On the Performance of the CREL System

Chin-Ming Kuo
Daniel P. Miranker
James C . Browne

Department of Computer Sciences
The University of Texas at Austin

TR-92-41 November 1992

4/

On the Performance of the CREL System*

Chin-Ming Kuo
Daniel P. Miranker
James C. Browne
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

July 13, 1991

Abstract

This paper presents the performance results of a comprehensive approach to the parallel execution of rule systems. It describes the semantics of a Concurrent Rule Execution Language, CREL, and the architecture of the system that compiles and executes CREL programs. The system has been designed to avoid runtime overhead by performing extensive compile time analysis and by parallelizing compilation. Static dependency analysis, based on serializability, coupled with a set of optimizing transforms, partitions the program into subsets, called clusters. Clusters execute independently of each other and communicate though asynchronous message passing. At runtime two additional sources of parallelism are exploited – run-time consistency checking allowing multiple rules to fire, and match-level parallelism.

The CREL system is implemented on a Sequent Symmetry shared-memory computer. This paper presents the results of a factorial experiment that isolates and evaluates each source of parallelism in the CREL system and each possible combination of those methods. The results suggest that multiple-rule-firing is the single most important source of parallelism in CREL programs and that the use of static dependency analysis based on serializability is instrumental to effectively exploit parallelism.

*This research is partially funded by a grant from the Office of Naval Research and a grant from Texas Instruments.

1 Introduction

The production system, or rule-based system paradigm, is a widely used method of building expert systems and artificial intelligence applications involving knowledge representation and knowledge base search. As the complexity and scope of expert system applications expand, performance requirements can impede the application of the technology. Parallel execution is an attractive approach in attempting to accelerate the execution of production system programs.

In this paper, a rule-based program is a set of rules of the form "if P then A" where P is a predicate on the current state of a database of facts called the working memory (WM) and A is an update transaction on the database. A predicate P is composed of a conjunction of condition elements (CEs). Sequential execution of the program proceeds by evaluating the predicates. A conflict set of rule instantiations is formed. An instantiation is a pair containing a rule name and an ordered set of facts satisfying, or matching, the rule. A single rule instantiation is selected, a process called conflict resolution, and the rule's actions executed. The *match, select, act* cycle is repeated until a fixed point is reached.

Most early efforts toward the parallel execution of rule based programs focused primarily on the parallel execution of these incremental match algorithms which had been reported to require over 90% of the execution. The early efforts could also be characterized as parallelizing the execution of the underlying interpretive mechanism.

In the mean-time other research into accelerating the execution of production-system programs has concentrated on the development of improved incremental matching algorithms, such as RETE and TREAT,[2, 16, 17, 24] and by using sophisticated compiling techniques. It has been demonstrated, in at least one system, OPS5.c, that better match algorithms and improved compilation can reduce the sequential execution time of rule-based programs by two orders of magnitude over traditional interpreted RETE match implementations.[15] We also note that the OPS5.c compiler often reduces the proportion of time spent in match to less than 50%. By Amadhl's law, the introduction of parallelism exclusively to the match phase of such optimized implementations will yield a maximum of two fold parallelism. Hence, it no longer beneficial to apply parallelism to a single aspect of the execution of a production-rule program.

Other researchers working on the parallel execution of production system programs have focused on firing multiple rules per production system cycle. Ishida and Stolfo described a system that fired commutative rules concurrently.[19, 8] This early work is one of the base for the work described in this paper. More recently, other researchers have applied serializability or other specializations of the Bernstein conditions to determine when rules may be fired in parallel. [7, 12, 9, 23, 25]

1.1 Approach

This paper describes the results of a comprehensive approach to the parallelization of rule-based programs. The CREL system exploits parallelism in and among all phases of the production system execution. CREL is built on the OPS5.c compiler. As a result of the high performance of the sequential component of this system, the theme of the CREL system has been to remove run-time overhead by performing extensive compile time analysis.

This work represents a fundamental change in the operational semantics of the OPS5 production-rule language. We define a new language, syntactically identical to OPS5, called CREL.[3] CREL is the acronym for the Concurrent Rule Execution Language. CREL and OPS5 differ only in their conflict resolution strategy. OPS5 associates a timestamp with each working memory element (WME) and selects a single instantiation from the conflict set based on the recency of the instantiation's timestamps. Each instantiation is fired only once. Conceptually, CREL also only fires a single rule instantiation per cycle and fires an instantiation only once per cycle, but the instantiation is selected nondeterministically. We note that in the original presentation of recency as a conflict resolution heuristic that the intent was to introduce a method that guided the system to the shortest path to a solution.[13] Recency was not intended as a correctness criteria. From our experience translating OPS5 programs to CREL, we conclude that the relaxed resolution strategy does not complicate rule-based programming.

A correct parallel execution of a CREL program as any serializable execution.[14] Serializability has its origins in database systems where it is applied to database transaction systems and serializability is enforced on ad-hoc, short-lived, transactions by using locking protocols. CREL programs represent a static collection of continually executing transactions. Thus, through compile time analysis, a CREL program can be partitioned into subsets, called clusters, such that any two rules in different clusters can fire concurrently without any runtime checking or locking, and without violating serializability. Clusters execute independently from each other and communicate through asynchronous message passing. CREL and its parallel execution semantics are defined in section 2.

Simple compile time analysis is insufficient to partition the system into an small uniform clusters. A set of optimizing transforms that increase the number of clusters and reduces their size is also introduced. The sequentiality introduced by context (or goal) element patterns, called *control variables*, prevents sequentially written production rule programs from displaying much parallelism among the rule clusters. The copy and constrain technique (C&C) [1] is introduced to further create additional clusters.[26] Clusters created by this technique substantially increase the available parallelism. The parallelizing compilation technique and optimizations are described in section 3.

Within a cluster, the free pattern variables make it impossible to predetermine precisely which rule instantiations can fire concurrently. At runtime, the CREL conflict resolution phase determines a collection of instantiations that can fire in parallel without violating serializability. Since conflict sets can be very large[17] and every element of the conflict set must be compared with every other element of the conflict set this is an expensive operation. The cost of this operation is reduced by determining a set of serializability predicates at compile time. The cost of this operation is further reduced by the static decomposition of the program into clusters. Low-level match parallelism through spawning separate tasks for each "join" is also implemented. See section 4 and 5.

All these forms of parallelism have been brought together in a single system. A factorial collection of experiments are performed on all possible combinations of each source of parallelism. These systems were run against 4 different CREL programs, translated from OPS5, and on a number of different size data sets.[11] The performance results are presented in section 7. These experiments incidate that multiple-rule firing is the single most important source of parallelism. Low-level match parallelism is not very beneficial. Multiplying the number of clusters using the C&C technique, when applicable, is very effective and introduces little runtime overhead. C&C improves performance by introducing additional parallelism at the match level and improves the constants when dynamically computing multiple rule firing. These experiments also make clean that the nondeterministic semantics of CREL is insufficient to overcome the intrinsic sequentiality of programs naively translated from OPS5.

2 The CREL Language and Execution Model

The execution of production systems can be viewed as a state-space searches, where the contents of the working memory represent the current state and the actions of a rule firing move the system from one state to another. Conflict resolution determines a particular search alternative. Conflict-resolution strategies, such as the OPS5 LEX and MEA strategies, uniquely determine the execution

[1] Independently named, the data reduction technique.[5]

path for a given production system program in a depth-first fashion based on the the *recency* of working memory time tags and the *specificity* of the rules.[3, 13]

Since one of our goals is to partition the rule system into clusters that execute independently from each other, with as little synchronization as possible, it becomes necessary to relax the recency constraints on the conflict resolution. Removing recency as a criteria introduces nondeterminism in the program executions. The CREL resolution strategy retains specificity. To ensure a correct execution we use the database serializability theory as the basis of the CREL execution model. A parallel execution of a set of database transactions is serializable if there exists a sequential execution of those transactions such that both the parallel and serial execution result the same final database state.[27] We will define a parallel execution of a CREL program as correct if and only if the concurrent execution of the actions of a set of rules are serializable. From this basic definition we are able to define and implement a parallelizing compiler and to precompile aspects of the run-time system, reducing their dynamic overhead.

2.1 CREL Execution Model

In CREL, serializability is defined within a single cycle. A complete execution path is the concatenation of the firing sequences of all cycles involved. Given a production system program P with N rules, P= $\{P_1, P_2, \ldots P_N\}$, a parallel firing E_j of P in cycle j is defined as the set of instantiations selected for firing in cycle j:

$$E_j = \{I_{j_1} \parallel I_{j_2} \parallel \cdots \parallel I_{j_m}\} \tag{1}$$

where m is the total number of instantiations in E_j, $\forall\, i, j$, I_{j_k} = kth instantiations in cycle j, and I_{j_k} is an instantiation of rule P_{j_k}. For each j,k, C_{j_k} is defined as the set of instantiations from rule P_{j_k} in cycle j. The conflict set in cycle j is $(\cup_{\forall k} C_{j_k})$.

Definition 1 Cycle Serializability
E_j *is cycle serializable if and only if there exists a serial execution path of* E_j, E_j', *such that* E_j' *produces the exact same result as* E_j. □

Definition 2 Execution Serializability
It follows from definition 1 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 cycle serializable.* □

Definition 3 Correctness of CREL programs
A CREL program is correct if and only if all eligible serial execution paths reach correct terminal states. □

Note that, other than the conflict resolution strategy, the execution behavior of a CREL program does not change from OPS5. In other words, the CREL system still executes the match select act cycles. From the programmer's point of view restrictions such as global execution cycles and single rule firing per cycle still exist. To construct a correct CREL program an OPS5 programmer need only restrict himself to rules such that any rule instantiation can be fired at random and still ensure a correct terminal state.

2.2 Dependency Analysis

To determine which rules can fire in parallel it is necessary to determine which rules *interfere* with each other. That is, which rules may remove WMEs that satisfy another rule, or due to negation, may add objects that invalidate some or all of a rule's instantiations. The CREL static dependency analysis is based on the bipartite dependency graph representation first proposed by Ishida and Stolfo.

A dependency graph G_m is defined as $G_m = (V, E)$ where the vertices represent either rules or sets of *pattern- equivalent* WMEs, illustrated by "circles"(◯) and "squares" (□) respectively. In [8] pattern- equivalent sets were determined by class name. In CREL, pattern- equivalent sets are determined by class name and the constant appearing in each condition element. The optimizing transforms further specialize the pattern-equivalent memory sets. Edges E in G_m represent the types of data dependency relations between WMEs and rules. An edge is drawn from a pattern-equivalent set of WMEs, W_i to rule P_i if W_i appears in the the LHS of rule P_i. An edge is labeled with a positive(negative) sign if W_i appears in P_i's positive(negative) condition elements. An edge is drawn from rule P_i to WMEs, W_i if W_i appears in the the RHS of rule P_i. An edge is labeled with a positive sign if W_i is in P_i's *make* action elements. An edge is labeled with a negative sign if W_i is in P_i's *modify* or *remove* action elements.

Based on the dependency graph we identify two types of conflicts where firing one rule invalidates instantiations from the other rules:

(A) $P_i \xleftarrow{+} W_m \xleftarrow{-} P_j$

Rule P_j is deleting or modifying an WME-node (W_m), which is also positively referenced by

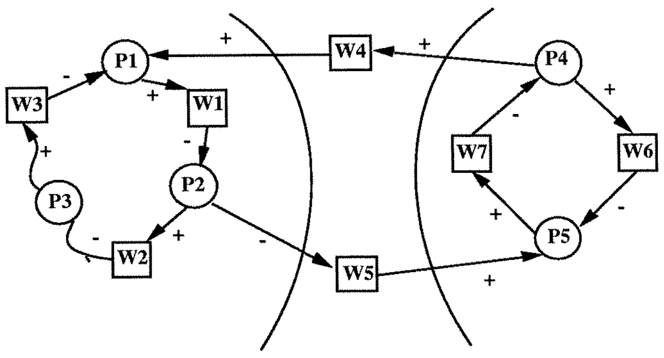


Figure 1: Dependency graph and mutual exclusions

rule P_i. In other words, the firing of P_j *may* deletes some entries of W_m, which may constitute parts of the current conflict set of P_i.

(B) $P_i \overset{-}{\leftarrow} W_m \overset{+}{\leftarrow} P_j$

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

Figure 1 presents an example of such a dependency graph and illustrates the case of interferences between two rules P_1 and P_2, where the curved arrows divide the dependency graph into two "mutual exclusion" sets.

A mutual exclusion dependency relation occurs when there are conflicts between rules in accessing the same data. The concept of a mutual exclusion set is developed where a mutual exclusion set is a set of rules for which we cannot statically determine if their instantiations can be fired concurrently. In other words, using the static analysis of the dependency graph, it cannot be determined if concurrently firing all rules in the same mutual exclusion set will violate serializability.

For example, the firing sequence of $\{P_4, P_5\}$ in Figure 1 is not a valid serial execution of parallel execution $\{P_4 \parallel P_5\}$. Because of the duality of the problem, only the cases of rule P_j interferes with rule P_i will be analyzed.

Definition 4 Interference

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

Given a pair of rules, a single interference relationship will note violate serializability. The two interference relations impose a total ordering on the serial execution sequence of the rules involved. If interference, such as ($P_i \overset{+}{\leftarrow} W_m \overset{-}{\leftarrow} P_j$), exists between two rules, P_i and P_j, a valid serial execution must execute P_i before P_j. The ordering imposed by interference relations are transitive.

2.2.1 Mutual Exclusion

Definition 5 Mutual Exclusion Set
A mutual exclusion set is set of rules connected by a cycle of interference relations.

Figure 1 illustrates examples of cycles in a dependency graph. Two of the cycles form mutual exclusion sets, while a third, P_1, P_2, P_5, and P_4 does not.

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

Proof: The proof is by induction. Given that there is a mutual exclusion set $\{P_1, \ldots, P_N\}$, we first prove the base case of N is 2, N being the size of the mutual exclusion set. Assuming the two conflicting instantiations of interferences are ($P_1 \overset{-}{\leftarrow} W_1 \overset{+}{\leftarrow} P_2$) and ($P_2 \overset{-}{\leftarrow} W_2 \overset{+}{\leftarrow} P_1$), 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 ensure 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 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 ($P_1 \overset{-}{\leftarrow} W_1 \overset{+}{\leftarrow} P_2$) implies $\triangle cs^1 \neq \emptyset$ and ($P_2 \overset{-}{\leftarrow} W_2 \overset{+}{\leftarrow} P_1$) 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 [2]

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

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.

For the induction step, the mutual exclusion set is size N, $\{P_1, \ldots, P_N\}$, and the dependency cycle is

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

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, first compute the mutual exclusion sets 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: Theorem 1 states that all parallel rule firings of instantiations from the same mutual exclusion set (confirming to the mutual exclusion constraints) are serializable. Since there exists no cycle of interferences between rules from different mutual exclusion sets, for the cases of parallel firings of rules across multiple 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 3, 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. □

[2]Notice the cycle count index.

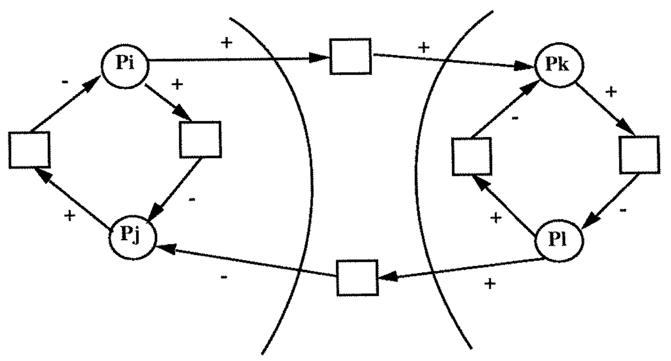


Figure 2: An example for asynchronous execution. (See the proof of Theorem 3 for definition of symbols.)

2.2.2 Clustering

Close examination of the proof of Theorem 1 and 2 reveals that there exists no sequencing constraint on rules that do not interfere with one another. Specifically, there is no requirement for synchronization between the execution cycles of two mutual exclusion sets, provided no cycle of interference exists. The following theorem formally states such an observation.

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 3, so we only need to prove that such parallel execution is always serializable. Without loss of generality, we assume the system contains two mutual exclusion sets, M_1 and M_2, as illustrated in Figure 2. The mutual exclusion constraints exists between the (P_i, P_i) and (P_k, 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 6 Cluster

A cluster is a set of rules where synchronization is needed for each execution cycle to ensure the *correctness of the execution. It follows from Theorem 3 that a cluster is the transitive closure of the mutual exclusion sets.*

The transitivity of the cluster relation guarantees that the partitions of rules resulted from clustering are disjoint. Consequently clusters of rules may execute independent match select act cycles without global synchronization.

3 Optimizing Transformations

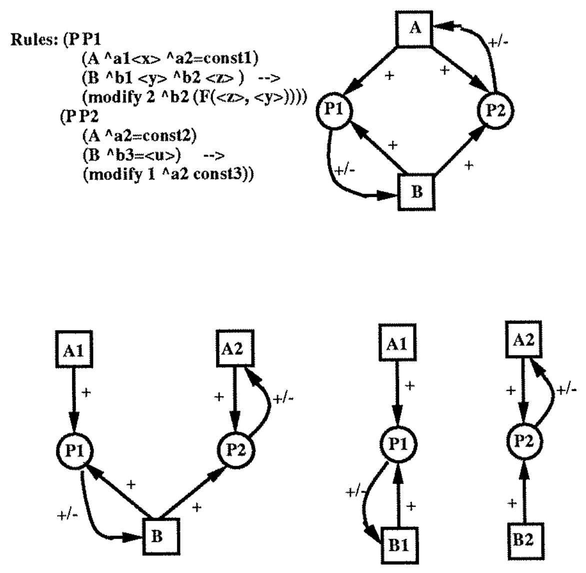
Static clustering for a number of OPS5 programs did not achieve satisfactory results. Because of the lack of both control and data structure constructs in production system languages, static clustering generally partitioned the programs into a single large cluster together with a number of clusters containing only 1 or 2 rules. See the column labeled "no optimization" in Table 1. Results are presented as the number of clusters and the maximum number of rules per cluster.

To improve the effectiveness of static clustering 4 optimizing transforms are introduced. The idea behind three of these transforms is to break interference relations between the rules by increasing the granularity of the pattern-equivalent sets of working memory in the dependency graph (i.e. by dividing the boxes into smaller boxes).

There are many results on the static dependency analysis of computation graphs in conventional languages for various parallel environments.[7, 4, 10] The fundamental idea behind these results can be used in optimizing the CREL dependency relations. Specifically, increasing the granularity of the data nodes in the dependency graphs can remove certain redundant dependencies. Recognizing the use of "control variables" in PS programming is another way of reducing the connectivity of the dependency graphs. By recognizing the equivalence between the LHS testings and the query forms in relational algebra, techniques used in database query decomposition can also act as types of optimizing transformations for the CREL static analysis.[18, 1] The goals of these optimizing transformations are two fold. The first is to increase the numbers of clusters in the system so that the concurrency of the system increases. The other goal is to reduce the complexities of run-time tasks in each cluster by reducing the sizes of clusters. We discuss in details each of the optimizing transformation in the sections follow.

3.1 Control Dependency

Because of the lack of procedural control structures in the PS languages, a common strategy called "secret-messages"[21] is used in PS programming to emulate the procedural controls. Such a



```
Rules: (P P1
          (A ^a1<x> ^a2=const1)
          (B ^b1 <y> ^b2 <z> )  -->
          (modify 2 ^b2 (F(<z>, <y>))))
       (P P2
          (A ^a2=const2)
          (B ^b3=<u>)    -->
          (modify 1 ^a2 const3))
```

Figure 3: Examples of refinements on WME nodes.

strategy uses a designated WME (usually called goal elements) as the control variable. At any given time during the execution, by assigning a different value to the control variable, the active rules are constrained to a group of specific rules. The rules responsible for transitions between different stages make use of the "specificity" resolution strategy to impose priorities on rules.

Such programming practice can be used to remove the interference relations between rules. Specifically, by identifying the control variables, rules can be partitioned into disjoint rule sets, with each set containing rules that test the same value of the control variable. All the interferences between between rules that belong to different rule sets are removed. We call this transformation "control variable smart". Results of analysis performed on the benchmark programs proves the technique of "control variable smart" significantly improves the quality of the CREL static analysis.

3.2 Propagating Constants and Disjoint Attribute Tests

One of the improvements that can be made to the dependency graph is to provide better classification of the data units used in the bipartite dependency graph. Increasing 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. In other words, CEs with the same class name but with different constant bindings should be treated as different "□" nodes. Another improvement one can make is to identify cases, where

CEs refer to the same class but contain disjoint attribute references. These too, can be treated as different nodes in the dependency graph. Figure 3 illustrates the gradual refinements on the WME nodes by first propagating constants on attribute a2 of class A, and by identifying disjoint attribute sets. In this figure, the upper graph is partitioned into subgraphs by recognizing disjoint references to different portions of working memory elements A and B.

3.3 Horizontal Partitioning With Constrained Copies

The equivalence between a LHS match work and the query forms in relational algebra is given in [11]. Techniques used in database query decomposition can also act as types of optimizing transformations for the CREL static analysis. The C&C technique shares the same basic idea with the "tuple Substitution" technique used in Ingres query optimization algorithms[28] by instantiating potential variables into disjoint hash buckets, thus replicating the query into independent sub-queries. An example of a rule P_i with two CEs is given:

```
(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 [3]. Rule P_i can be copied 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.

In addition to the benefits of LHS match optimization, constrained copying can also increase the number of clusters in a system by explicit partition the pattern-equivalent WME nodes into disjoint copies. Figure 4 gives the resulting graph by performing constrained copying and propagation(C&C) on a part of the Waltz program. The constrained copy algorithm may decompose a fragment of the dependency graph into a number of independently executable entities.

3.4 Static Analysis Results

We have implemented the software to generate dependency graph representations of rule systems, to perform mutual exclusion analysis and form rule clusters and to apply our optimizing transforms. The transforms are enumerated as follows

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

Two rules from OPS program "Waltz" are listed below:

```
(P one-one-out_P1
 (stage reduce-candidates)
 (junction ^junction-ID <X> ^line-ID-1 <l-ID>)
 (junction ^junction-ID {<Y> <> <X>} ^line-ID-1 <l-ID>)
 (labelling-candidate ^junction-ID <X> ^line-1 out)
 -(labelling-candidate ^junction-ID <Y> ^line-1 in)
 -->
 (remove 4))
```

```
(P two-two-minus_P2
 (stage reduce-candidates)
 (junction ^junction-ID <X> ^line-ID-2 <l-ID>)
 (junction ^junction-ID {<Y> <> <X>} ^line-ID-2 <l-ID>)
 (labelling-candidate ^junction-ID <X> ^line-2 -)
 -(labelling-candidate ^junction-ID <Y> ^line-2 -)
 -->
 (remove 4))
```

Part of the original dependency graph is:

If hash buckets on <junction-ID> is N, we can copy the cluster into N clusters:

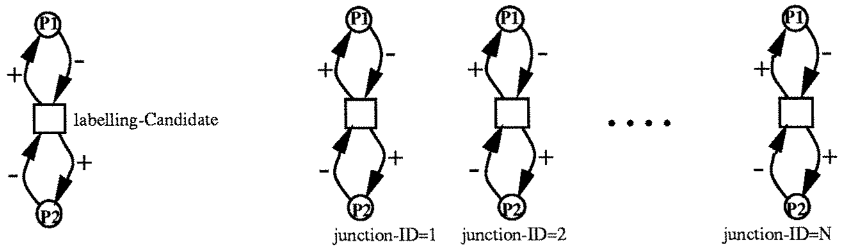


Figure 4: C&C algorithm and its example.

1. Propagating LHS constants into RHS.
2. Find out disjoint attributes among CEs.
3. Control Variable Smart. (CVS)
4. Constrained Copying with Propagation. (C&C)

In the table, the first column lists the benchmark program. The three entries with C&C are cases where C&C is applied to Life, and twice of Toruwaltz. The second column lists the total number of rules. The third to sixth columns list clustering results with various combination of optimization. To illustrate the concurrency as well as the density of the clusters, the results are expressed as pairs, (x,y), where x, y are the number of clusters and the maximum number of rules per cluster, respectively. For instance, Life-NR is partitioned from 10 rules to 3 clusters but one of these clusters contains 8 rules.

One can observe from the table that the performance of a particular optimization depends heavily on the nature of the program. For instance, Optimization (1) performs well in Tourney and Rubik, but not in Life, Judge, etc. Another observation is that combinations of (1) and (2) improve the connectivity of the systems slightly. This is because optimizations (1) and (2) can identify disjoint rule sets when there are many unbounded variables. The table shows results of applying C&C on Life with the hash size of 4(life.nr.4)[4] and on ToruWaltz with two different free variables and hash sizes of 3 and 4 each.

[4]We hashed a LHS variable into 4 buckets.

Program	Rules #	No Opt.	With Opt.			
			1	2	1+2	1+2+3
Life.NR	10	(3,8)	(3,8)	(3,8)	(3,8)	(8,2)
Life.NR.4	18	(3,16)	(3,16)	(3,16)	(3,16)	(14,2)
Tourney	16	(2,9)	(4,8)	(2,9)	(11,5)	(13,4)
Waltz	32	(3,30)	(5,28)	(3,30)	(5,28)	(11,18)
ToruWaltz	27	(2,26)	(2,26)	(2,26)	(4,24)	(4,24)
Toru.Waltz.3	63	(2,62)	(2,62)	(2,62)	(4,24)	(4,24)
Toru.Waltz.4	99	(3,85)	(3,85)	(3,85)	(7,24)	(7,24)
Rubik	66	(2,64)	(4,60)	(2,64)	(11,54)	(18,13)
Judge	245	(48,41)	(48,41)	(48,41)	(58,31)	(60,26)

Table 1: Results of Static Dependency Analysis

3.5 Dynamic Behaviors of Clusters

Static analysis does not reflect the number of clusters that will actually be active concurrently. To determine the dynamic parallelism, we developed a simple CREL execution system on a shared-memory Sequent Symmetry system. This system provides minimal support of run-time management and each cluster is mapped to a single processor.

Within each cluster only a single rule is selected for firing each cycle (This rule may be partitioned into multiple executions as described under Section 5, "Join level Match Parallelism"). Concurrency profiles were developed to illustrate the run-time behaviors. Figures 5 and 6 give the snapshots of the number of clusters active at any instant for the Life program, without optimization and with C&C applied. The X-coordinate is the run time of the system in milliseconds. The Y-coordinate indicates the number of active clusters with rule firings.

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 the PS programming style of using control variables. Since static analysis can not instantiate these free variables, the initial clustering result without optimization always contains a heavily connected cluster.

Optimizations 1 and 2 produce minimal improvement on all three benchmark systems. However, optimizations 1 and 2 are essential to make the C&C technique effective.

The use of the control variable in a sequential form severely limit the concurrency of the systems, as in Figure 5. C&C always provides good results, provided there are eligible free variables to partition. This can be observed in both the static result table and the concurrency profile in Figure 6. As this study of the dynamic behaviors of these systems demonstrates, there is a need to further

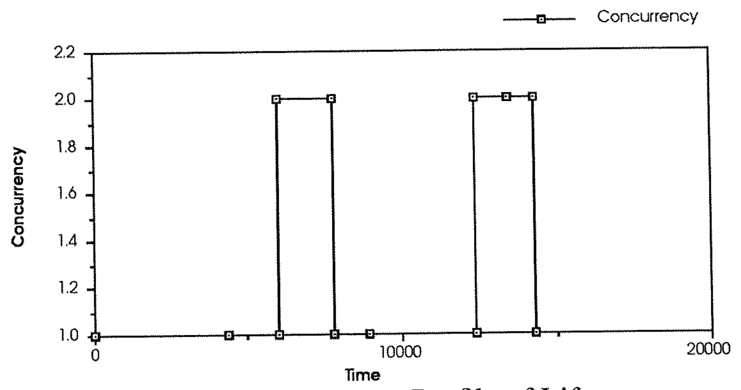


Figure 5: Concurrency Profile of Life

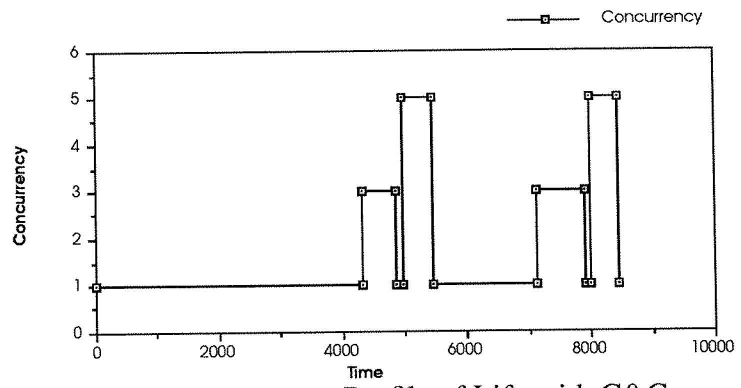


Figure 6: Concurrency Profile of Life with C&C

exploit run-time parallelism with clusters.

4 Run-Time Checking for MRF

Within a CREL cluster multiple rule instantiations can be fired. But to ensure serializability the values bound to the free pattern variables of the rule instantiations must be tested for interference. Hence, the distinction between static analysis for multiple rule firing and run-time checking is the ability to test variable bindings.

In order to allow MRF within a cluster, one needs to first check whether each pair of instantiations in the conflict set can fire simultaneously. This problem is named [RTC1]. Given the results of the above checking, one then needs to compute a subset of instantiations from the conflict set, that can be fired in the current cycle without violating the serializability requirement. This problem is named [RTC2]. The complexity of problems RTC1 and RTC2 is $O(N^2)$, where N is the number of instantiations in the conflict set. Since N can be very large, reducing the cost of these problems is a primary concern.

4.1 Problem RTC1

Given a conflict set we need to determine for each pair on instantiations, I_m and I_n, whether simultaneous firing of I_m and I_n violates serializability. This can be done by verifying if there is an interference relationship between the two instantiations despite the values bound to the pattern variables. Potential interference relations are easily determined from the dependency graph. The specific variables that must be compared are easily determined from the text of the interfering rules.

The worst-case size complexity of the conflict set is exponential in the number of CE in a rule. Although the average size of the conflict set is small and worst-case is rarely, if ever achieved, the effect of large polynomial terms do appear.[17] One concern is that the overhead of executing a large number, $O(N^2)$, of run-time interference tests could easily overwhelm the performance obtained by the optimized compilation of the match code.

To reduce the execution time of problem RTC1, The run-time interference check predicates are precompiled into C code, similar to that produced for matching by the OPS5.c compiler itself. The example in Figure 7 demonstrates the idea.

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
)
```

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

Figure 7: An Example of RTC1, where precompiling generates low overhead run time checking.

Interference between instantiations I_1 and I_2 exists because firing I_1 removes WME (B 1 2) and (B 1 2) is in I_2. In addition, firing I_2 creates a new WME, (C 2), which invalidates $LHS_1(I_1)$.

However, there exists no interference between I_1 and I_3. Simultaneous firing of I_1 and I_3 can not be allowed by the static dependency analysis, since there exist cycles between rule P1 and P2 through pattern-equivalent WME classes B and C and free variable $< n >$ and $< y >$.

Therefore, a combination of the static analysis and the observation that at run-time, it is the case that all the free variables are fully instantiated leads to the idea of precompiling run-time checking predicates based on the static analysis approach. Specifically, the differences between instantiation pairs (I_1, I_3) and (I_1, I_2) reside in the binding of variable $< n >$ and the second WME of class B. The precompile and optimized run-time checking predicate for rule P_1 and P_2 then can be expressed in the following pseudo code:

$$((y_1 + 1 \neq x_2) \wedge (x_1 \neq y_1)) \tag{2}$$

This example illustrates the basic idea of how CREL precompiles run-time checking predicates to reduce run time overheads. To summarize, to allow MRF within a cluster, we precompile and optimize the set of RTC checking predicates for all rule pairs in the cluster. At run time in the conflict resolution phase, all pair of instantiations are checked for interference by using these predicates.

4.2 Problem RTC2

Assume the size of the conflict set (CS) is N and we have run through the checking predicates on all pairs of instantiations. The results of the checking are stored in a matrix called RTC[N][N]. Problem RTC2 is to find a subset, CS′, from CS, such that firing all instantiations in CS′ guarantees serializability. In other words, there exists no interference relations between any pair of instantiations in CS′.

For this type of problems, the best known sequential algorithm to compute a optimal subset has complexity $O(N^2)$, N being the size of the conflict set.[20] To avoid excessive run-time overhead for cases of large N, a sub-optimal grouping algorithm of $O(N)$ complexity, illustrated in Figure 8, is used. The sorting step in the grouping algorithm, is performed in parallel, with one task responsible for sorting all instantiations from the same rule.

4.3 The Effects of Clustering on Run-Time Checking

The purpose of the run-time 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

```
Given IG = (V,E), where $V = \{ Ii \mid I_i \text{ in } CS \}$,
                 $E = \{ E_{ij} \mid RTC1_{mn}(I_i, J_j) = F \}$
                 $I_i$ is from rule $P_m$, $I_j$ is from rule $P_n$.

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

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

Figure 8: Grouping Algorithm for RTC2

the total number of cycles. Computing [RTC1] and [RTC2] introduces overheads that does not exist in the sequential execution cycles of OPS5. Efficiency, therefore, is the 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 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 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. This will be elaborated in Section 7.

5 Join-level Match Parallelism

In most production rule systems, testing and binding of variables is performed by accessing working memory through an index structure called an alpha-memory.[5] There is one alpha-memory per condition element.[6] Alpha-memories provide fast access to the subset of data that matches intracondition pattern constraints, such as constant tests. The process of testing and binding pattern variables is analogous to a database join operation. The basic step of the TREAT incremental

[5]Or AMem[] in our description of the algorithms.

[6]Identical condition elements can share a single alpha-memory.

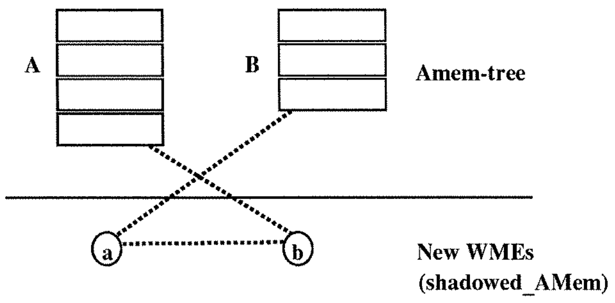


Figure 9: Example of join-level parallelism

match is to initiate a multiway join for each working memory update. Let B and C represent the alpha-memories of rule p2 in figure 7 and b is the new WME to be added to B. Then the conflict set is determined by the join of these two alpha-memories. The derivation of the conflict set before and after the updates(CS and CS') is shown below:

$$\begin{aligned}
&\mathrm{CS} = \mathrm{B} \bowtie \mathrm{C}\\
&\mathrm{B}' = \mathrm{B} + \mathrm{b}\\
&\mathrm{CS}' = \mathrm{B}' \bowtie \mathrm{C}\\
&\quad = (\mathrm{B} + \mathrm{b}) \bowtie \mathrm{C}\\
&\quad = (\mathrm{B} \bowtie \mathrm{C}) + (\mathrm{b} \bowtie \mathrm{C})\\
&\quad = \mathrm{CS} + (\mathrm{b} \bowtie \mathrm{C})
\end{aligned}$$

The second aspect of run-time parallelism in the CREL system is to introduce join-level match parallelism by performing concurrent multiway joins, one for each alpha-memory update. Despite the previous statement on the reduction of match time percentage it was conjectured that it is important to parallelize all aspects of production system execution to maximize overall speed-up. Join-level parallelism was introduced by altering the OPS5.c compiler to produce new parallel join code that does not require locking the alpha-memory structures.

5.1 Join Subtasks

To illustrate the organization of the join execution consider the example in Figure 9. In this figure, A, and B represent the initial alpha-memory structures. a and b represent the newly created WMEs, and the diagram indicates matching of a with B and A with b. The sequence of actions by the sequential TREAT algorithm on this example is given in(3):

$$(\mathrm{a} \bowtie \mathrm{B}),\ (\mathrm{A}' = \mathrm{A} + \mathrm{a}),\ (\mathrm{b} \bowtie \mathrm{A}'),\ (\mathrm{B}' = \mathrm{B} + \mathrm{b}) \tag{3}$$

In (3), multiple updates to the alpha-memory structures are processed by interleaving a update with a join operation. One possible solution is to execute the sequence in (3) by pipeline parallelism using locks on the alpha-memory structures – similar to the token-level parallelism exploited by Gupta.[6] Locking will impose unnecessary sequential constraints and was a noted bottleneck in Gupta's work. Alternatively, with the help of a compile-time transformation, we were able to design parallel joins without locking alpha-memory structures.

To avoid locking alpha-memory structure the structure must remain constant for the entire join execution. Therefore, an auxiliary data structure, called "shadowed-alpha-memory", is used to store all alpha-memory updates. The shadowed-alpha-memory structures establish a clear boundary between the "old" alpha-memory structures and the new ones. The join phase execution (3) can then be decomposed into the following parallel join tasks(4)

$$(a \bowtie B) \parallel (b \bowtie A) \parallel (a \bowtie b) \tag{4}$$

The term, $(a \bowtie b)$, in (4) is necessary to compute the same join results as in (3). The type of join between a 'new' alpha-memory cell (a) and an 'old' alpha-memory structure (B) is called a regular join. All join tasks must complete before the alpha-memory structures A and B can be updated, introducing a new source of barrier synchronization.

Although this appears simple and convenient the simple approach can lead to $(2^N - 1)$, where N is the number of CE in the rule. In addition to spawning N primary subtasks, one for each shadow-alpha-memory, a naive approach requires $(2^N - N - 1)$ subtasks to compute the missing crossproducts; far to many to be effectively handled in the run-time environment.

Using algebraic manipulation, the $(2^N - 1)$ terms can be restructured into N parallel joins. The structure of these parallel joins is as follows. the alpha-memory structures are represented as $A_1, A_2, \ldots, A_N$ and the shadowed-alpha-memory structures are represented as $a_1, a_2, \ldots, a_N$.

$$\begin{aligned}
& (A_1 \bowtie A_2 \bowtie A_3 \ldots \bowtie a_N) \\
\parallel\ & (A_1 \bowtie A_2 \bowtie A_3 \bowtie A_4 \ldots \bowtie a_{N-1} \bowtie (A_N + a_N)) \\
\parallel\ & \ldots \\
\parallel\ & (A_1 \bowtie A_2 \bowtie a_3 \bowtie (A_4 + a_4) \ldots \bowtie (A_N + a_N)) \\
\parallel\ & (A_1 \bowtie a_2 \bowtie (A_3 + a_3) \bowtie (A_4 + a_4) \ldots \bowtie (A_N + a_N)) \\
\parallel\ & (a_1 \bowtie (A_2 + a_2) \bowtie (A_3 + a_3) \bowtie (A_4 + a_4) \ldots \bowtie (A_N + a_N))
\end{aligned} \tag{5}$$

```
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;
    }
  }
End

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

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

Figure 10: Algorithm **PJOIN1**

Further, each of the N joins can be *seeded* by the change to alpha-memory,(a_i above), and composed with a fixed join structure. This seeded structure is consistent with the organization of the code generator in the OPS5.c compiler, which generates a distinct code segment for the join seeded by each CE.[15] CREL modifies the OPS5.c join code generator using the **PJOIN1** algorithm, making it possible to produce concurrent join code at compile time.[11]

Figure 10 lists the pseudo-code of the parallel join algorithm. Algorithm **PJOIN1** first updates the shadowed-alpha-memory structures. 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 subtask. Each join subtask 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. Assuming the CE index of the join seed is i, the fixed join pattern is such that the join seed only joins with all the old alpha-memory structures with CE index smaller than i, and then joins with both the old alpha-memory structures *as well as* the shadow-alpha-memory structures for the ones with CE index larger than i.

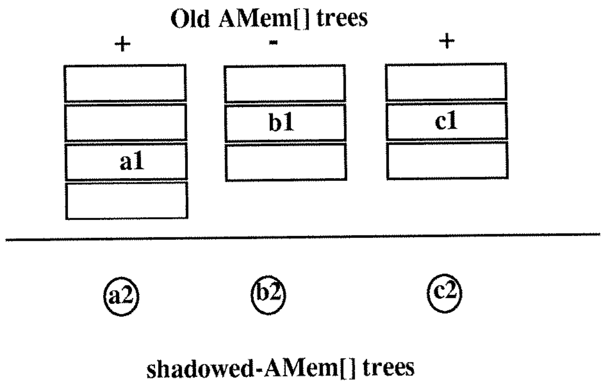


Figure 11: The effects of -CE.

5.2 Effects of Negated CEs

Negated CEs, (–CEs), pose an additional problem for join-level parallelism. 'Making' a WME that matches a –CE may invalidate some of the instantiations in the conflict set. This is not a problem for the sequential implementation of TREAT algorithm, since any invalidated instantiations are guaranteed to be present in the conflict set. This is not the case for the parallel join structure represented by **PJOIN1**. There is no sequencing constraint among the parallel join subtasks of positive or negated CEs. A join seeded from –CE may compute an instantiation that is to be invalidated that has not yet been calculated by a join seed from a positive CE.

An example is illustrated in Figure 11, where multiway joins requires a specific order of evaluation and all negated condition elements must appear at the end of a rule. This can be done through the compiler. 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, since the instantiation, (a_1, b_2, c_2), is a result from the join work of $(a_1 \bowtie b_1 \bowtie c_2)$, where c_2 is the join seed.

Based on the following observations:

1. At the beginning of the current match cycle, any instantiation containing WME entries exclusively from the "old" alpha-memory structures is guaranteed to be in the conflict set

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

3. Reordering CEs has no effect on the completeness of Algorithm **PJOIN1**.

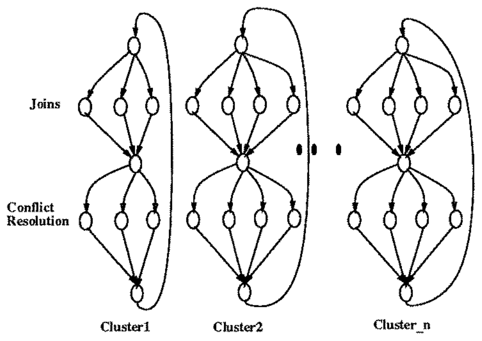


Figure 12: CREL run-time global picture.

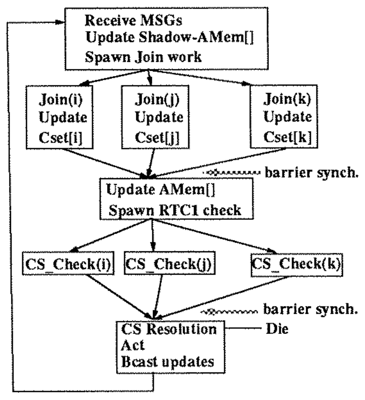


Figure 13: Cluster flow chart.

It is sufficient to mechanically rearrange the join order such that all negated CEs appear at the end of a rule. This was also performed by altering the OPS5.c compiler.

6 The CREL Run-Time Environment

Given a CREL program, there may exist multiple clusters in the system. Each cluster in the system executes the same basic code block as in the sequential production system cycles, as illustrated in Figure 12. Figure 13 gives the flow chart of the code structure within each cluster. Recall that within each cluster only one rule will typically execute in parallel. The decomposition of this rule by the join algorithms in Section 5 may lead to multiple executable parallel tasks.

Compile-time analysis clusters the systems and generates code for parallel join subtasks. The CREL run-time manager's responsibility is to effectively coordinate the execution of these tasks under the resource constraints, such that the total execution time of the CREL program is optimized. For shared-memory machines such as the Sequents, the resource constraint is captured by the task allocation to the processing elements. excessive overheads on the run-time management, the CREL run-time manager should provide little intervention on the program execution. The CREL run time manager should introduce as little overhead into program execution as is possible. Therefore, we designed a simple, self-guided process management scheme as follows. A shared work queue is allocated for each cluster in the system. Within each cluster, the individual subtask is responsible for maintaining a proper sequence of execution for the cluster tasks. At the beginning of a execution cycle, for example, a task called *mgr1* is responsibility to receive WME update messages from other clusters, process the WME updates, and spawn the join subtasks. To maintain the proper sequence of program execution, the *mgr1* also needs to append a task (*mgr2*) to the work queue. The function of *mgr2* is first to do a barrier wait on all join subtasks, and process alpha-memory updates, etc.

Each PE can freely access all the work queues and execute the task on a work queue. Mutual exclusive access to each work queue is ensured through locking. Each PE has a different access pattern in scanning all the work queues to avoid contention. Other implementation issues in CREL run-time management, such as modular code design to reduce run-time overhead, cluster communication, and memory management are discussed in details in [11].

7 Analysis of the Results

This section presents results from the performance measurements of the CREL system on a set of benchmark programs. To observe individual effects of the sources of parallelism, the CREL system is implemented so that factorial experiments can be performed.

Mea.	Parallel Match	Clustering	MRF
1	N	N	N
2	N	N	Y
3	N	Y	N
4	N	Y	Y
5	Y	N	N
6	Y	N	Y
7	Y	Y	N
8	Y	Y	Y

Table 2: CREL Performance Matrix.

The CREL performance matrices are organized by the key issues of the CREL system. The measurement plan is illustrated in Table 2. Specifically, effects from individual or combined sources

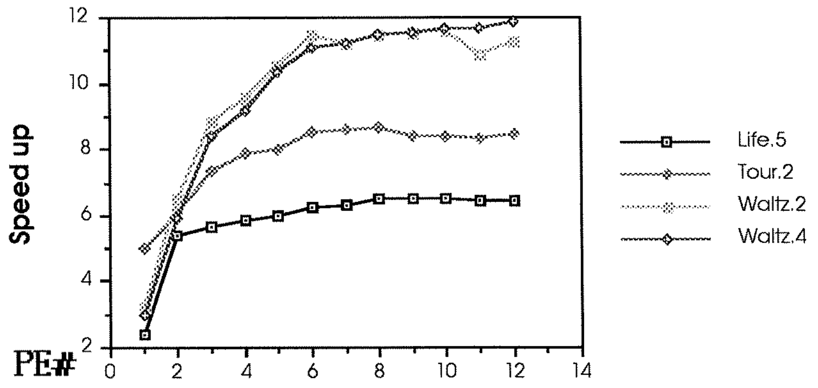


Figure 14: CREL overall speedups.

of parallelism can be measured by switching on or off these key features. The measurement # 6 in Table 2, for example, with N PE and multiple rule firing, without clustering, is to measure the combined effects of the parallel match multiple rule firing, but with no static clustering.

In addition to the total execution time, additional efforts were made to collect measurement data to facilitate further understanding of the system behavior. Measurements of such include the message passing overheads, the costs of run-time checking, etc.

8 Benchmark Programs

A set of benchmark programs wer chosen for the performance analysis of the CREL systems: Life, Waltz, and Tour-Waltz, and a revised version of the Tourney program. The Life is a data driven type of program. The Waltz implements the Waltz labeling algorithm. Toru-Waltz is a different implementation of the Waltz algorithm with more rule parallelism. Tourney is a search program to solve the tournament scheduling problem. The four benchmark programs were examined to ensure they are scalable in data sizes. The input data size and the number of PEs add additional dimensions to the spectrum of the CREL performance matrices.

8.1 Overall Speedups

This section discusses the results of the performance benchmarks. "Overall speedup" means the CREL system activate all the features given earlier. In other words, the comparison is made between the sequential execution of OPS5 systems and the CREL implementation with parallel match, static clustering/asynchronous execution, and multiple rule firing. Figure 14 plots the speedup ratios of the full fledged CREL system with C&C technique. The speedups obtained on these system (with

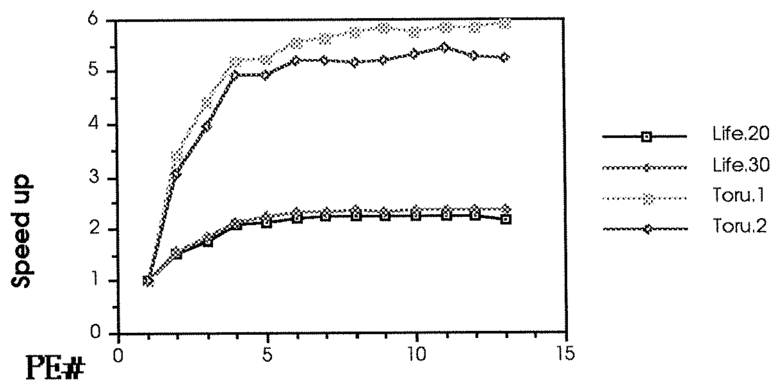


Figure 15: CREL speedups(a).

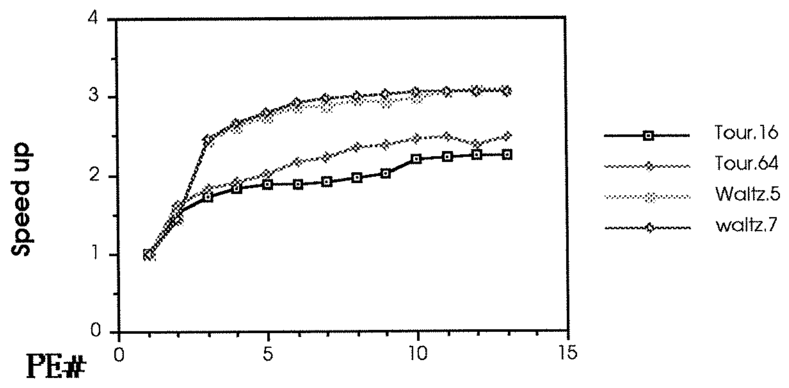


Figure 16: CREL speedups(b).

2 fold replications of C&C) range from 6 to 12. However, for some systems, the combined effects of CREL and C&C can result in arbitrary improvements.

Figures 15, 16 give the speedup rations of the CREL systems without using C&C, with multiple data set overlayed. The CREL system achieves speedups ranging from 2.5 to 6 on the the set of original benchmark programs. The number of PEs is limited to 12 due to the hardware limitation. The following sections discuss individual contributions from various sources of parallelism.

Through careful study of the system behaviors of the benchmark programs, it was determined that a majority of these benchmark programs are highly sequential by the control variable. The sequentiality is embedded in the algorithm design, and most of our approaches can not substantially increase the parallelism. We also show that by applying the C&C technique to replicate the rule sets in these systems, we can achieve significant performance improvements by reducing the sequential effects of the control variable. Such results further show that the traditional programming style used in these benchmark systems is the bottleneck in the CREL parallel execution model.

Another explanation for the limited performance increase is that the TREAT based sequential implementation of the OPS5.c compiler already achieved significant performance improvements when compared with the original RETE based systems. Because of this, some of the fundamental issues in parallelizing production systems have changed. Match, for example, no longer is the primary candidate for parallelization. Even the incremental match network is emerging as a bottleneck for the CREL systems because of its close-coupled data structures. These observations, however, do not conflict with the fundamental design philosophy of the CREL system. In other words, the CREL methods such as static clustering, parallel match, and run-time checking for multiple rule firing, can still be applied to newer generations of parallel productions systems.

8.2 Join-Level Match Parallelism

This section discusses the performance improvements obtained from the join-level parallel match. During the performance measurements, only join-level parallel match is activated and all other sources of the CREL run-time parallelism are turned off. Figure 17 plots the speedups from match parallelism, for each application with multiple data set overlayed. The results given in this figure confirm the previous statement on the match time percentage and the limitations of match parallelism. Table 3 lists the match time and the percentages of these benchmark programs.

To study to effectiveness of the CREL join-level parallelism, the actual match time is isolated from the total execution time and compared between the systems with and without match paral-

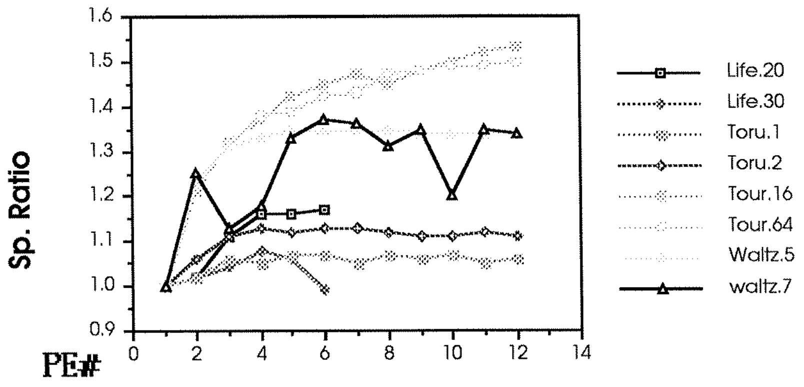


Figure 17: Match speedups.

	Life.20	Life.30	Toru.1	Toru.2	Tour.16	Tour.64	Waltz.5	Waltz.7
Time	103954	363323	80333	183149	25370	978182	262542	460102
Percent.	21.63	27.75	34.06	37.16	55.51	57.24	56.60	57.99

Table 3: Match time and its percentage.

lelism. Figure 18 and 19 illustrate the reductions in total match time in the systems as the number of PEs increases.

The limited success of the join-level parallelism can be attributed to the management of the alpha-memory structures. Specifically, the select codes in the CREL system – the part of the code that loops through WME changes and updates the corresponding alpha-memory structures – are executed sequentially to avoid locking overheads. From the statistics gathered, the sequential select codes take up from 6.5% to 10% of the total execution time. During the updates of the alpha-memory structures by these select codes, there is essentially no parallel join work available for the idle PEs to execute. The sequential select code, coupled with the use of control variable in the benchmark programs, severely reduces the overall system utilization in the match phase.

8.3 Clustering

This section discusses the effects of clustering on various aspects of the CREL run-time system, especially the effects of clustering on parallel match. Other issues related to clustering will be discussed later in the section of multiple rule firing.

The effects of clustering on the CREL system with parallel match but without run-time checking can be observed by comparing the actual run times of the same system, both with and without clustering. Figure 20 and 21 give the improvement ratios of the systems with clustering compared

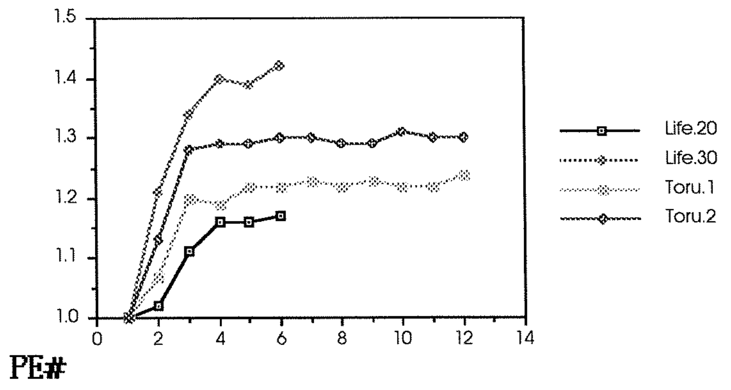


Figure 18: Match time reduction by join-level parallelism(a).

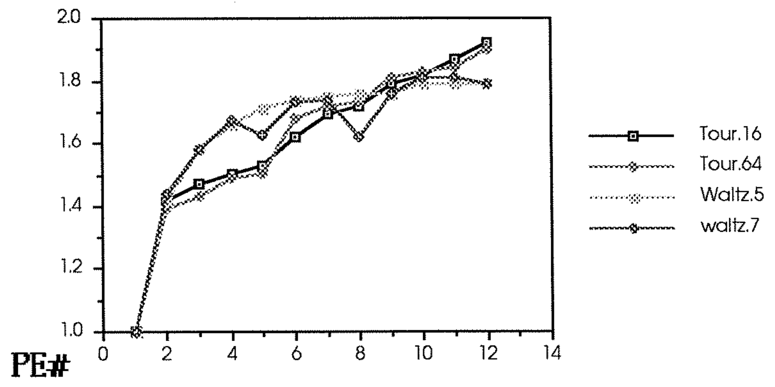


Figure 19: Match time reduction by join-level parallelism(b).

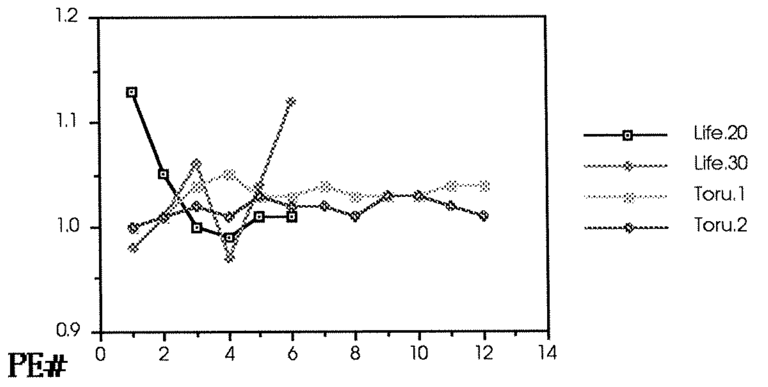


Figure 20: Match and clustering speedups(a).

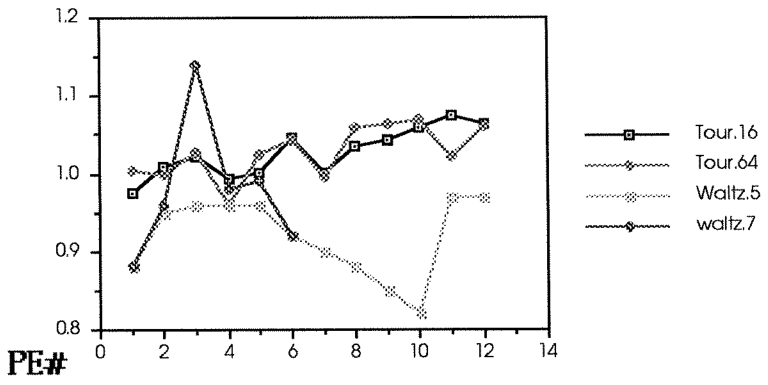


Figure 21: Match and clustering speedups(b).

to the ones without clustering.

The results from Figures 20 and 21 show that there is essentially no effects of static clustering on the overall system performance. This can be attributed to the sequential PS programming style, i.e., the sequential use of control variable. Specifically, although static clustering produce multiple clusters in a system, the actual concurrency of the clusters is limited by the sequential value of the control variable.

Multiple clusters and asynchronous execution introduce communication overhead between clusters. Communication overhead occur in both the sending and receiving ends. Specifically, the sender should ensure that WME changes will not be sent to "unrelated" clusters. "Unrelated" clusters mean those clusters that will not have alpha-memory cell changes due to the WME changes. It is therefore necessary for a sender to record a separate set of WME changes for every other cluster in the system. Once the messages are sent, the receiving end will copy the messages from the message queues and process these messages accordingly. Figure 22 and 23 illustrates the ratios of

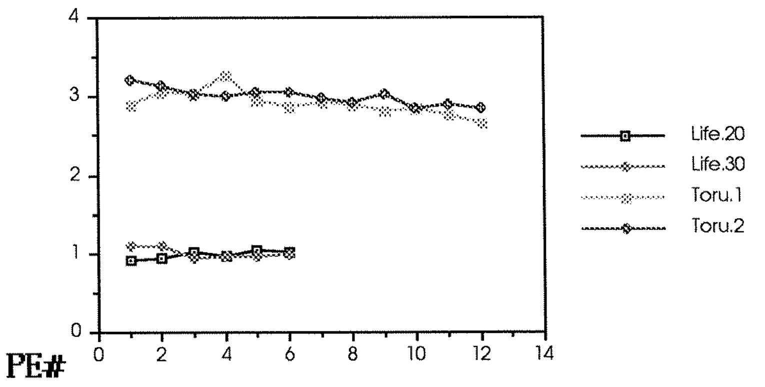


Figure 22: Clustering effects of MSG overheads(a).

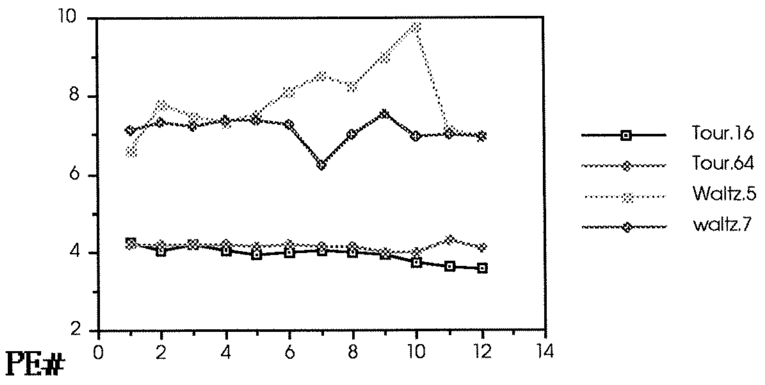


Figure 23: Clustering effects of MSG overheads(b).

message passing overheads compared with the ones with no clustering. Although one can observe an increase of message overhead through the clustering technique, the small percentage (from 4% to 9%) of the message overheads was offset by the reduction in the number of active rules through clustering.

8.4 Run-Time Checking and Multiple Rule Firing(MRF)

To observe the effect of MRF, see Table 4, which illustrates the performance gains from applying only the MRF technique to the CREL systems running on a single PE. Specifically, the first row in Table 4 shows the speedups of applying MRF to the benchmark systems with only one PE in the systems. As one can observe from these statistics, MRF contributes a major portion of the overall CREL speedups. To illustrate the effects of MRF from another viewpoint, Table 4 also lists the statistics of these systems in terms of the numbers of total cycles per execution, and number of firings per cycles. The second row in the table gives the total numbers of cycles for the single rule

Desp.	Life.20	Life.30	Toru.1	Toru.2	Waltz.1	Waltz.3	Waltz.5	Waltz.7
Cpu.	1.57	1.56	3.32	3.07	1.49	1.62	1.62	1.59
Cyl.(S)	3279	6703	1033	1561	160	2436	1611	2145
Cyl.(M)	856	878	11	11	85	1176	11	11
Fir.(M)	2876	6702	1031	1559	160	2436	1629	2177
S. Cyl.	845	1865	0	0	60	2100	0	0
Fir/Cyl	184	372	93	141	5	5	148	198
SD.	176	380	163	243	0	0	170	226

Table 4: Effects of MRF.

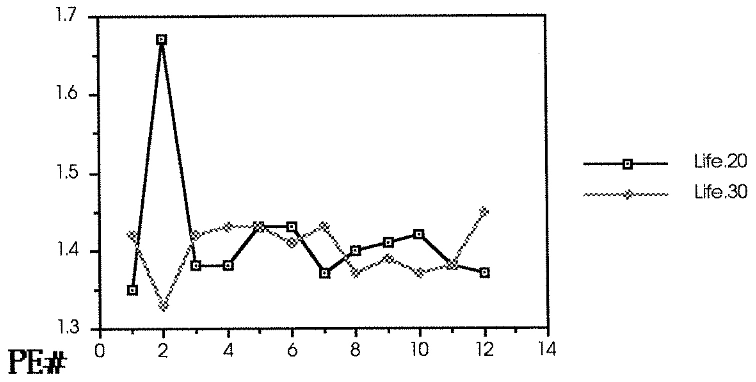


Figure 24: Comparisons of clustering effects on MRF(a).

firing cases. Thus they also represent the total numbers of rule firings in the systems. The third and fourth rows list the total numbers of cycles for the cases of MRF, and the number of rule firings, respectively. The fifth row lists the parts of the systems that are intrinsically sequential, and the sixth and seventh rows list the average numbers and the standard deviations of the rule firings per cycles.

With only multiple rule firing , it is possible that there are parts of a program which are intrinsically sequential and that no improvements can be made by MRF only. For the parts of the systems that do allow MRF, the apparent advantages of MRF over the single rule firing systems can be observed from these numbers.

As stated earlier, clustering significantly reduces the costs of run-time checking. Figure 24 to 25 plot the ratios of MRF costs of the ones without clustering, to the ones with clustering. The positive effects of clustering on MRF can be easily observed.

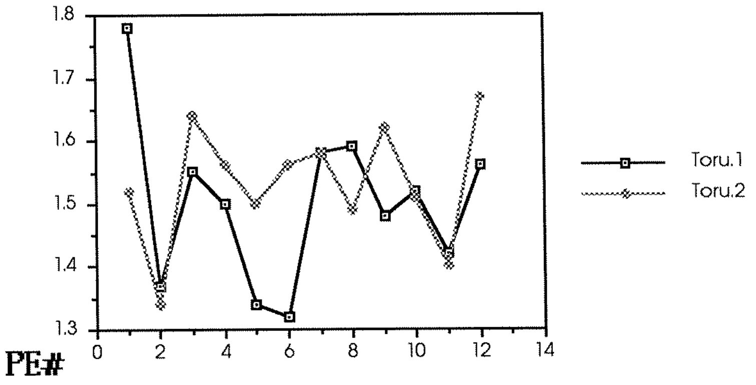


Figure 25: Comparisons of clustering effects on MRF(b).

8.5 Orthogonality Study between MRF and Match Parallelism

This section investigates the orthogonality between match parallelism and multiple rule firing. Such orthogonality implies the parallelism from multiple rule firings and parallel match are independent, and the combined speedup of these two techniques is the product of the individual speedups from each approach. One can observe, from the statistics gathered, that such orthogonality does not exist.

Figures 26 through 29 plot, for each application, the speedups from parallel match, the speedups from the combined effects of parallel match and MRF, and the ratios between them. One can observe that the ratios are almost flat across the span of a number of PEs and conclude there exists no orthogonality between parallel match and MRF. This is contradictory to Gupta's results. The reasons are the total amount of match work does not increase due to MRF, and part of the match work for the next cycle is shifted to the conflict resolution in the current cycle.

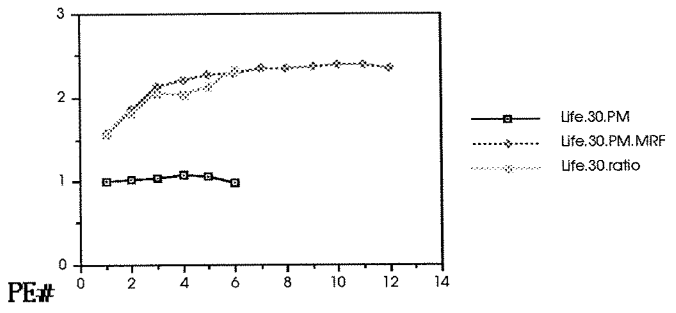


Figure 26: Orthogonality study of MRF and PM(1).

8.6 Constrained Copying

Results presented so far show that the control structures embedded in the benchmark programs are the main cause of the limited success in CREL systems. The best method to cope with this is the C&C technique. To demonstrate the effectiveness of the CREL approaches, this section discusses the results of code replication to increase parallelism.

Three benchmark programs are expanded and the same performance measurement procedures were performed on these replicated systems. Figures 30 and 31 illustrate the overall speedups of the three replicated systems, compared to the original systems. One can observe the substantial improvements on the replicated systems. Study of the execution traces show that the increased concurrency among clusters is the main cause of the performance improvements. Since the increased concurrency is a direct result of reducing the negative effects of the use of control variables, this experiment demonstrates the effectiveness of the static clustering and run-time checking methods.

To illustrate the effectiveness of the join-level match parallelism, the match times of the replicated systems were compared to the sequential ones. Figure 32 plots the match time ratios of the original systems to the replicated ones. These results once again illustrate that the limited improvements in the original systems are caused by the lack of large numbers of active clusters. In addition, the improvements in the match phase can also attributed to the reduction of the join complexity, which is the original goal of Constrained Copying.[22].

From the viewpoint of CREL programming, the results in this section shows that one has to program with parallelism in mind to take full advantages of the CREL environments. The first step toward proper CREL programming is not to rely on using sequential "control variables". In other words, even if control variables are used in the program developments, one should not program a system in such a way that the concurrency of the system is limited by the existence of a single control variable. Instead, multiple occurrences (WMEs) of the control variables or a more complex structure should be used to allow more parallelism that can be exploited by the CREL execution scheme.

9 Conclusions

The technical innovations of the CREL systems can be divided into the following five categories; the language and its execution model, static analysis and optimizing transformations, run-time checking for multiple rule firing, a parallel match based on the TREAT algorithm, and issues in the CREL run-time management.

The CREL language and its execution model eliminate the OPS5 recency resolution strategy and allow asynchronous execution of independent clusters, along with multiple rule firing within each cluster. The static analysis and transformations identify, at compile time, clusters of rules that can be executed independently and asynchronously. Through compile time generated run-time checking functions, the CREL system detects dynamic interference relations among instances from the same cluster and performs conflict resolution thus allowing multiple rule firing within each cluster. The CREL parallel match algorithm achieves match parallelism at the rule level while avoiding the need of fine-grained locks on the alpha-memory cells, and reduces run-time management overheads by generating a number of parallel join tasks invoked by the new WMEs through assistances from the CREL compiler. The CREL run-time management system ensures the partitioning of works and balances of the overall system loads, while maintaining minimum amounts of overheads.

A factorial measurement study on the CREL system was performed to determine the effects of various features of the system. The observations made in the result section 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 attempts to parallelize production systems has been shifted. 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 on the set of OPS5 benchmarks. This can be attributed primarily to the use of control variables in these 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 eliminates the OPS5 recency strategy. The clustering technique also reduces the overhead of run-time checking for MRF by generating optimized checking functions at compile-time, and by reducing the complexity of the task of run-time checking by an order of the size of clusters.

The most significant performance improvement in the CREL systems is derived from run-time checking to allow MRF in one cycle. The effectiveness of MRF is significant in the original benchmark programs, and is further enhanced for the cases of the systems with constrained copying. We observe that through MRF, the cycles of a system can be significant reduced. Since the execution cycles introduce synchronization overheads in every parallel system, the focus of attention in parallelizing production systems should be to reduce the total execution cycles, instead of reducing the cycle time.

Last but not least, is the observation that the technique of constrained copying helps in every aspect of the system. The results from the replicated systems support such an observation. The results from the experiments of constrained copying further support our belief that the bottleneck in the CREL system results from the use of control variable in a sequential form exhibited in the OPS5 benchmarks. Such an observation leads to the conclusion that to exploit parallelism in production systems, always program with parallelism in mind. Specifically, do not use control variable(s) in a pure sequential form.

References

[1] CUMMINGS, B. *Expert Data Base Systems: Proceedings of the First International Conference*, (edited by L. Kirchburg) , 1986,

[2] FORGY, C. Rete: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence 19* (September 1982), 17–37.

[3] FORGY, C. L. Ops5 user's manual. CMU-CS 81-135, Carnegie-Mellon University, Department of CS, 1981.

[4] GAJSKI, D., AND PEIR, J. Essential issues in multiprocessor systems. *IEEE Computer 18*, 6 (June 1985), 39–88.

[5] GANGULY S, S. A., AND S., T. A framework for the parallel evaluation of datalog queries. In *Proc. of the 1990 ACM SIGMOD Conference on management of Data* (1990), ACM.

[6] GUPTA, A. Parallelism in production systems. Tech. rep., Carnegie-Mellon University, Department of CS, 1986. Ph.D. Thesis.

[7] HWANG, K., AND BRIGGS, F. *Computer Architectures and Parallel Processing*. McGraw-Hill, Inc., San Francisco, 1984.

[8] ISHIDA, T., AND STOLFO, S. J. Simultaneous firing of production rules on tree structured machines. *ICPP* (84).

[9] J.G. SCHMOLZE, S. G. A Parallel Asynchronous Distributed Production System. In *Proceedings of the 1990 National Conference on Artificial Intelligence* (July 1990), AAAI, pp. 65–71.

[10] KIM, S., AND BROWNE, J. Scheduling of parallel computations on mimd multiprocessor architectures. In *ICPP*, 1988.

[11] KUO, C.-M. Parallel execution of production systems. Tech. Report, University of Texas, Dept. of CS., 1991. Ph.D. Thesis.

[12] KUO, S., MOLDOVAN, D., AND CHA, S. Control in Production Systems with Multiple Rule Firings. In *Proceedings of the IEEE International Conference on Parallel Processing* (1990), vol. II, IEEE, pp. 243–2246.

[13] MCDERMOTT, J., AND FORGY, C. *Production System Conflict Resolution Strategies.* Academic Press, 1978.

[14] MIRANKER, D., KUO, C., AND BROWNE, J. Parallelizing Compilation of Rule-Based Programs. In *Proceedings of the IEEE International Conference on Parallel Processing* (1990), vol. II, IEEE, pp. 247–251.

[15] MIRANKER, D., AND LOFASO, B. J. The organization and performance of a treat based production system compiler. *IEEE Trans. on Knowledge and Data Engineering* ((to appear)).

[16] MIRANKER, D. P. Treat: A better match algorithm for ai production systems. AI TR 87-58, UT-AI, UT Austin, Jul 1987.

[17] MIRANKER, D. P., BRANT, D., LOFASO, B., AND GADBOIS, D. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence* (July 1990), AAAI, pp. 685–692.

[18] MORGENSTERN, M. The role of constraints in data bases, expert systems, and knowledge representation. In *Proceedings of the First International Workshop on Expert Data Base Systems* (Kiawah Island, SC, October, 1984) pp. 207-223.

[19] NILSSON, N. *Principles of Artificial Intelligence.* SRI International.

[20] PAPADIMITRIOU, C. H., AND STEIGLITZ, K. *Combinatorial Optimization, Algorithm and Complexity.* Prentice-Hall, Inc., 1982.

[21] PASIK, A. J. A methodology for programming production systems and its implementations on parallelism. Tech. rep., Columbia University, Department of CS, 1988.

[22] PASIK, A. J., AND STOLFO, S. Improving production system performance on parallel architectures by creating constrainted copies of rules. Ineternal, 1987.

[23] RASCHID, L., SELLIS, T., AND LIN, C. Exploiting Concurrency in a DBMS Implementation for Production Systems. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems* (1988).

[24] SELLIS, T. K. Multiple-query optimization. *ACM Trans. on Database Systems 13*, 1 (1988).

[25] SRIVASTAVA, J., HWANG, K.-W., AND TAN, J. S. Parallelism in Database Production Systems. In *Proceeding Sixth International Conference on Data Engineering* (1990), IEEE Computer Society Press, pp. 121–128.

[26] STOLFO, S., MIRANKER, D., AND MILLS, R. A Simple Preprocessing Scheme to Extract and Load Balance Implicit Parallelism in the Concurrent Match of Production Rules. In *Proceedings of the AFIPS Symposium on Fifth Generation Computing* (1985), AFIPS.

[27] ULLMAN, J. D. *Principles of database Systems.* Addison-Wesley Publishing, 1983.

[28] WONG, E., AND YOUSSEFI, K. Decomposition – a strategy for query processing. *ACM Trans. on DataBase Systems 1*, 3 (Sep 1976).

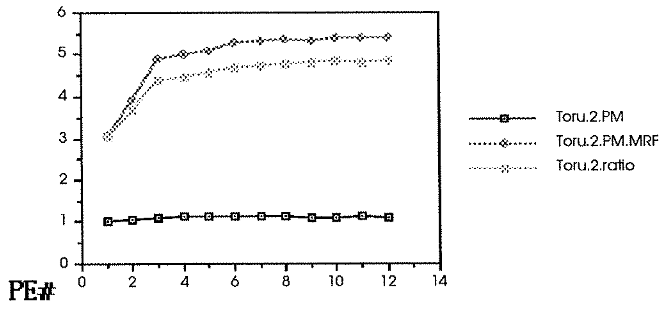


Figure 27: Orthogonality study of MRF and PM(2).

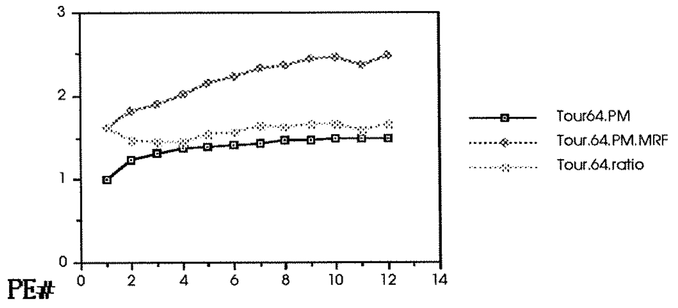


Figure 28: Orthogonality study of MRF and PM(3).

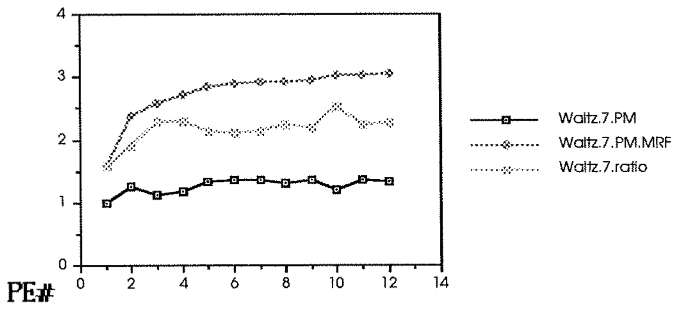


Figure 29: Orthogonality study of MRF and PM(4).

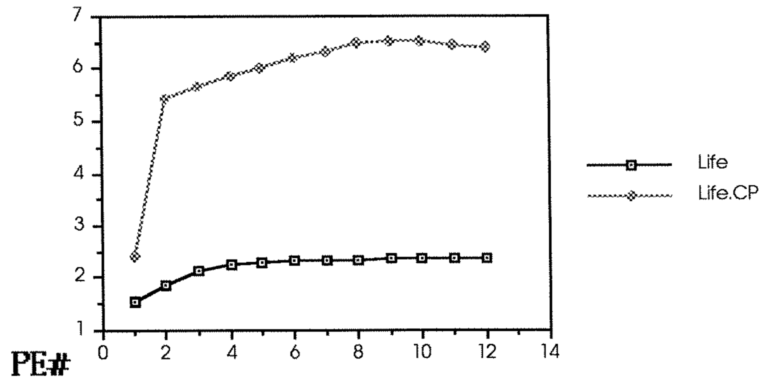


Figure 30: Speedups from C&C technique(a).

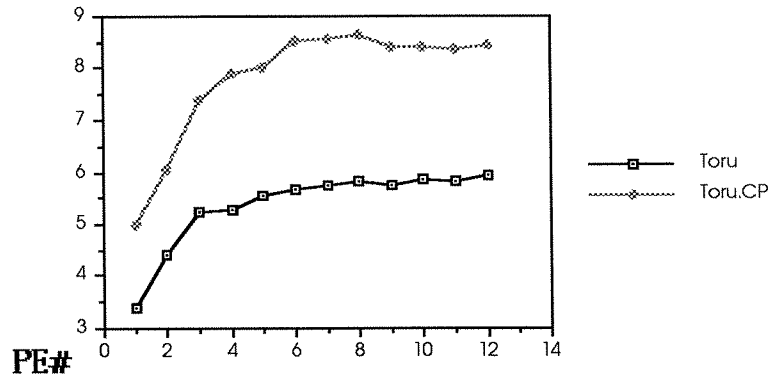


Figure 31: Speedups from C&C technique(b).

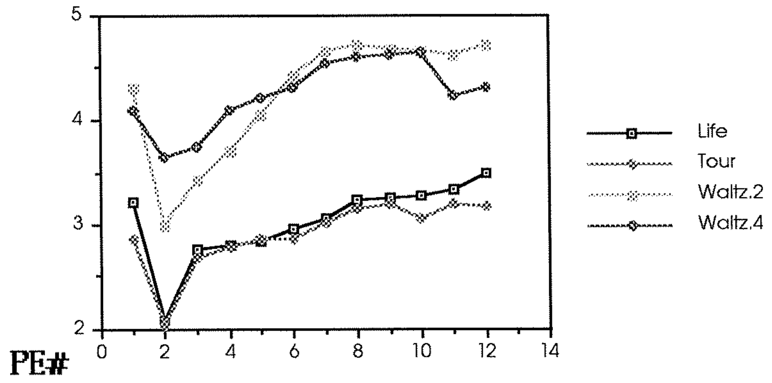


Figure 32: Comparisons of match time between the replicated and the originals.