_{ij}$, where

$$RTC1_{ij} \triangleq ((RHS_i(I_m) \wedge LHS_j(I_n)) \wedge (RHS_j(I_n) \wedge LHS_i(I_m))) \tag{5.18}$$

Since the complexity of (5.18) is still high for run-time execution, ideally, one would like to optimize such check functions between at compile-time, with the instantiation pair as parameters. An easy way to represent instantiations is by variable bindings. Assume I_i and I_j be represented as $(x_1, x_2, \ldots, x_m)$ and $(y_1, y_2, \ldots, y_n)$, where m, n are the total numbers of local variables in RHS_i, RHS_j, respectively. (5.18) then becomes a boolean function of a tuple variable $(x_1, \ldots, x_m, y_1, \ldots, y_n)$, as

$$\begin{aligned} RTC1_{ij} \;=\; & ((RHS_i(x_1, \ldots, x_m) \wedge LHS_j(y_1, \ldots, y_n)) \wedge \\ & (RHS_j(y_1, \ldots, y_n) \wedge LHS_i(x_1, \ldots, x_m)) \end{aligned} \tag{5.19}$$

Again using the example in Figure 5.8, the check function $RTC1_{12}$ becomes

$$\begin{aligned} & ((\{(A\ x_1)(B\ x_1\ x_2)\ -(C\ x_2)\} \wedge \{(C\ y_1 + 1)\}) \wedge \\ & \quad (\{(B\ y_1)(C\ y_2)\} \wedge \{(A\ x_1 + 1)\ -(B\ x_1\ x_2)\}) \end{aligned} \tag{5.20}$$

Since all the class names in (5.20) are constants, further simplification shows the the truth value of $RTC1_{12}$ in (5.20) is equivalent to

$$((y_1 + 1 \neq x_2) \wedge (x_1 \neq y_1)) \tag{5.21}$$

(5.20) and (5.21) illustrate that for each rule pair (P_i, P_j), we can construct, at compile-time, a checking function ,$RTC1_{ij}$, that usually involves only simple comparisons between attribute values. The rare exceptions are cases where RHS actions contain some complex *compute* or external function calls. These cases are addressed when we discuss the implementation details of [RTC1].

Implementation Details The actual implementation of [RTC1] is too tedious for us to explain. Instead, we will explain the key components in computing (5.19). From (5.20) and (5.21), we know only (+/–) pairs of facts of the same class may cause "contradictions". To check if a pair of rules, P_i, P_j, indeed causes contradictions, for every attribute testing in LHS_i, we need to check it against the same attribute specification in RHS_j. In the example in Figure 5.8 and (5.20), the checks are $(y_1 + 1 \neq x_2)$ for class C, and $(x_1 \neq y_1)$ for class B. Each pair of attribute checking, in general, can be expressed as

$$OP(attr_{lhs}, attr_{rhs}) \tag{5.22}$$

where OP can be any boolean operator specified by the CREL LHS grammar, $attr_{lhs}$ contains constants or variables, and $attr_{rhs}$ can be any CREL RHS attribute terms, possibly with *compute* or external function calls.

Because of the way CREL's match mechanism is implemented, checking between a pair of *remove* RHS action and +CE can be optimized to a simple WME address comparison. Therefore, given a pair of rules P_i, P_j, the [RTC1] algorithm loops through the RHS actions of P_i and check against each action/CE pair to see if they are from the same class and with different signs. After all the techniques from compile-time analysis, such as constant propagation and disjoint attribute sets, etc., are applied, a list similar to (5.22) is dumped for the check function generator, if no compile-time analysis can be used to determine the two WME specifications are different.

Figure 5.9 gives a more complicated example, with an output of the [RTC1] analysis on two instantiations from the same rule, *tmp*. The first number in each list in the output identifies the type of testing: 0 for WME address comparison, and 1 for boolean testing. For example, given two instantiations from the same rule *tmp*,

```
(literalize classA A1 A2 A3 A4)
(literalize classB B1 B2 B3 B4)
(p tmp
   (classA  ↑A1 > 20  ↑A2 <x>  ↑A3 <y>)
   (classB  ↑B1  <x>)
   -(classB  ↑B1  <x>)
   -->
   (bind <z1> (compute <x> + <y>))
   (bind <z2> (compute <x> - 20))
   (modify 2  ↑B1 <z1>)
   (make A  ↑A1 <z2>)
   ))
```

RTC1 Output:

```
((0 2 2)
 (1 ( ((COMPUTE ((1 3) + (1 4)))
     ((EQ (VAR-REF (1 3))))) )))
```

Figure 5.9: A More Complex Example of [RTC1]

I_1, I_2, the first check function, (0 2 2), is to check if the second WME in I_1 is the same as the second WME of I_2. If they are indeed the same, the execution of rule *tmp*'s action will modify, therefore remove, the second WME. When making a new WME of class B, on the other hand, we need to check if the B1 attribute ($<x>+<y>$) is the same as $<x>$, as the second list in RTC1 output indicates. Notice that at run-time, all the variable references, such as $<x>$, and $<y>$, are converted to a (WME, attribute) index pair into instantiations. In this case, $<x>$ and $<y>$ are converted to (1 3) and (1 4), respectively.

Generally speaking, for each pair of rule, further reductions among these multiple boolean expressions is feasible to reduce the overhead of the run-time checking functions. An such example is given in Figure 5.9 where the EQ testing between ($<x> + <y>$) and $<x>$ can be reduced to a checking of ($<y>=0$). Although such reductions of multiple boolean expressions are difficult in general cases – there is no restriction on both the numbers of variables and the degrees

of variables involved in the boolean forms –, actual results from the analysis on benchmarks show that the majority of the checking functions generated contain simple WME address comparisons or simple boolean expressions on linear orders of some free variables. For those checking functions that do contain either *compute* or external functions, they are rarely complex.

Once the checking functions is generated in the form as in Figure 5.9, the RTC1 function generator then generate a matrix of C function pointers and the actual function codes for CREL run-time system calls.

5.3.4 Problem RTC2

In the conflict resolution stage of each execution cycle, assuming the size of the conflict set(CS) is N, the RTC1 checking is performed on all pairs of instantiations in CS, and the interference relations are kept in a matrix called RTC[N][N], problem RTC2 is to find a maximum subset, CS′, from CS, such that firing all instantiations in CS′ guarantees serializability, or $\forall I_i, I_j, \in CS'$, $RTC1_{ij}(I_i, I_j)$ is true. The best known sequential algorithm to compute disjoint subsets, such as CS′, is of complexity N^2[53] , N being the size of the graph. To avoid potentially severe run-time overhead, we select the following grouping algorithm of O(N) complexity to balance the tradeoff between the "goodness" of the algorithm and the run-time costs. We choose not to implement a parallel version of the grouping algorithm to avoid the synchronization overhead and further complications on the data structure design. Problem RTC2 can be solved by first constructing an interference graph, IG, and then by grouping nodes in IG into subsets to find out CS′. A detailed descriptions of these two steps are:

1. The graph IG is first constructed by assign one node for each instantiation in CS. We then loop through all instantiation pairs in CS. For each pair (I_i, I_j) in

CS $RTC1_{ij}$ is computed to determine if (I_i, I_j) can be fired simultaneously. An edge between nodes (I_i, I_j) is drawn *iff* $RTC1_{ij}(I_i, I_j)$ is false.

2. To find a subset, CS′, we use the grouping algorithm in Figure 5.10.

```
Given IG = (V,E), where V = { Ii | Ii in CS },
                E = { Eij | RTC1mn(Ii, Jj)= F}
                Ii is from rule Pm, Ij is from rule Pn.

Let CS′ = empty set, /* the result variable */
neighbor(Ii) = neighbor nodes of Ii

Begin
  V := Sort V into ascending order of the degree of nodes;
  While V  < > 0 do {                                              10
      Vhead = The head of V;
      V = V − {Vhead};
      CS′ = CS′ + { Vhead };
      V = V − neighbor(Vhead);
  }
End;            /* CS′ holds the set of instances for firing */
```

Figure 5.10: Grouping Algorithm for RTC2

The first part of solving RTC2, constructing the interference graph, has no short cut but to compute all pairs of instantiations in the conflict set, which is of $O(N^2)$, N = |CS|. However, since each instantiation will have to be inserted in the conflict set in order, the first O(N) can be embedded in the match stage, when individual instantiation is inserted in the conflict set. For each instantiation I_i, the second order of O(N) occurs when I_i needs to check against all other instantiations in the conflict set. Here is a potential source for parallel implementation. We will discuss such a possibility in detail in Section 6.4.

The second part of solving RTC2, finding CS′ by the grouping algorithm in Figure 5.10 is of worst case complexity O(N). Again, possibility of a parallel version is studied in Section 6.4.

Theorem 6 *The grouping algorithm in Figure 5.10 is guaranteed to find one subset (CS') from CS such that simultaneous firing of all instantiations in* CS' *guarantees serializability.*

Proof: Since no edge among any node pair in CS' implies no interference exists among these node, we only need to prove there exists no edge in CS'. This can be observed easily since during the construction of CS', all neighbors are removed. □

The Effects of Clustering on Run-Time Checking The purpose of the run-time consistency checking is to allow firing of multiple instantiations for each cluster, within every cycle. The final objective of doing this is to speed up the execution time by reducing the total number of cycles. Computing [RTC1] ad [RTC2] introduces overhead that does not exist in the sequential execution cycles of OPS5. Efficiency, therefore, is our most important criteria.

The complexities of solving both [RTC1] and [RTC2] depend on N, the size of the conflict set(s) involved, which in turns depend on the total number of rules in a system. From the results in Table 4.1, static clustering can reduce the rule number of a cluster by an order of magnitude. Since the firing of two rules from two clusters has no effect on the correctness of the execution, run-time consistency checking needs not be performed on a pair of instantiations from different clusters. Static clustering, therefore, can reduce the complexity of both [RTC1] and [RTC2] significantly. We will elaborate on this in Chapter 7, where real data can be plotted to illustrate such effects. section

5.4 Conclusions

In this Chapter, we first studied the potential sources of CREL run-time parallelism from the viewpoint of a cluster. Two sources of parallelism were iden-

tified: the match parallelism and the conflict resolution strategy for multiple rule firing. For parallel match, two levels of granularity were recognized for the purposes of reducing run-time overhead–such as locking and avoiding large number of tasks, and balance of the system loads.

As for the resolution strategy for multiple rule firing, we designed a two step method to allow simultaneous firing of multiple instantiations within each cluster without taking up too much resources at run-time.

Up until now, efforts were spent on various individual sources of parallelism in the CREL run-time environment. From a global viewpoint, the imminent task is to consider all of these sources of parallelism and design an efficient run-time manager. The next chapter discusses the integrations of these sources of run-time parallelism and describes the global CREL run-time manager and some of the design rationales.

Chapter 6

CREL Run-Time Management

In Chapter 4, we discussed various aspects of compile-time dependency analysis and transformations to improve parallelism in the dependency graphs. In Chapter 5, we discussed the run-time characteristics of a typical CREL system, and the potential sources of run-time parallelism worth exploiting. The remaining issue now is how to efficiently and effectively manage these tasks at run-time to balance the system load and optimize the total execution time.

In this chapter, we first give a brief background description of the general problem of task assignment and load balancing. We then discuss the feasibility of applying various conventional methods to our systems. The remaining parts of the chapter discuss in detail the proposed model, rationales, and some important implementation issues.

6.1 Background

There have been many studies of task assignment and load balancing of programs in distributed systems. Assigning a set of computational tasks onto a distributed system in general is a extremely complex problem. Studies based on graph theories normally model the computations in terms of static acyclic graphs (DAGs) with nodes representing computation tasks and edges representing data communications and precedence relationships among nodes. Physical processors

are modeled as processor graphs. The task assignment problem then becomes a problem of graph matching between the computation and the processor graphs. This class of problems has been proven to be in class *NP*[19, 34]. Various heuristics, simplifying assumptions, and some special cases of the objective functions have been studied. Stone, for instance, studies optimal scheduling of two processor systems[63] by applying network flow algorithms. Monma presented some linear-time scheduling algorithms for special cases of N equal-length tasks with tree-like precedence relations on homogeneous processors[48]. Mcdowell studies the optimal scheduling of acyclic program graphs on linearly-connected processor architectures by graph coloring algorithms[43]. Kim presented a generally applicable scheduling algorithm based on the linear-cluster technique[35].

When precedence relationships are not available or ignored, the task assignment problem becomes a 0-1 integer linear or quadratic programming problem. This class of problems with more than three physical processors again is in class *NP*[3]. Various heuristic algorithms have been designed for some special cases to reach near optimal solutions. Kung and Irani, for example, both study the problem with the objective of minimizing interprocessor communication in MIMD and SIMD architectures[37],[2]. Arora studies two heuristics for the case where inter-process communication costs are functions of the computational tasks involved instead of functions of the physical architectures[33]. He further explored the solutions of cases where additional constraints such as limited storage resources are present[32]. Lo studies cases where different objective functions such as minimizing total execution time and communication costs are weighted at the same time[39]. Kruskal uses a probabilistic model to derive closed forms of the expected total execution time based on the assumption that the computation costs are independent random variables[36]. Hariri uses branch-and-bound techniques to solve the task assignment problem with the objective to maximize reliability and minimize delays[26]. Sinclair uses branch-

and-bound with underestimate techniques to optimize total execution time including computation and communication times of all tasks in the system[60].

All of the approaches mentioned above deal with static computations without run-time load balancing. When the behavior of the system is highly dynamic, however, additional run-time load balancing is needed to balance the overall system loads and increase the system utilization. There exist numerous studies on load balancing in distributed systems and computer networks. Load balancing can be categorized into centralized and decentralized decision mechanisms with a variety of transfer (why to migrate) and migration (where to migrate) policies[9, 31, 14]. Load balancing can be preemptive or non-preemptive. Several excellent surveies and comparisons of various policies and decision algorithms can be found in [12, 13, 65]. In general, the criteria of an effective load balancing scheme are:

- Policies such as transfer and migration policy should be kept as simple as possible due to run-time overhead.
- The main objective of load balancing is keeping all PEs in the system busy instead of averaging the loads of all PEs.
- The design of the decision policies should avoid process thrashing where the frequency of process migration is so high that the associated overhead degrade the overall system performance.

6.2 Problem Statement

CREL run-time manager's responsibility is to effectively partition CREL tasks and coordinate the execution of these tasks under the resource constraints, such that the total execution time of the CREL program is optimized. From Chapter 5, the CREL run-time tasks from every cluster can further be categorized as work from

the match, conflict resolution, and act stages of the execution cycles. In terms of our target machine with shared-memory architecture and UNIX operating systems, the resource constraints are mainly from the processing elements since the run-time allocation of data objects on the physical memory is beyond the control of the CREL run-time manager and the allocation of both data and executable codes are also irrelevant for bus-based shared memory systems. The abstract model for CREL run-time environment is:

> Given a CREL system, with the cluster set $\mathcal{CL}$ and processing elements PE.
>
> $\mathcal{CL} = \{\mathrm{CL}_1, \mathrm{CL}_2, \ldots, \mathrm{CL}_\mathrm{N}\}$, $\mathcal{PE} = \{\mathrm{PE}_1, \mathrm{PE}_2, \ldots, \mathrm{PE}_\mathrm{M}\}$,
>
> where $\mid \mathcal{CL} \mid = \mathrm{N}$ and $\mid \mathcal{PE} \mid = \mathrm{M}$.
>
> The run-time manager needs to provide effective allocation of works from all clusters onto physical processors such that the total execution time is optimized.

Although, as we stated earlier, the areas of task decomposition and allocation in distributed systems are well studied already. The CREL run-time environment differs with the application domains of the traditional scheduling/mapping problems in the following senses:

1. Information such as the computation and communication costs of the clusters is incomplete. The dynamic nature of the problems and the inability to estimate the costs of the run-time tasks makes it difficult to formulate the problem.

2. The structure of CREL system is cyclic, i.e., the execution of a CREL program is the repetitions of the basic execution cycle of match, conflict resolution, and act. Therefore, it is inappropriate to assume the traditional approaches based on acyclic computation graphs.

3. There are no suitable logical boundaries on the tasks in CREL systems for us to apply the traditional task assignment strategies. As one can observed from the static analysis results in Chapter 4, the clustering of a program does not necessarily represent a suitable partition because of the dynamic nature of the problems.

Because of these factors, we choose to focus on logical partitioning of the CREL run-time tasks, which is described in Chapter 5. For the run-time management, we adopt the simple concept of work pools of ready tasks and let processing elements access and execute these work pools without preset allocation strategies.

6.3 CREL Target System Architecture

The target machine we have chosen to implement the CREL system is a Sequent's Symmetry model multiple-processor with 16 processing elements(20Mhz i386) and 32MB of shared memory[28, 51]. The Symmetry uses a high speed common bus to link together all the processing units and the memory modules. The operating system is Sequent's proprietary version of the UNIX called DYNIX, which is a hybrid of both the 4.2 BSD and SYS V UNIX.

The characteristics of the Symmetry model, both in terms of the hardware architecture and the operating system are the following:

True Multiprocessor The Symmetry is a true MIMD machine with multiple PEs and multiple threads capabilities.

Tightly Coupled All PEs shared a common bus and have access to the entire shared memory address space.

Symmetric All PEs are identical and all processors have the capabilities of executing both the user and kernel codes.

Dynamic and Transparent Load Balancing The philosophy of Dynix OS is to fully utilize all the PEs by balancing the executable processes throughout the entire PE set. The load balancing mechanism is transparent to the users.

Programs use the concept of UNIX processes to take advantages of the available parallelism from multiple PEs, by *(micro)forking* multiple processes. A series of parallel programming constructs, such as forking processes, locking/unlocking, and barrier wait, etc., are added to the standard C library to facilitate the parallel program development process. In terms of the shared memory, the extension to the standard C language is the addition of a a new storage class called *shared*. The Sequent parallel C, therefore, contains three storage classes: *shared* global, private global, and private local. Variables declared as *shared* have only one copy among all processes. Private global variables, on the other hand, act exactly the same as global variables in standard C, with one copy created for every process. From CREL implementation viewpoint, although these characteristics simplify our design process, they also limit the control capabilities of CREL's run-time management.

6.4 CREL Process Management

Chapter 4 static analysis tell us which rules are mutually exclusive and and form clusters to allow asynchronous executions. Chapter 5 studies various sources of run-time parallelism within each cluster. From the conclusions of Section 6.2, we need to concentrate on the management of these run-time tasks of parallel match and multiple rule firing. To design an effective run-time manager, a global picture of the CREL system is summarized below.

Given a CREL program, there may exist multiple clusters in the system. From a cluster's viewpoint, the system executes the same basic code block as in the sequential cycles. The modifications necessary to explore parallelism are the

message passing mechanisms between clusters, and the management of run-time tasks described in Chapter 5. Specifically, each cluster, CL_i, executes the code of: [1]

[Match]

For each WME in the Remove_Queue[i], call Remove_Wme(WME), which may involves some join works if the WME appears as a -CE. There are also some cleanup operations on the data structures [2].

For each WME in the Add_Queue[i], do:

1. Perform constant testings, if necessary.
2. Update shadowed-alpha-memory lists.
3. Spawn join work unit(s) on the work queue.

[Conflict Resolution1]

1. Barrier waits on all spawned join work units.
2. Link the shadowed-alpha-memory back to the alpha memory.
3. For each instantiation in the conflict set(s), spawn RTC1 work units, whose responsibilities includes consistency checking of all other instantiations in the conflict set(s).

[Conflict Resolution2]

1. Barrier waits on all spawned RTC1 work units.
2. Clean up the data structures for run-time checking purpose.

---

[1] See Section 5.1 for detail descriptions on the key data structures and functions.
[2] There is no attempt in the CREL system to implement this step in parallel.

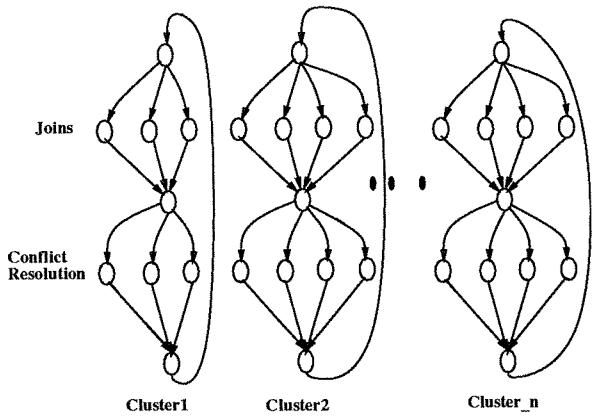

Figure 6.1: The CREL run-time global picture.

3. For each instantiation in the conflict set(s), spawn RTC2 work units, whose responsibilities includes sorting the conflict set(s) by the number of inconsistent instantiations.

[Act]

Barrier wait on all RTC2 work unit, clean up data structures, and perform the grouping algorithm (Figure 5.10). Execute the RHS actions of all instantiations selected for firing.

The global picture of a CREL system, then, can best be illustrated in Figure 6.1. A more detailed pseudo-code is listed in Figure 6.2. Both the token and intra-token parallelism were explained in Section 5.2 and 5.2.1. The parallelism implemented in the conflict resolution part was described in Section 5.3.

Because of the characteristics of CREL run-time systems, as described in Section 6.2, we choose to define the task units by their logical partitions, such as the join work units spawned by each mew WME, or the RTC1 checking functions. The run-time system employs a self-guided scheduling scheme to allow asynchronous execution and achieve load balancing without much overhead.

```
Match: {
        Receive_Msg();
        For each WME in Remove_Mbox[] + Remove_Queue do
          For each Amem_Entry in WMEs del_list[cl] do
                RemoveWme(WME);
        Broadcast(Remove_queue[]);
        For each WME in Add_Mbox[] + Add_Queue[] do {
          For each AMem[i] involved with WME do { /* compiler generated */
                Malloc(AMem_Entry);
                Build Shadow_Amem link;                                    10
                if (rule_active && local ) {
                  Wqueue[cl] =+ Join(WME, Amem_Entry_Index);
                  N = N +1 ;
      } } }
        Barrier_wait(N);
        For each Amem in cluster do
          merge Shadow_Amem[i] with Amem[i];
} /* MATCH */

Select: {                                                                  20
        For each inst I Cset[] do {
          Wqueue[cl] =+ CS_Check(I,cl); N = N +1;
        }
        Barrier_wait(N);
        Firable[] = CS_Resolve(); /* May be parallelized */
} /* Select */

Act: {
        For rhs_action in RHS of Firable do
          call rhs_action ; /* update Remove_Queue[] and Add_Queue[] */      30
} /* Act */
```

Figure 6.2: CREL Run-Time Pseudo-Code