PARALLELIZING TRANSFORMATIONS FOR A CONCURRENT RULE EXECUTION LANGUAGE*

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

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-89-30 October 1989

* This research was supported in part by the Office of Naval Research under contract N00014-86-K-0763 and by a grant from Texas Instruments.

Parallelizing Transformations for A Concurrent Rule Execution Language

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

October 25, 1989

Abstract

Most work on parallelizing forward-chaining production system programs may be described as parallelizing sequential production system interpreters. We are now studying an approach that parallelizes the entire execution of a production system program. We first modify the semantics of the OPS5 production system language into a rule language suitable for parallel execution. We then define a compilation method that partitions rule systems into disjoint subsets that execute asynchronously with respect to each other and which communicate through asynchronous message passing. Within each subset rules may be fired in parallel. The approach is similar to the methods used in parallelizing compilers for block structured languages and is founded on the formalisms developed for assuring the correct operation of concurrent database systems. Our primary results to date involve the definition of the syntax and semantics of a parallel production system language and the development of an ensemble of optimizing transforms. [1]

1 Introduction

In general, a *production system* is defined by a set of rules, or *productions*, that form the *production memory* together with a database of current assertions, called the *working memory* (WM). Each production has two parts, the *left-hand side* (LHS) and the *right-hand side*, (RHS). The LHS contains a predicate over *pattern elements* that are matched against the working memory. Pattern elements may be negated. The RHS contains directives that update the working memory by adding or removing facts and directives that invoke external side effects, such as reading or writing an I/O channel.

[1]This research was supported in part by the Office of Naval Research under contract N00014-86-K-0763 and by a grant from Texas Instruments.

In operation, a production system interpreter repeatedly executes the following cycle of operations:

1. Match. For each rule, compare the LHS against the current WM. Each subset of WM elements satisfying a rule's LHS is called an *instantiation*. All instantiations are enumerated to form the *conflict set*.
2. Select. From the conflict set, choose a subset of instantiations according to some predefined criteria. In practice only a single instantiation is selected.
3. Act. Execute the actions in the RHS of the rules indicated by the selected instantiations.

It has been previously reported that over 90% of the execution time of a production system program is spent in the match phase. [5] Thus, most efforts towards improving the performance of production system environments, for both sequential and parallel computers, have focused on improving the speed of the match portion of the cycle.[10]

A highly optimized sequential OPS5 compiler, called ops5c, has changed the characteristics of the problem in parallel production system execution. The new compiler is based on the TREAT matching algorithm.[11] The compiler incorporates new techniques for compiling rules into in-line code. We have discovered that by compiling the rules and using the newer match algorithm much more successful optimization strategies may be brought to bare than has been possible in earlier work. Preliminary results demonstrate programs running 50 to 200 times faster than interpreted production system environments and significant performance advantages over other production system compilers.[8] Of greater significance is that detailed analysis of the new compiler reveals that our compilation methods have reduced the proportion of time in match to considerably less than 90%. All but the most match intensive programs compiled by our system spend less than 50% of their time in match. Since most of the previous parallelizing efforts have dealt with only the match phase we can conclude from Amdahl's law that these approaches alone will not demonstrate significant speed-up over the programs produced by the new compiler.

1.1 Parallelism and CREL

The OPS5 execution cycle stipulates that all of the rule instantiations must be enumerated and a single rule selected for firing. In a parallel environment selecting and firing a single rule instantiation each cycle forms a double-headed bottleneck. It is contradictory in a parallel environment to select a single anything to execute. Further, as defined, the select strategy represents a barrier synchronization point. Barrier synchronization forces those processors that finish their workload quickly to lay idle while waiting the processor with the longest task to complete.

To achieve significant improvements in the parallel execution of production system programs we must simultaneously exploit all sources of parallelism. An effective system must demonstrate parallelism among each of the phases as well as within each phase. Our new approach is to fire rules as soon as they become satisfied, parallelizing the act phases and breaking the synchronization forced by the select phase. Removing synchronization introduces nondeterminism into the

program. The semantics of current production-system languages are derived from their sequential execution models and exploit synchronization heavily. It is precisely these sequential execution semantics that have limited the success in this area. As part of our effort to exploit all possible sources of parallelism we have specified a new concurrent production rule language, CREL. The syntax of CREL is identical to OPS5.[4] The semantics of CREL, described in Section 2.3 and 2.3, are derived from a combination of the theories of serializability, that underlies the correct execution of concurrent transactions on databases and the Unity language, a seminal declarative parallel-programming model. [2, 3] CREL programs that run correctly in a sequential environment are guaranteed to run correctly in a parallel environment. The basic idea is to guarantee that the result of the parallel execution of the system agrees with some serial execution of the same system and that all serial executions reach a correct fixed point. Pragmatically we have dropped the recency strategy of the OPS5 selection strategy but have retained specificity. The semantic constraints due to negation disallow a pure data-driven approach that allows any rule to be fired as soon its predicate has become satisfied. We say those CREL rules that cannot be fired in parallel are *mutually exclusive.*

A primary problem vexing researchers in this area is that the size of the fundamental tasks that can be executed in parallel is sufficiently small that if dynamic methods are used to the determine which of the tasks may operate in parallel the methods must be extremely simple or else run-time overhead becomes a dominant component of the execution of the system. Thus we are developing techniques that may be applied statically at compile time to eliminate, and if not, to dramatically reduce the dynamic overhead required to determine mutual exclusion.

In organization our approach is similar to the methodology employed in optimizing and vectorizing compilers.[1] As in these approaches, we compile the production-rule program into a dependency graph representation, described in Section 2.2. We use a graph representation, originated by Ishida and Stolfo[6], to represent the dependency relations and interactions between rules and working memory. Since any satisfied rule may fire at any time the graphs do not represent dataflow dependencies as do the graph representations used in vectorizing Fortran compilers and thus do not form a strict ordering. We will give more detailed descriptions later.

We can extract statically from the rule-dependency graphs whether the side-effects of any rule from a set of firable rules will interfere with the satisfaction of any of the other rules in that set. Those rules that may interfere with each other are *mutually exclusive.*[9] We have developed compiler algorithms that determine independent sets of rules, called clusters, such that a rule in one cluster is not mutually exclusive with any rule in any other cluster. Mutually exclusive rules are connected by a cycle in the dependency graph that contains labeled edges of different sign. A cluster is formed by calculating the closure over all such cycles over all rules in the cluster. By construction, clusters operate asynchronously with respect to each other with no overhead. Pairs of clusters may implicitly synchronize by passing messages. The rules within each cluster must synchronize. The techniques for firing rules in parallel in a synchronized environment may be applied within a cluster.

In our initial analysis we discovered that the dependency graphs are densely connected. Straight forward application of the mutual exclusion analysis does not reveal an substantial number of independent clusters. In response, we have developed a set of optimizing transformations, de-

scribed in Section 3 that break the mutual exclusion dependencies and thus increase the available parallelism. These transformations involve the recognition of special rule forms that introduce semantic information into the analysis, the propagation of constants and the horizontal partitioning of the working memory.

2 Dependency analysis

2.1 Definitions

Before we give the formal definitions of terms such as Mutual Exclusion and Synchronization Set, a brief description of the notations and the definition of serializability is given.

Definition 1 Serializability and related notations
Given a production system program $\mathcal{P}$ with N rules, $\mathcal{P}$= $\{P_1, P_2, \ldots P_N\}$, define a parallel firing E_x of $\mathcal{P}$ in cycle x *as the set of instantiations selected for firing in cycle* x*:*

$E_x = \{I_{x_1} \parallel I_{x_2} \parallel \ldots \parallel I_{x_m}\}$
where m is the total number of instantiations in E_x,
$\forall\, i, x, I_{x_k} = kth$ *instantiations in cycle* x, *and*
I_{x_k} *is an instantiation of rule* P_{x_k}.

For each x,k, we further define CS_{x_k} as the set of instantiations from rule P_{x_k} in cycle x. *It can be shown that $\cup_{\forall k} CS_{x_k}$ = the Conflict Set in cycle* x. *E_x is serializable if and only if there exists a serial execution path of E_x, E'_x, such that E'_x produces the exact same result as E_x and E'_x is a permutation of E_x.* □

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

Definition 2 *A CREL program is* **correct** *if and only if all eligible serial execution paths reach correct terminal states.* □

In order identify serializability, or the lack of it, among parallel rule frings, we use the following dependency graph to facilitate the analysis.

2.2 Bipartite Data Dependency Graph

The bipartite data dependency graph adopted from Ishida and Stolfo[7] can be used to express the interactions among rules and working memory elements. A mutual exclusion dependency relation occurs when there are conflicts between rules in accessing the same data. A mutual exclusion dependency graph G_m is defined as $G_m = (V, E)$ where

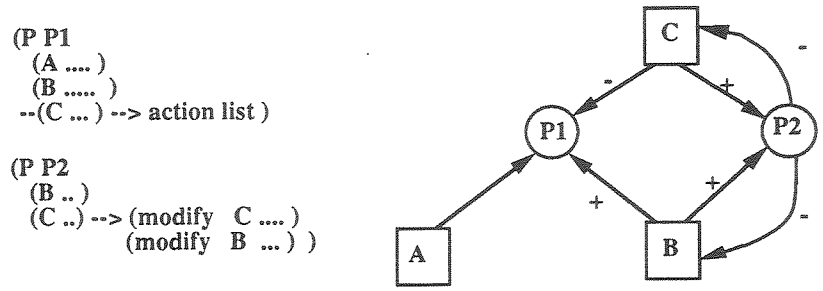

Figure 1: A bipartite data dependency graph example.

Nodes V can be classified into rule–nodes and object–nodes as:

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

Objects
A working memory object is a unit of data, referenced or updated by production rules. A set of *pattern equivalent* object nodes are represented by "square" (□) symbols in graph G_m.

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

(+/–) referenced
A type of edges from object–nodes to rule–nodes. A **referenced** edge is drawn from a working memory object O_i to a rule P_i if working memory object O_i appears in the the LHS of rule P_i. The edge carries a positive(negative) sign if O_i appears in P_i's positive(negative) condition elements.

(+/–) changed
A type of edges from rule–nodes to object–nodes. A **changed** edge is drawn from a rule P_i to a working memory object O_i if the working memory object O_i appears in the the RHS of rule 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** action elements.

Figure 1 gives an example of how to construct such a bipartite data dependency graph, where rule P_1 has a positive reference to data object W_1 (class A) and data object W_2 (class B). P_1 also has a negative reference to data object W_3 (class C). Rule P_2, on the other hand, has a positive reference and a negative change to object W_2 and a positive reference to object W_3. Figure 1 also illustrates cases of interferences between two rules P_1 and P_2.

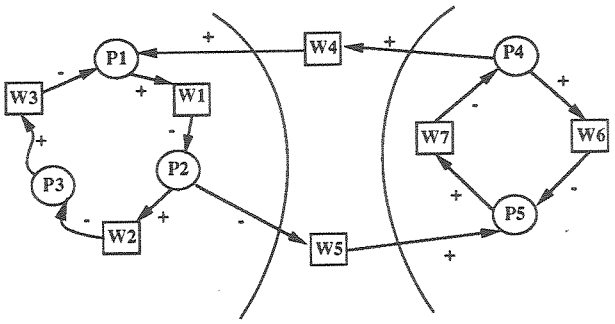

Figure 2: Example of Mutual Exclusions

2.3 Mutual Exclusion Set

A mutual exclusion set is defined to be a set of rules that cannot be statically determined to be executable 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 of multiple rule firing. A synchronization set of a rule is the closure of the rule over the mutual exclusion sets. The following two cases identify the possible situations where firing of one rule invalidates instantiations from the other rules (using the bipartite data dependency graph representation):

(A) $P_1 \stackrel{+}{\leftarrow} W_1 \stackrel{-}{\leftarrow} P_2$
Rule P_2 is deleting or modifying an object (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 in turn is part of the current conflict set of P_1.

(B) $P_1 \stackrel{-}{\leftarrow} W_1 \stackrel{+}{\leftarrow} P_2$
Rule P_2 is making an object (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 in turn *may* invalidate some instances of P_1 conflict set due to negative reference.

Definition 3 *Cases (A) and (B) described above are defined as two types of* **interference** *between rules P_1 and P_2, meaning 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 one 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.

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

The following algorithm can be used to compute all mutual exclusion sets in a production system:

Algorithm 1 *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 due to interferences.* □

Figure 2 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.

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

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 instances 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 from rule P_1 and P_2 respectively. To insure serializability without run–time checking, a parallel firing of any pair of instantiations,$\{I_1, I_2\}$, from $\{CS_0^1, CS_0^2\}$ should be serializable.

Let A_1 and A_2 be the RHSs of rule P_1 and P_2, and let conflict set CS_i represent the state of the system at cycle i. The effect of individually firing of I_1,I_2 is

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

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

where $\triangle cs^1$ and $\triangle cs^2$ are the updates to the conflict set from the firings of the other rule. 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 state of the system, after parallel firing of $\{I_1$,$I_2\}$, is

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

On the other hand, for serial firing of $\{I_1, I_2\}$, the changes of state variable is[2]

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

[2]Notice again the cycle count index!

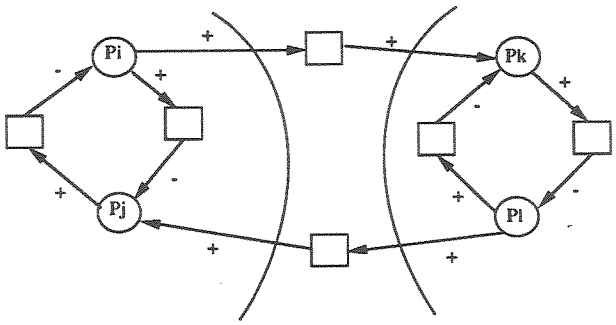

Figure 3: The example used in Theorem 3.

Again without run–time checking, $\triangle cs^2 \neq \emptyset$ due to 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 goes to the cases of $\{I_2, I_1\}$, thus concludes the proof of N=2.

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

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

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

Theorem 1 establishes the basis for parallel rule firings of production system programs. To correctly execute multiple rule firing, we have to first compute the mutual exclusion sets using Algorithm 1, 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 the same mutual exclusion set confirming the mutual exclusion constraints and are serializable. As for parallel firings of rules across multiple mutual exclusion sets, since there exists no cycle with conflicting interference between rules from different mutual exclusion sets, any arbitrary interleaving of these rules is a valid serial execution. Thus, any parallel execution confirming the execution scheme is serializable.

From Definition 2, any serial paths 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. □

2.4 Global vs. Local Synchronization

Given parallel execution, If we carefully examine the process of finding a valid serial execution path in that execution, there exists no sequencing constraint on rules that do not interfere with one another. Thus, we can relax the requirement of all rules be synchronized globally. Among rules interfering with each other, however, synchronization is still required to insure the correctness of parallel execution.

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[3], we assume the system contains two mutual exclusion sets, M_1 and M_2, as illustrated in Figure 3. The mutual exclusion constraints exists between P_i P_i and P_k P_l pairs.
Any parallel execution without global synchronization between M_1 and M_2 can be expressed as a regular expression

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

meaning the execution path can be arbitrary occurrences of one rule 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, not to mention the number of occurrences. □

We define a Synchronization Set as a set of rules where global synchronization is needed to insure serializability. From the proof of 3, it can be shown that a synchronization set is the maximum cycle among mutual exclusion sets where all synchronization is needed. Also note that the class relation of Synchronization Set is transitive. This guarantees the property that the partitions of rules by Synchronization Set are mutually exclusive, which is crucial to our mapping and load balancing 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 observation can also help the scheduling/mapping problem by condensing the computation graph into clusters of mutual exclusion sets where synchronization boundaries meet with rule partitions boundaries.

3 Optimizing Transformations

3.1 Control Variables

In writing production system programs, a common strategy called "secret-messaging" [13] is used to emulate the block structures in conventional languages. Such a strategy uses a designated class,

[3]This will also be true for cases of more than two synchronization sets.

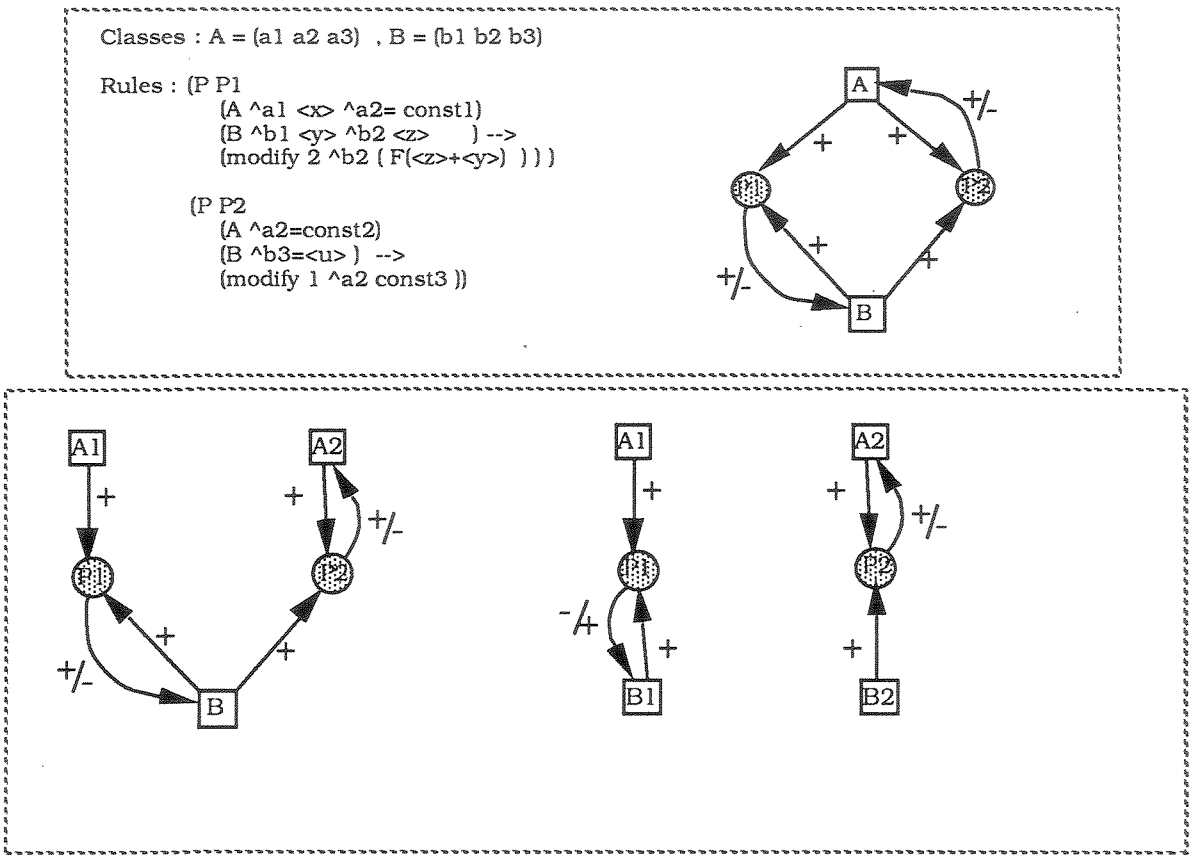

Figure 4: An Example Tranformation by Constant Propagation and Detecting Disjoint Attribute Sets

(usually named **goal**), to constrain the range of active rules to a certain set so that different stages of problem solving are partitioned into different sets of rules. In general programs that use secret messages have only one such message in existence at a time, thus forcing disjunctive sets of rules to be active at any one time. When we can identify classes that represent secret messages we can eliminate those graph edges that represent dependencies between disjoint subsets.[4] We call this transform *control-variable smart* (CVS).

3.2 Constant Propagation and Hash Partitioning

Ishida and Stolfo's earlier work on forming dependency graphs of OPS5 systems used only the record/class type to form pattern equivalent sets of working memory elements, represented by squares. However, additional constants in the patterns may be used to create finer sets of working memory with which to develop the dependency graphs and thus reduce the connectivity of the graph. We have identified several transformations that introduce additional constants and thus divide the square nodes in the graph. The first is *constant propagation*. In OPS5 the RHS modify command may specify a subset set of the attributes of a working memory element matched

[4]Currently we use pragmas to determine the presence of goal elements.

on the LHS. Those constants appearing in the LHS and not modified by the RHS may be safely propagated to the square representing the newly modified working memory element. Second, if condition elements access disjoint attributes of the working memory we can represent those working memory elements as disjoint.

Figure 4 illustrates the partitioning of a dependency graph as a result of constant propagation and detection of disjoint attribute sets.

When additional constants can not be determined through constant propagation the pattern equivalent sets of working memory may be partitioned using hash functions. This technique, called copy-and-constrain, has already been used to increase the discernible parallelism in production system programs.[14, 12, 15] The basic idea is to find a convenient variable in the LHS of the rule and to hash its domain into a number of disjoint sets. The representation of the rules that access the partition working memory set are replicated once for each partition. If the variable is selected correctly the rule copies may result in additional synchronization sets.

4 Preliminary Results

4.1 Benchmarks

We have completed 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

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

In the table, the first column lists the benchmark program. The three entries with CCP are cases where techniques of CCP is applied to LIFE-NR (life-nr-4) and TORUWALTZ (toru3 and toru4). The second column lists the total number of rules in a system. The 3rd column belongs to dependency analysis without tracing control variable. The results are expressed as pairs, (x,y), where x is the number of clusters and y is the maximum number of rules per cluster. The number of clusters in a system gives us a sense of the parallelism in the system and the max. number of rules in a cluster can usually show whether the system is heavily concentrated in some dominant clusters. The 4th through 7th columns are cases where various combinations of transformations are taken into consideration. For instance, without control variable, the dependency analysis on Rubik (No. 8) generates 11 clusters, while the largest cluster contains 54 rules. With control variable, the same Rubik program can be partitioned into 18 clusters, while the largest cluster contains only 13 rules.

Compile–Time Dependency Analysis (# Clusters, Max # Rules per Cluster						
Program	# of rules	No Optimization	Optimization			
			1	2	1+2	1+2+3
Life-NR	10	(3,8)	(3,8)	(3,8)	(3,8)	(8,2)
Life-NR-4 (with CCPx4)	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)
Toru-Waltz	27	(2,26)	(2,26)	(2,26)	(4,24)	(4,24)
Toru-Waltz-3 (with CCPx3)	63	(2,62)	(2,62)	(2,62)	(4,24)	(4,24)
Toru-Waltz-4 (with CCPx4)	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)

As we can see from the table, the performance of a particular optimization depends heavily on the nature of the problem itself, as well as the programming style involved. For instance, Optimization (1) performs well in cases like Tourney and Rubik, but not in Life, Judge, etc. The second observation is that combinations of (1) and (2) improve the connectivity of the systems slightly. We believe the reason is that optimizations (1) and (2) can identify disjoint rule subsets when there are many unbounded variables. Notice that the combination of (1), (2), and (3) can usually produce vast improvements in graph connectivity.

In terms of CCP results, the Table shows results of applying CCP on LIFE with the hash size of 4(life-nr-4), on ToruWaltz with two different free variables with hash sizes of 3 and 4 each. To decide the criteria of choosing a good variable to perform CCP is difficult and we are looking into this area.

Note that we did not present these preliminary results in terms of "speedup" or "degree of parallelism" because the computation cost of a rule is not included in the static analysis, thus it can not be justified that we can derive some real speedup numbers from the static analysis. What this table does show is that by combining various techniques discussed in this paper, we can have a balanced computation graph for load–time task assignment and asynchronous execution among different clusters. Any imbalance between clusters, be it in a single rule cluster or in the cluster with max. number of rules, will be taken into consideration during load–time task assignment, where computation and communication costs will be associated with clusters. Run–time load balancing will also reduce such imbalance.

5 Future Work

Implementation of the compiled CREL system with asynchronous parallel execution on shared memory multi-processor systems (Sequent Symmetry) is currently underway. Future work includes behavior analysis of the CREL system, load–time cluster scheduling, and run–time load balancing.

References

[1] Allen and Kennedy. Automatic translation of fortran programs to vector form. *ACM TOPLAS*, 9(4), October 1987.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

[4] C.L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie Mellon University, July 1981.

[5] A. Gupta. Measurements on production systems. Technical report, Carnegie-Mellon University, 1984.

[6] T. Ishida and S. J. Stolfo. Toward the Parallel Execution of Rules in Production Systems Programs. In *Proceedings of the IEEE International Conference on Parallel Processing*, pages 568–575. IEEE, 1985.

[7] Toru Ishida and Salvatore J. Stolfo. Simultaneous firing of production rules on tree structured machines. *ICPP*, 84. Perform dependency analysis and concludes with synchronization set for multiple firings

[8] B. J. Lofaso Jr. On Join Optimization for an OPS5 Compiler. Master's thesis, Unviersity of Texas at Austin, 1988.

[9] H.F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, 1986.

[10] Russell C. Mills. An Algorthimic Taxonomy of Production System Machines. Technical Report CUCS-340-88, Columbia University, Computer Science Department, April 1988.

[11] Daniel P. Miranker. TREAT:A Better Match Algorithm for AI Production Systems. In *Proceedings of the National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, August 1987.

[12] A. Pasik and S.J. Stolfo. Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules. Technical report, Columbia University, 1987.

[13] Alexander J. Pasik. A methodology for programming production systems and its implementations on parallelism. Technical report, Columbia University, Department of CS, 1988. Ph.D. Thesis Proposal.

[14] S.J. Stolfo, D.P. Miranker, and R. Mills. 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*. AFIPS, 1985.

[15] Eugene Wong and K. Youssefi. Decomposition – a strategy for query processing. *ACM Trans. on DataBase Systems*, 1(3), Sep 1976. methodology of Tuple Substitution !!