A Source-to-Source Transformation for Increasing Rule-Based System Parallelism

Alexander J. Pasik

***Abstract*— Rule-based systems have been hypothesized to contain only minimal parallelism. However, techniques to extract more parallelism from existing systems are being investigated. Among these methods, it is desirable to find those which balance the work being performed in parallel evenly among the rules, while decreasing the amount of work being performed sequentially in each cycle. The automatic transformation of *creating constrained copies of culprit rules* accomplishes both of the above goals.**

Rule-based systems are plagued by occasional rules which slow down the entire execution. These culprit rules require much more processing than others, causing other processors to idle while they continue to match. By creating constrained copies of culprit rules and distributing them to their own processors, more parallelism is achieved, as evidenced by increased speed up. This effect is shown to be specific to rule-based systems with certain characteristics. These characteristics are identified as being common among an important class of rule-based systems: expert database systems.

I. INTRODUCTION

Parallelism in rule-based systems has been studied in order to improve performance. Early work has indicated that there is only a limited amount of parallelism in rule-based systems which has been explained as being due to high temporal redundancy (little work available to be performed during each cycle) and poor load balance among the rules being matched [7], [8]. Specifically three points are made in order to explain the deficiency of simply distributing rules to multiple processors:

1. Few changes to working memory occur in each cycle.
2. Few rules are affected per working memory change.
3. There is high variability in match time among these affected rules.

Performance of rule-based systems, it is therefore contended, cannot be substantially improved by parallel processing. By directly addressing the load balancing issue, however, the source-to-source transformation of *creating constrained copies of culprit rules* [12], [14] provides a mechanism for the effective distribution of computation on a massively parallel architecture. *Culprit rules* are those which require substantially more computation time due to their condition elements requiring comparisons to many more working memory elements than other rules. These rules are copied, with each of the copies constrained to match only a subset of the working memory matched by the original rule. The constrained copies are distributed to different processors and execution time is improved.

II. BACKGROUND

Rule-based systems are divided into three parts: *rule memory, working memory,* and an *inference engine* (or *interpreter*). The rule memory contains a knowledge base of rules that are matched against the assertions in working memory. The inference engine performs this match, selects one satisfied rule instantiation to fire, and executes the actions specified. The execution of the selected rule alters working memory so that on the next cycle, new rules will be satisfied. This is called the *recognize-act cycle*. Execution continues until no rules are satisfied or the system is explicitly halted [4].

In order to determine rule satisfaction, the preconditions of each rule are matched against the working memory elements. The match can be broken down into two parts. First, the intracondition tests correspond to a relational selection on the working memory elements. Then, the intercondition tests are join operations on the relations which were selected [20].

The problem of efficiency in the match phase of rule-based system execution led to the development of the Rete algorithm [2], [3], the TREAT algorithm [11], and several related optimizations [1], [15], [16], [17]. The Rete algorithm uses a dataflow network compiled from the preconditions of all the rules. Newly added or removed working memory elements are processed through this pattern-matching network resulting in activations at the leaf nodes of new instantiations for the conflict set. Figure 1 shows a rule and its corresponding Rete network fragment. As working memory elements are asserted into or removed from the system, they are added as tokens to the top nodes of the network. When a token enters a selection test node (such as **type = connect** or **class = goal**), the specified attribute of the token is tested. If the test succeeds, the token is passed on to the next node. When a token reaches a select memory node, it is stored there indicating that the token successfully matched all intraelement tests. New tokens in a select memory are joined with tokens in another memory node if the two nodes flow into a join node. The successful joins form tokens comprised of multiple working memory elements and the process continues until a token reaches a leaf node. At this time the token will contain the ordered tuple of working memory elements corresponding to an instantiation of the rule which the leaf node represents.

TREAT is based on Rete, but differs in that less state is saved. In particular, join memories are not saved, but are recomputed when needed. This additional computation seems to imply that TREAT performs more poorly than Rete, but analysis of its benefits explains the empirical results of performance improvements [11].

Two detailed studies of the parallelization of rule-based systems are presented in the theses of Gupta [8] and Miranker [11]. The studies result in different recommendations, solutions, and directions for future research in the area. According to Gupta's thesis, rule-based system parallelism is limited to 30-fold performance improvements. In addition to deriving this by projecting from a set of empirical studies, a hardware solution was proposed based on the results. Gupta approached the problem of parallelism in rule-based systems by first examining rule parallelism, then node parallelism (nodes in the Rete network), and finally intranode parallelism. Gupta concludes that only two-fold speed improvements can be obtained by using rule parallelism. In order to overcome this limitation, Gupta proposes that the parallelism be exploited at a finer grain. However, communication costs increase using these mechanisms, contributing to the 30-fold limit. The additional communication required by the fine-grain parallelism dictates shared memory architectures.

Rule parallelism allows for distributed memory architectures; matching each rule separately results in the addition or deletion of instantiations. The system need only synchronize after the entire match and then communicate the results to the conflict resolution mechanism. The distributed memory approach towards the parallel execution of rule-based systems was investigated in the *DADO* project [19]. *DADO* is a massively parallel, distributed memory, binary tree machine. *DADO* accelerates rule-based system execution as backend pattern matching processor.

1. At compile time, precondition patterns are loaded into *DADO* according to some algorithm-specific distribution.

2. Postconditions are stored in the host machine.

3. Initial working memory is broadcast to all processing elements, each processor retaining only the relevant portion.

4. The match is performed in parallel by any algorithm selected.

5. Conflict resolution occurs quickly using simple specialized hardware to compare values in each processor identifying the minimum in one instruction cycle. The selected instantiation is reported to the host.

6. The working memory changes specified by the postcondition actions of the selected rule are broadcast to all *DADO* processing elements.

7. The cycle of steps 4 through 6 is repeated until no rules are satisfied or an explicit halt occurs.

Several match algorithms have been proposed and implemented for rule-based system execution on this architecture [18]. Rule parallelism can be achieved by using the *full distribution* method. Rules are distributed to the available processors according to some partitioning scheme (*e.g.,* round robin). Each processor thus contains a set of rules and performs the match locally. When working memory changes are broadcast, only relevant elements are kept by each processor (*i.e.,* those elements which partially match some local condition element). Changes in the local instantiations are reported during the parallel conflict resolution stage. Directly before conflict resolution there is a synchronization point; all processors much finish their local matching before conflict resolution takes place.

Although other algorithms have been proposed, the simplest approach to parallel match is full distribution, using any existing efficient pattern match algorithm in each processor. Thus, the set of tests for each rule is performed simultaneously. For a given cycle, the changes to working memory are processed by each rule resulting in a revised conflict set of instantiations. There is only a small variation in the number of relational selection tests performed by each rule; most rules are approximately the same size in terms of number of condition elements and number of constants in each. However, the number of join tests per rule varies greatly because it is dependent on how many working memory elements exist which match each condition independently. This results in poor load balancing among processors.

III. CREATING CONSTRAINED COPIES OF CULPRIT RULES

Rule-based systems which contain *culprit rules* obtain large performance improvements using massive parallelism by *creating constrained copies of these culprit rules*. In addition, culprit rules are conspicuously present in expert database systems [22] and rule-based systems using table-driven rules [13], [6]. The benefit of this optimization is related to working memory size and precondition complexity, *not* working memory contents of the same size [12]. This is analogous to the phenomenon in database systems in which query complexity is affected by the size, but not the contents, of the database [21].

Creating constrained copies of culprit rules is a source-to-source transformation which allows load balancing to occur at the rule level, providing a similar function to intranode parallelism [7], but not requiring special hardware and software combinations to support the optimization.

Working memory elements are matched by the rules' preconditions and created or removed by the actions of selected rules upon firing. The conditions of a given rule match zero or more working memory elements on each cycle. If each condition is

either not matched by an existing working memory element or is only matched by a single one, then the time required to match the rule is proportional to the number of conditions, c, and working memory elements, w: $O(c \infty w)$. On the other hand, if multiple working memory elements match a single condition, each creates a tuple in the selected relation which must be joined with the relations formed by the remaining conditions, requiring many more intercondition tests: $O(w^c)$. Rules that are particularly plagued generate a cross product of instantiations between two or more large sets of elements being joined. These culprit rules slow down the execution of the entire system; in parallel implementations this is even more detrimental because conflict resolution must occur after all instantiations are created and thus a single culprit rule will cause the other processors to idle during the match phase. This situation tends to occur frequently in programs which represent a portion of the knowledge base as large tables in working memory [13] and in programs which analyze large amounts of data in working memory [23].

Certain working memory element types can be identified which are likely to appear in greater numbers than others. For example, it may be known *a priori* that very few working memory elements of type **goal** will exist whereas many elements of type **employee** are likely to reside in working memory at a given time. Thus, rules which match on **employee** working memory elements will require more join tests to determine precondition satisfaction than rules which match only on **goal** elements. Each of the former rules should be rewritten as a set of constrained copies of the original. Each copy would match on a subset of the **employee** elements during the selection phase, reducing the number of instantiations overall for joining. Also, each of the copies can be matched simultaneously.

Suppose, for example, that the following rule is written in order to identify two jigsaw puzzle pieces of the same color and fit them together:

```
(p possible-connection
    (piece     ^color   <X>
               ^id      <I>)
    (piece     ^color   <X>
               ^id      { <J> <>Ê<I> })
-   (goal      ^type    connect
               ^id1     <I>
               ^id2     <J>)
    -->
    (make      goal     ^type   connect
                        ^id1    <I>
                        ^id2    <J>))
```

There may exist many (say $n = 100$) elements of type piece. The first two preconditions would each create selected relations containing n tuples. Then, $n^2 = 10{,}000$ join tests would be required to create a possibly large number of tuples in the joined relation (all pairs of two pieces with the same color), which would in turn be matched in the remaining join tests in the rule. The rule can be copied, say $m = 20$ times, each copy constrained to match only a subset of the elements. For example, the domain of the color attribute may be known to be a fixed set of 20 colors. One of the 20 copies would include the following as its second and third conditions:

```
    (piece     ^color   RED
               ^id      <I>)
    (piece     ^color   RED
               ^id      { <J> <>Ê<I> })
```

The other copies would only match one of the other 19 possible values. Assuming that there is an even distribution of the *colors* among the *pieces* in working memory, each condition would create its selection relation with approximately (n/m) tuples. Each of the m rules would require $(n/m)^2$ join tests: a factor of m fewer comparisons overall even on a sequential

implementation. These m rules, however, could be executed in parallel by m processors. In this example, therefore, the process would be sped up by a factor of $m^2 = 400$.

The method described requires knowledge of the domains of the attributes in order to constrain the copies. This assumption can be circumvented by employing a hashing scheme; each copy of the rule would be constrained to match only those working memory elements with a particular hash value[1]. Once an attribute with enough variability is selected, a new attribute is defined for the working memory element type. Its value will be the result of a hash function performed on the selected attribute. Thus, even if the *colors* of the *pieces* were unknown, the copies could still be created, constrained by differing values of the hash attribute. The copies generated if *pieces' colors* were hashed into four buckets are shown in figure 2.

The generated copies result in an increase in the number of rules active during selection testing. More work is performed in this phase resulting in more selection operations in parallel, each of which would result in a smaller relation to be subsequently joined. According to Gupta [7], the average size of the affect set (the set of rules affected, or required to do join testing in a given cycle) is 30 rules per cycle. This supports the conjecture that massive parallelism is inappropriate for rule-based system execution; no more than 30 processors would be needed if the rules were distributed optimally. These few processors would, however, have to deal with the occasional culprit rule which would slow the execution of the entire system. By creating constrained copies of culprit rules and distributing them to many more processors, each works on a smaller subset of the changes to working memory yielding improved performance. Much of the work is shifted from the join test phase to the easily parallelizable selection test phase.

In addition to the speed up obtained, this technique also provides the advantage of smaller memory requirements for each rule. On fine-grained parallel systems with distributed memory, the number of tuples in the selection-generated relations created by certain preconditions can become large, overflowing the limited memory of the processing element. Upon creating constrained copies of the rules and assigning each to its own processing element, the number of tuples for each can be tuned to the memory limitations.

The selection of which attributes within which classes to constrain requires knowledge of the domain. However, simply selecting the classes of which there will be many working memory elements and the attributes of those classes with high variability provides good results. Although it may occasionally be difficult to assess these parameters, this method was used successfully throughout the experiments presented herein and in [12].

The creation of constrained copies must be performed carefully in order to preserve the overall behavior of the system. For example, assume it is determined that the attribute *color* of the class *piece* is to be constrained. Also, assume that the new attribute *hash-color* will take on one of n values. Several scenarios for the creation of the constrained copies exist, as described in the following copy creation algorithm:

- If there is only one occurrence of a variable as a class's attribute (*piece's color*) in the precondition, only n copies of the rule need be created.

- If two or more occurrences of a variable exist as a class's attribute in a rule, yet they are all tested to be equal to each other, still only n copies are required. This is demonstrated in the above example in which both occurrences of *pieces' colors* are bound to the same variable <*x*>.

- However, if m different variables occur, or if one occurrence must be <*x*> whereas another must be <> <x> then n^m copies must be made. The rule on the left of figure 3 requires $3^2 = 9$ copies if the *pieces' colors* are hashed into three buckets. The rule on the right abstractly represents each of the nine copies, with (*a*, *b*) being (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3), in the different copies.

- Occurrences of variables in negated condition elements cannot be constrained unless they are equality tested with another similarly constrained variable in a positive condition element.

The manual application of this technique is tedious at best; creating constrained copies of culprit rules in a large system by hand is prohibitive. In order to automate the process of creating the constrained copies of culprit rules, a preprocessor was developed which, according to declarations in the source code, produces the constrained copies, accounting for the subtle conditions affecting the number of copies necessary and the assignment of hash values to each of the copies. The declarations driving the algorithm consitute the only domain dependent aspect of the process. The developer needs to assess which classes of working memory elements will have many instances, which attributes of those classes are most variable, into how many buckets the attributes should be hashed, and optionally, which rules to copy. If the rule declarations are omitted, all rules matching on the declared classes will be copied. Given this information, the algorithm described above automatically creates constrained copies of culprit rules.

The case of equality testing of variables, which results in a linear number of copies and a linear speed up on a sequential machine, corresponds to the application of hash indexing the join tests [8], [15]. These optimizations to the Rete algorithm improve the match phase by only equijoin testing new elements which hash to the same value as the existing elements. Similarly, each constrained rule has only the relevant working memory elements to join. However, the creation of constrained copies of culprit rules has three additional advantages:

1. Joins which are not equality tests are also sped up in parallel environments by creating copies for each combination of hash bucket values.

2. Memory requirements are significantly reduced so as to permit small local distributed memory architectures.

3. The transformations occur during a preprocessing source-to-source phase and are applied without modifications to the underlying interpreter.

The technique is also similar to explicitly adding more constant tests, a method employed in SOAR systems in specific instances where constant values are known at compile time [15]. Creating constrained copies of culprit rules generalizes this optimization by using its hashing mechanism.

Creating constrained copies of culprit rules has the effect of transforming intranode parallelism into rule parallelism. Each copy of an original rule parallel processes its intercondition tests which would have occurred within one node of the original. This migration of parallelism into the rule level allows the match algorithm to be orthogonal to the host architecture. Full distribution of rules onto all processors at compile time can be achieved in a variety of hardware topologies. Indeed, since the benefits achieved are due to creating rules which can be matched independently, and because the speed improvements obtained derive from the match phase which is asynchronous, the technique can be successfully applied on parallel architectures independent of their interconnection topology or memory distribution mechanism. There is no interprocessor communication required during the match, or introduced by creating the constrained copies. Therefore, specific topologies or memory organizations (such as distributed versus shared) do not affect the resulting improvement.

The scalability of the technique is limited by the size and variability of working memory, and the proportion of time spent in the match. By adding more processors and concurrently increasing the number of hash buckets and thus the number of constrained copies, a system can be increasingly sped up until the data scarcity or the variability of the hashed attribute limit the

working memory element distribution. Of course, this techniques improves the match time only; its effect will diminish as the other phases of rule-based execution (such as conflict resolution) dominate the execution time.

IV. EXPERIMENTAL RESULTS

In order to evaluate the effect of creating constrained copies of culprit rules on parallelism, each of several systems is executed in an instrumented simulation environment which provides values for sequential match time, T_1, and parallel match time on p processors, T_p, assuming a round robin partitioning of the rules. The times are based on counting the number of comparisons performed[2]. Sequentially, this is

$$T_1 = \sum_{cycle} \text{Comparisons}(cycle)$$

whereas parallel execution time is computed as (where M signifies maximum)

$$T_p = \sum_{cycle} \mathop{\mathrm{M}}_{pe=1}^{p} \text{Comparisons}(pe, cycle)$$

The rationale for this calculation is that all processors perform the match for a given cycle simultaneously. The synchronization point at conflict resolution causes all processors to wait until the one requiring the most comparisons is done. Hence the maximum comparisons over all the processors for a given cycle is the match time for that cycle. All the cycle match times are then summed.

The systems are then preprocessed by creating constrained copies of the identified culprit rules. Speed up (T_1/T_p) is determined relative to two values of T_1: that of the original system on a single processor, and that of the system with constrained copies when run on a single processor. The speed up over the copied and constrained system may be larger or smaller; the former indicates that creating the copies introduces redundancy and thus extra work in the uniprocessor whereas the latter demonstrates that for the particular system, creating constrained copies improves the sequential time as well.

This is illustrated in the schematic graph shown in figure 4. In both cases (*i.e.*, speed up relative to sequential original or sequential copy and constrained systems), the speed up due to creating additional constrained copies eventually stops increasing. However, in the graph in figure 4, the speed up is greater when calculated relative to the original system. This is because the copy and constrained version is faster on the sequential machine as well because of a predominance of equijoins being copied and constrained. In other cases, the curves are reversed; this indicates that creating constrained copies slows down the sequential version by introducing overhead. The improvement is only evident when the copies are distributed because copies in these cases are made from culprit rules with non-equijoins.

Both parallel speed improvements are reported. The number of processors assumed (p) is as many processors as the number of rules in the system after copying. This corresponds to the α operator of the Connection Machine's CM Lisp [10]. It is interesting to note that although some systems may have little speed up due to limited parallelism, many processors may still be required to achieve that speed up.

The table in figure 5 summarizes the characteristics of the systems presented, showing the number of rules and condition elements, and the number of comparisons performed in the executions under differing conditions. Additional systems were tested (12 in total), but the three detailed herein are representative of the findings [12].

A. Tourney

The rule-based system Tourney was made available for this experiment by its author Bill Barabash. The system schedules a bridge tournament among 16 players. The system makes candidate working memory elements corresponding to every possible combination of players. One rule then selects combinations such that no players having already played play together. This is repeated for five nights of play. The approach taken results in a system plagued with culprit rules requiring an enormous number of comparisons. Another study demonstrated that Tourney could be sped up only by a factor of 2.2 on a specific multiprocessor architecture [9].

The Tourney system, however, is an ideal candidate for parallelization by creating constrained copies of its seven culprit rules. As shown in figure 6, a 61-fold speed up is achieved by constraining the copies into three buckets resulting in a 373 rule system. Although memory constraints in the simulation environment used did not allow for further experiments, it is possible that greater speed improvements could be achieved by constraining the culprit rules into 11 buckets resulting in a system of over 60,000 rules. Four of the seven culprits are fourth degree rules which are responsible for the large number of copies generated when hashed into 11 buckets. The sixteen different values for the working memory element attribute being hashed would allow for a continuing speed up increase beyond the 11 bucket scenario. The speed up would continue to increase because 11 buckets is still below the asymptotic 16 buckets.

The continuing speed up when copying and constraining up to 11 buckets is, of course, hypothetical, but the actual measured speed up of 61 fold using 373 processors contradicts the assumed almost complete lack of parallelism in the Tourney system [9]. The speed up is greater when compared to the original system because creating the constrained copies speeds up the system even on a sequential machine.

B. Waltz

The Waltz system written by the author and Donald Ferguson at Columbia University, performs labeling of line drawings of three dimensional objects according to David Waltz's constraint propagation algorithm. Speed up from rule level parallelism was measured to be very limited (2.5 fold). However, creating constrained copies of culprit rules results in a speed up of 13.3 fold over the original system (see figure 7). It is interesting to note that with small amounts of copy creation (creating systems with approximately 50 rules) the sequential system is sped up as compared to the sequentially executed system without copies. This is shown by the speed up factors in figure 7 being higher when T_1 is calculated as the number of matches in the original, uncopied system. As the number of copies increases, however, the redundancy caused by the additional rules' intracondition tests causes the sequential versions to run slower than the original.

C. Energy Analysis Assistant

The Energy Analysis Assistant is an expert system which, given a set of alternatives for a building's design, selects the most energy efficient alternative, predicts its energy performance, and suggests design directives which could improve the building's energy performance. It was written by Bharat Dave at the Department of Architecture at Carnegie-Mellon University.

For the Energy Analysis Assistant, the creation of constrained copies of culprit rules results in speed improvements of over 50 fold under certain conditions. The conditions for achieving this speed up provide insight as to the kinds of systems which benefit most from this technique.

The Energy Analysis Assistant was run with three building alternatives to analyze. These experiments showed minimal speed up and creating constrained copies of culprit rules did not improve the performance beyond four fold. The system was then run with 50 building alternatives to analyze simultaneously. In a database environment, expert systems would be expected to analyze a much larger number of scenarios than the three-building sample provided by Bharat Dave in the example file. The 50 building execution showed only 1.1-fold speed up with rule parallelism alone. However, creating constrained copies of the culprit rules resulted in a 50-fold speed up (see figure 8). Creating a system with 130 rules results in a 38-fold speed up over the sequential time of the uncopied system. About half of the speed up is due to the sequential improvements provided by hashing, whereas the remaining speed up is due to the increased parallelism. As the number of copies is increased so as to comprise a system of over 200 rules, the overhead in the sequential execution of the larger rule base outweighs the hashing advantage, but the improvement due to parallelism increases to provide a 50-fold speed up.

The large degree of parallelism shown to exist in this system provides evidence that the creation of constrained copies of culprit rules is particularly suited to expert systems performing data analysis on large working memories. An increasingly important use of rule-based systems in industry is expert database systems [5], [22]. The poor performance of these systems is addressed directly by parallel processing in combination with creating constrained copies of rules. In order to further demonstrate the effect of the size of working memory on the available speed up, the Energy Analysis Assistant was executed analyzing 10, 30, and 50 building alternatives. The results of these experiments, shown in figure 9, show that the size of working memory is directly related to the speed up obtained. Figure 9 demonstrates that the effectiveness of increasing the number of constrained copies of culprit rules is limited by the number of working memory elements which can be distributed among the different processors.

V. CONCLUSION

Based on the individual experiments reported above, several conclusions can be reached concerning the effect of creating constrained copies of culprit rules on performance via parallelism.

All of the systems tested show a performance improvement when culprit rules were copied and constrained. The magnitude of this improvement varies as a function of the complexity of the culprit rules (*i.e.*, how many joins are in the query) and the size of the working memory (*i.e.*, the size of the database). A representative example of the first case is found in the Tourney system, in which a group of culprit rules perform up to 14 joins each on a possible working memory set of 16 elements. Whereas this particular situation plagues other parallel approaches to its speed up [9], creating constrained copies of these culprit rules demonstrates a 60-fold speed up (which is certainly not its upper bound).

The effect of increasing working memory size on the parallelism provided by copying and constraining is demonstrated by the series of tests on the Energy Analysis Assistant. In those experiments, as working memory size is increased, speed up obtained via copying and constraining increases up to a measured 50 fold.

Generalizing these findings, it is conjectured that the class of expert systems performing database analysis will benefit most from the application of this technique when executed on parallel processors. Their large working memories make them ideal candidates for this optimization.

Although there has been substantial pessimism concerning the parallelization of rule-based systems, there are still many unexplored methods for extracting more parallelism from these programs. The copy and constrain method serves to load balance as well as extract additional parallelism from existing, sequentially written rule-based systems. The speed improvements obtained using this method alone are measured to be over 60 fold. Even on sequential implementations, systems plagued with large numbers of required join tests exhibit improved performance.

Methods to find exactly which rules to copy and constrain, identifying which attributes to hash on, and other factors require further investigation. Careful selection of the working memory element type and attribute to hash, and which rules to copy and constrain is necessary because of large number of copies which can be produced. For example, while only hashing on a single working memory element type and attribute into 10 buckets, a given rule could generate 1000 or more copies if three or more occurrences of the attribute were present and not bound to the same variable. If too many copies are created, such that more rules exist than processors, the overhead of combining different rules in one processor may outweigh the advantage of the added constraints. Another area for further study is fault tolerance which can be provided by modifications to this technique. By selecting overlapping hash functions, a system can be designed to degrade gracefully as processing elements fail.

Creating constrained copies of culprit rules by hashing provides a domain-independent mechanism for extracting additional parallelism from rule-based systems. The resulting speed improvements are encouraging. Driven by these results, other methods should continue to be investigated for extracting additional parallelism from rule-based systems.

References

[1] F. Barachini, *PAMELA: A Rule-Based AI Language for Process-Control Applications*, The First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, pages 860-867, 1988.

[2] C.L. Forgy, *On the Efficient Implementation of Production Systems,* Ph.D. Thesis, Carnegie-Mellon University, 1979.

[3] C.L. Forgy, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence 19(1)*: 17-37, 1982.

[4] C.L. Forgy and J. McDermott, *OPS, a Domain-independent Production System Language,* Proceedings of the International Joint Conference on Artificial Intelligence, 1977.

[5] D. Gordin and C. Castillo, *OD: A Generic Interface between Production Systems and Relational Databases,* IASTED International Symposium Expert Systems Theory and Applications, 1989.

[6] D. Gordin, D. Foxvog, J. Rowland, P. Surko, and G.T. Vesonder, *OKIES: A Troubleshooter in the Factory,* The First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, 1988.

[7] A. Gupta, Parallelism in Production Systems: The Sources and Expected Speed-up, Technical Report, Department of Computer Science, Carnegie-Mellon University, 1984.

[8] A. Gupta, *Parallelism in Production Systems*, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1986.

[9] A. Gupta, C.L. Forgy, D. Kalp, A. Newell, and M. Tambe, Results of Parallel Implementation of OPS5 on the Encore Multiprocessor, Technical Report CMU-CS-87-146, Department of Computer Science, Carnegie-Mellon University, 1987.

[10] W.D. Hillis, *The Connection Machine,* Cambridge, Massachusetts: MIT Press, 1985.

[11] D.P. Miranker, *TREAT: A New and Efficient Match Algorithm for AI Production Systems,* Ph.D. Thesis, Department of Computer Science, Columbia University, 1986.

[12] A.J. Pasik, *A Methodology for Programming Production Systems and its Implications on Parallelism,* Ph.D. Thesis, Department of Computer Science, Columbia University, 1989.

[13] A.J. Pasik and M.I. Schor, Table-driven Rules in Expert Systems, *SIGART Newsletter 87*: 31-33, 1984.

[14] A.J. Pasik and S.J. Stolfo, Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules, Technical Report CUCS-313-87, Department of Computer Science, Columbia University, 1987.

[15] D. Scales, Efficient Matching Algorithms for the SOAR/OPS5 Production System, Technical Report, Knowledge Systems Laboratory, Computer Science Department, Stanford University, 1986.

[16] M.I. Schor, T.P. Daly, H.S. Lee, and B.R. Tibbitts, *Advances in Rete Pattern Matching,* Proceedings of the American Association for Artificial Intelligence, pages 226-232, 1986.

[17] T. Sellis, C. Lin, and L. Raschid, *Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms*, ACM-SIGMOD International Conference on the Management of Data, pages 404-412, 1988.

[18] S.J. Stolfo, *Five Parallel Algorithms for Production System Execution on the* DADO *Machine,* Proceedings of the American Association for Artificial Intelligence, 1984.

[19] S.J. Stolfo and D.P. Miranker, DADO: *A Parallel Processor for Expert Systems,* The IEEE International Conference on Parallel Processing, 1984.

[20] S.J. Stolfo and D.P. Miranker, *DADO*: A Tree-Structured Architecture for Artificial Intelligence Computation, *Annual Review of Computer Science 1*: 1-18, 1986.

[21] J.D. Ullman, *Principles of Database Design,* Rockville, Maryland: Computer Science Press, 1982.

[22] G.T. Vesonder, Rule-based Programming in the Unix System, *AT&T Technical Journal 67(1)*: 69-80, 1988.

[23] G.T. Vesonder, S.J. Stolfo, J. Zielinski, F. Miller, and D. Copp, *ACE: An Expert System for Telephone Cable Maintenance*, Proceedings of the International Joint Conference on Artificial Intelligence, 1983.

Index Terms

Optimization, parallelism, production systems, rule-based systems, transformations.

Preferred Address

Alexander J. Pasik, PhD
Gartner Group, Inc.
56 Top Gallant Road
PO Box 10212
Stamford, CT 06904-2212

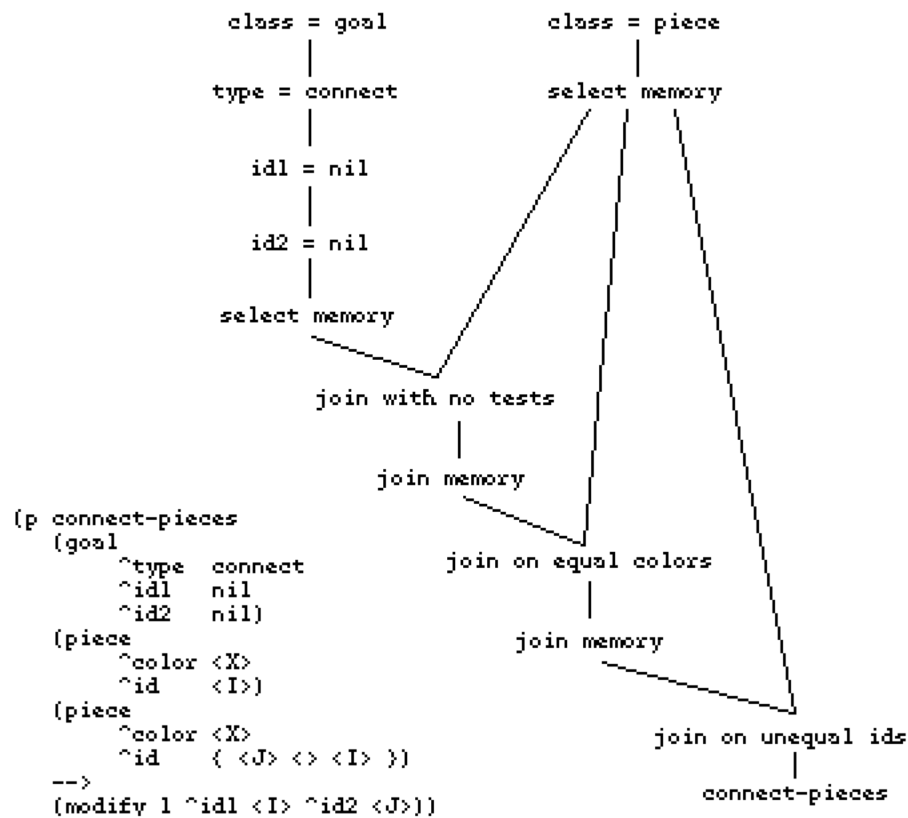


Figure 1

```
(p possible-connection-1
   (piece  ^color    <X>
           ^id       <I>
           ^hash-colorÊ1)
   (piece  ^color    <X>
           ^id       { <J> <>Ê<I>}
           ^hash-color 1)
-  (goal   ^type     connect
           ^id1      <I>
           ^id2      <J>)
   -->
   (make   goal      ^type   connect
                     ^id1 <I>
                     ^id2 <J>))
```

```
(p possible-connection-2
   (piece  ^color    <X>
           ^id       <I>
           ^hash-color 2)
   (piece  ^color    <X>
           ^id       { <J> <>Ê<I>}
           ^hash-color 2)
- (goal    ^type     connect
           ^id1      <I>
           ^id2      <J>)
   -->
   (make   goal      ^type   connect
                     ^id1 <I>
                     ^id2 <J>))
```

```
(p possible-connection-3
   (piece  ^color    <X>
           ^id       <I>
           ^hash-colorÊ3)
   (piece  ^color    <X>
           ^id       { <J> <>Ê<I>}
           ^hash-color 3)
-  (goal   ^type     connect
           ^id1      <I>
           ^id2      <J>)
   -->
   (make   goal      ^type   connect
                     ^id1 <I>
                     ^id2 <J>))
```

```
(p possible-connection-4
   (piece  ^color    <X>
           ^id       <I>
           ^hash-color 4)
   (piece  ^color    <X>
           ^id       { <J> <>Ê<I>}
           ^hash-color 4)
- (goal    ^type     connect
           ^id1      <I>
           ^id2      <J>)
   -->
   (make   goal      ^type   connect
                     ^id1 <I>
                     ^id2 <J>))
```

Figure 2

```
(p separate-pieces
    (piece   ^color   <x>
             ^id      <i>)
    (piece   ^color   <> <x>
             ^id      <j>)
    (together
             ^id1     <i>
             ^id2     <j>)
    -->
    (remove 3))
```

```
(p separate-pieces-a,b
    (piece   ^color   <x>       ^hash-color a
             ^id      <i>)
    (piece   ^color   <> <x>    ^hash-color b
             ^id      <j>)
    (together
             ^id1     <i>
             ^id2     <j>)
    -->
    (remove 3))
```

Figure 3

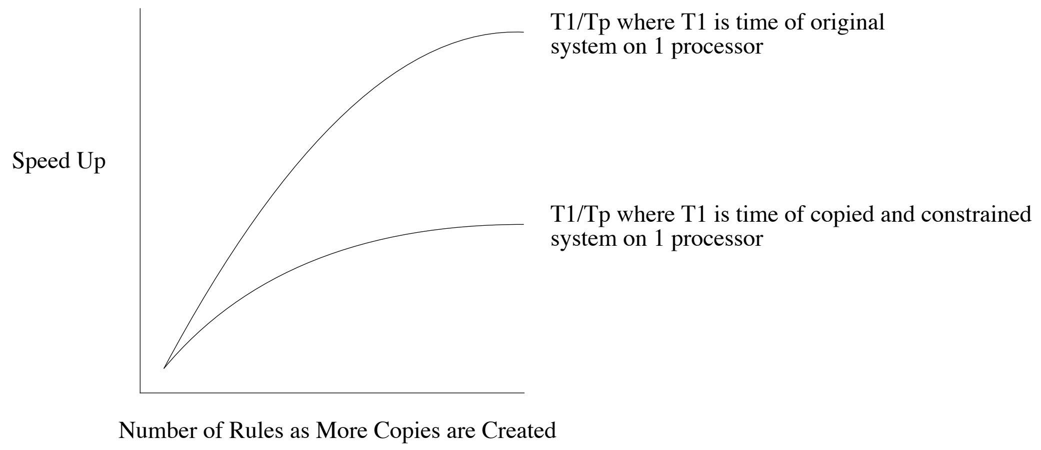


Figure 4

System	Rules	Rules w/Copies	T_1	T_p	T_1 w/Copies	T_p w/Copies
Tourney	17	373	84843558	84747077	23704456	1388079
Waltz	32	164	41025	16298	114477	3082
Energy	61	243	766758	682671	735119	15070

Figure 5

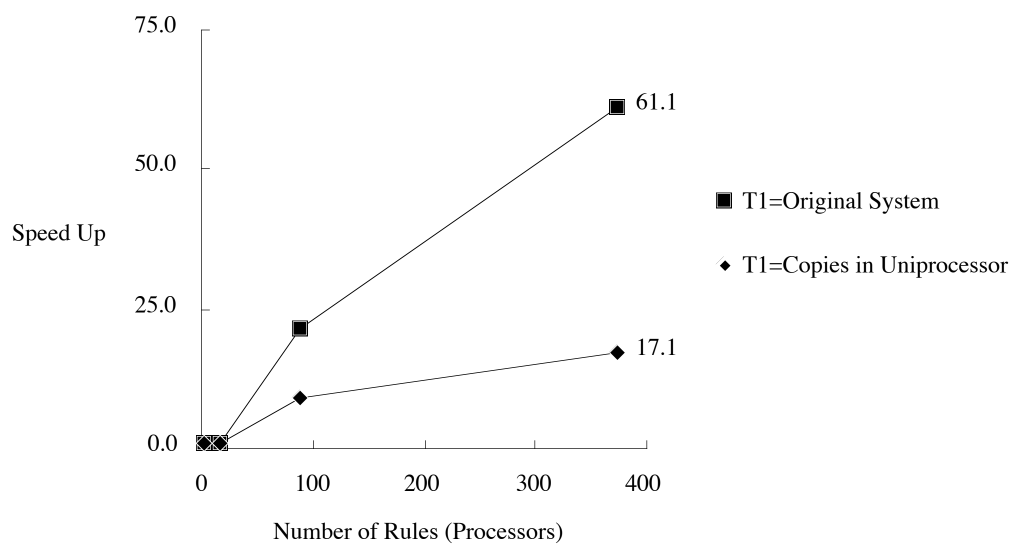


Figure 6

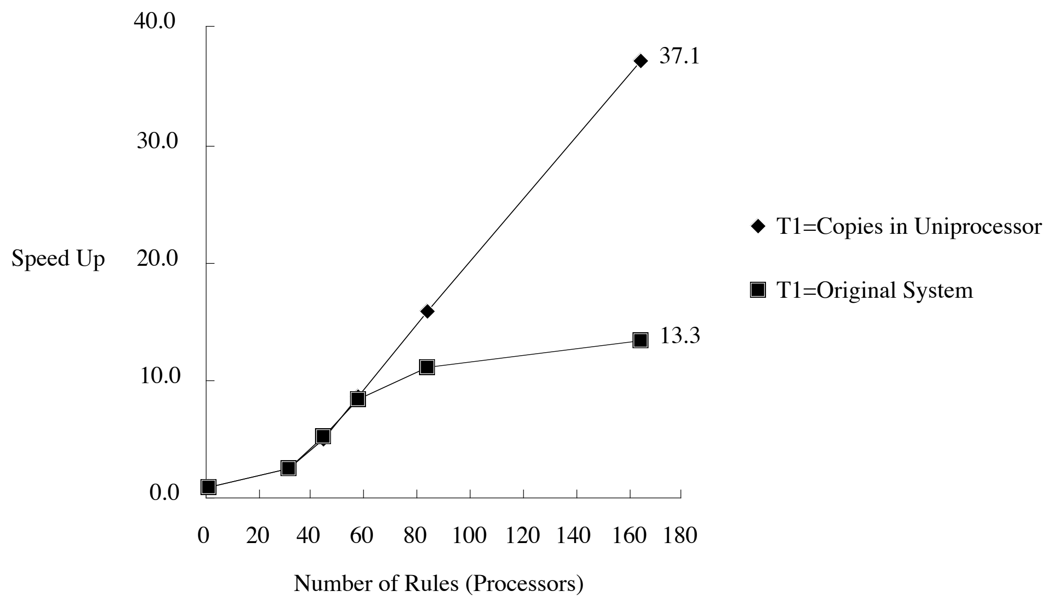


Figure 7

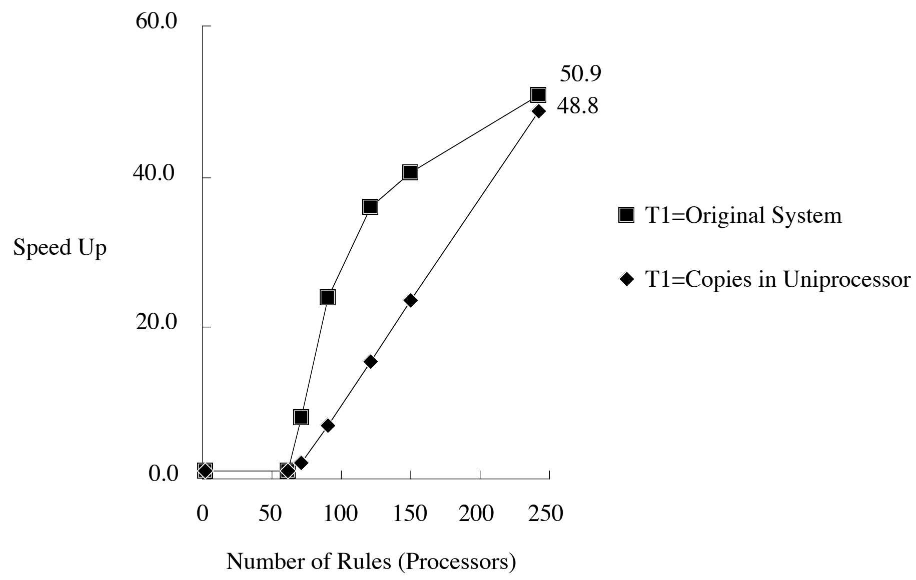
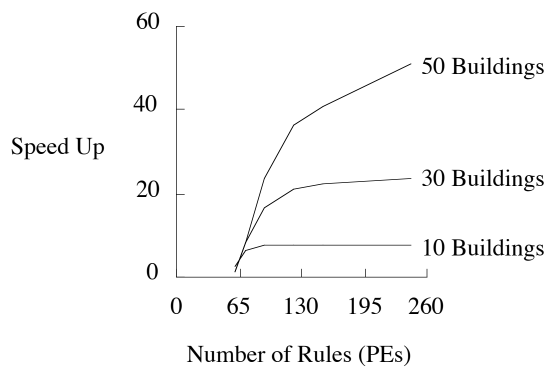


Figure 8



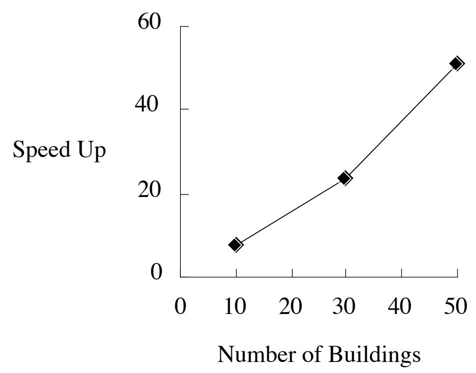


Figure 9

Figure Captions

Fig. 1. A rule and its Rete network.

Fig. 2. A rule copied and constrained into four buckets.

Fig. 3. A rule and the abstract representation of its nine copies.

Fig. 4. Speed up as more constrained copies are created.

Fig. 5. Number of rules and comparisons for the systems tested.

Fig. 6. Effect of copy and constrain on Tourney

Fig. 7. Effect of copy and constrain on Waltz.

Fig. 8. Effect of copy and constrain on Energy Analysis Assistant.

Fig. 9. Speed up due to copy and constrain with different sized working memories.

Biography

Alexander J. Pasik received his B.A., M.S., M.Ph., and Ph.D. degrees in computer science from Columbia University in 1982, 1984, 1986, and 1989 respectively; he maintains his affiliation as Adjunct Lecturer at Columbia University's Department of Computer Science. He is currently Program Director in the Advanced Technology Groups Service at Gartner Group, Inc. His current research interests include the integration of object-oriented, rule-based, and database technologies, with parallel computers as an enabling technology.

Dr. Pasik has been an Assistant Professor at New Jersey Institute of Technology and Assistant Vice President at the Citicorp Technology Office. He is a member of the IEEE Computer Society, the Association for Computing Machinery, and the American Association for Artificial Intelligence.

Footnotes

Affiliation of Author—A. Pasik is with the Department of Computer Science, Columbia University, New York, NY 10027.

[1]Independent of the domain, a randomizing hash function is used. Although better results may be obtained if the hash function is selected specifically for the domain by the programmer, with the the several systems tested in this work, a generic hash function demonstrated good distribution. This can be explained by the lack of clustering in the data resulting in a good distribution by an ordinary hash function.

[2]The tests were performed on a Motorola 68020 Apple Macintosh II running Common Lisp.