Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules

Alexander J. Pasik
Salvatore J. Stolfo

Department of Computer Science, Columbia University
New York City, New York 10027

CUCS 313-87

(212) 280-8119
al@cheshire.columbia.edu

(212) 280-2736
sal@cs.columbia.edu

Abstract

Production systems have pessimistically been hypothesized to contain only minimal amounts of parallelism [Gupta 1984]. However, techniques are being investigated to extract more parallelism from existing systems. Among these methods, it is desirable to find those which balance the work being performed in parallel evenly among the rules, while at the same time decrease the amount of work which must be performed sequentially in each cycle. The technique of *creating constrained copies of culprit rules* accomplishes both of the above goals. Production systems are plagued by occasional rules which slow down the entire execution. These rules require much more processing than others and thus cause other processors to idle while the culprit rules continue to match. By creating the constrained copies and distributing them to their own processors, each performs less work while others are busy, yielding increased parallelism, improved load balancing, and less work overall per cycle.

Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules

1. Introduction

Production systems are a widely used knowledge representation and programming paradigm for the implementation of artificial intelligence (AI) software. The search for efficient hardware and software systems in support of the high-speed execution of AI software is an important and active area of research. Naive implementations of the production system match phase require $O(|P| \times |W|^n)$ comparisons on each cycle, where $|P|$ is the number of productions, $|W|$ is the number of working memory elements, and n is the largest number of preconditions on a production's left-hand side. Using techniques popularized by the Rete match algorithm [Forgy 1982], the dependency on both the size of production and working memories can be reduced. Certain production system programs indeed exhibit running times which are insensitive to the size of production memory. Nevertheless, production system execution is characterized by large variations in the time required to match different rules. Regardless of the match algorithm used, certain productions will require more comparisons than others in determining left-hand side (LHS) satisfaction. By investigating and implementing techniques to balance the load of matching rules, production system execution can be more efficient, both in serial and parallel implementations. This efficiency improvement is a result of decreasing the variance of rule match times and often increasing the size of the *affect set*, that is the set of productions affected by the changes to working memory in a given cycle. The advantages of increasing the affect set size are apparent in parallel implementations because more processors can work simultaneously on smaller problems.

Existing production systems have been written in a fashion indicative of the sequential machinery on which they are executed. Parallel hardware for the execution of production systems has been hypothesized to provide only minimal speed improvements by using techniques such as parallel matching algorithms because the underlying design of these programs is sequential [Gupta 1984, Miranker 1986]. Whereas new, parallel-oriented production system languages are being investigated [van Biema *et al.* 1986], it is still desirable to find other methods of extracting more parallelism from the existing sequential production systems.

The technique discussed herein increases the amount of parallelism during each production system cycle while it decreases the variance of match times for all the productions being matched in a particular cycle. Both advantages are important; more parallelism implies greater

speedup, and less variance indicates that fewer processors will be idle. This is achieved by creating constrained copies of *culprit* rules. Culprit rules are those which require more comparisons than other rules which would therefore slow down the overall cycle time [Stolfo *et al.* 1985]. Each copy of a selected culprit rule is matched against a subset of working memory elements that was relevant to the original rule. Thus, each copy is matched with fewer comparisons and all the copies are processed simultaneously. Rules which require a large number of comparisons to determine precondition satisfaction are those selected for constrained copying. Therefore the variance is decreased while more work can be done in parallel.

2. Anatomy of Rule Matching

In order to determine rule satisfaction, the preconditions are matched against the working memory elements. The match can be broken down into two parts. First, the intracondition tests (α tests, in Rete terminology [Forgy 1982]) correspond to a relational selection on the working memory elements. Then, the intercondition tests (β tests, in Rete terminology) are equi-join operations on the relations which were selected [Stolfo and Miranker 1986].

A simple view of parallel match is that a processing element is assigned to each rule. Thus, the set of α tests for each rule is performed simultaneously, as is the set of β tests. For a given cycle, the changes to working memory are processed by each rule to result in a revised conflict set of instantiations. There will likely be only a small variation in the number of α 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 β tests per rule will vary much more because it is dependent on how many working memory elements exist which match each condition independently. This can result in poor load balancing among processors.

3. Criteria for Creating Copies

Working memory elements represent assertions. They are matched by the productions' preconditions and created or removed by the postconditions of selected rules upon firing. The preconditions of a given production match zero or more working memory elements on each cycle. If each precondition 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 production is proportional to

the number of preconditions and working memory elements: O(c×w). On the other hand, if multiple working memory elements match a single precondition, each creates an tuple in the selected relation which must be joined with the relations formed by the remaining preconditions, requiring many more β tests: $O(w^c)$. Rules that are particularly plagued in this way 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 [Pasik and Schor, 1984] and in programs which analyze large amounts of data in working memory [Vesonder *et al.* 1983].

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 *arithmetic-result* will exist whereas many elements of type *table-entry* are likely to reside in working memory at a given time. Thus, rules which match on *table-entry* working memory elements will require more β tests to determine precondition satisfaction than rules which match only on *arithmetic-result* 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 *table-entry* elements during the α test phase, reducing the number of instantiations overall for β testing. Also, each of the copies can be α and β tested simultaneously.

4. Time and Space Improvements

Suppose, for example, that the following rule is written in order to identify two pieces of the same color and fit them together (OPS5 syntax is used):

```
(p join-pieces
  (piece ^color <X>
         ^id    <I>)
  (piece ^color <X>
         ^id    { <J> <> <I> })
         (goal  ^type  try-join
         ^id1   <I>
         ^id2   <J>)
  -->
  (make  goal   ^type  try-join
                ^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 β tests would be required to create a possibly large number of tuples in the joined relation (all sets of two *pieces* with the same *color*), which would in turn be matched in the remaining β tests in the rule. The rule can be copied, say m = 5 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 {red, blue, yellow, green, nil}. One of the five copies would include the following conditions:

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

The other copies would only match one of the other four 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 $(n/m)^2$ β tests: a factor of m fewer comparisons overall even on a serial implementation. These m rules, however, could be processed in parallel. In this example, therefore, the process would be sped up by a factor of m^2 = 25.

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. 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 which would be generated if *pieces' colors* were hashed into four buckets are shown below.

```
(p join-pieces-1
  (piece ^color <X>
         ^id    <I>
         ^hash-color 1)
  (piece ^color <X>
         ^id    { <J> <> <I>}
         ^hash-color 1)
-(goal   ^type  try-join
         ^id1   <I>
         ^id2   <J>)
 -->
  (make  goal    ^type  try-join
                 ^id1 <I>
                 ^id2 <J>))
```

```
(p join-pieces-2
  (piece ^color <X>
         ^id    <I>
         ^hash-color 2)
  (piece ^color <X>
         ^id    { <J> <> <I>}
         ^hash-color 2)
- (goal  ^type  try-join
         ^id1   <I>
         ^id2   <J>)
 -->
  (make  goal    ^type  try-join
                 ^id1 <I>
                 ^id2 <J>))
```

```
(p join-pieces-3
  (piece ^color <X>
         ^id    <I>
         ^hash-color 3)
  (piece ^color <X>
         ^id    { <J> <> <I>}
         ^hash-color 3)
-(goal   ^type  try-join
         ^id1   <I>
         ^id2   <J>)
 -->
  (make  goal    ^type  try-join
                 ^id1 <I>
                 ^id2 <J>))
```

```
(p join-pieces-4
  (piece ^color <X>
         ^id    <I>
         ^hash-color 4)
  (piece ^color <X>
         ^id    { <J> <> <I>}
         ^hash-color 4)
- (goal  ^type  try-join
         ^id1   <I>
         ^id2   <J>)
 -->
  (make  goal    ^type  try-join
                 ^id1 <I>
                 ^id2 <J>))
```

The generated copies result in an increase in the number of rules active during α 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 joined during β testing. According to Gupta [1984], the average affect set size is 30 productions per cycle. This was presented in order to support the conjecture that massive parallelism was inappropriate for production system execution; no more than 30 processors would be needed if the productions were distributed intelligently. 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 will be working on a smaller subset of the changes to working memory yielding an improved performance. Much of the work is shifted from the β test phase to the easily parallelizable α test phase.

In addition to the speedup obtained, this technique also provides the advantage of smaller memory requirements for each rule. On fine-grained parallel systems, the number of tuples in the selection-generated relations created by certain preconditions can become large and thus overflow 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 is dramatically decreased.

5. How to Create Constrained Copies

The creation of constrained copies must be performed carefully in order to preserve the overall behavior of the system. 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. If there is only one occurrence of a variable as a *piece's color* in the LHS, only *n* copies of the rule need be created. If two or more occurrences of a variable exist as *pieces' colors* 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 *color* must be <*x*> whereas another must be <> <*x*> then *nm* copies must be made. The following rule requires 32 = 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.

```
(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))
```

Also, occurrences of variables in negative condition elements cannot be constrained unless they are equality tested with another variable in a positive condition element. A program was written to automatically create the constrained copies of rules adhering to all the above requirements.

6. Results

Three production systems were executed with different amounts of constrained copying in order to empirically measure the effect on performance. The working memory type and attribute to be constrained were selected by choosing the most commonly occurring type and its

attribute with the greatest variability. The number of α and β tests for each rule at each cycle of execution was recorded. A system configuration was defined as a particular production system copied and constrained into a set of rules such that the overall behavior of the system was unchanged from the original. In other words, the changes to working memory at each cycle were the same.

The three systems used were (1) *monkeys and bananas* which solves the problem of a monkey trying to reach bananas using tools, (2) *puzzle* which solves a jigsaw puzzle, and (3) *waltz* which labels line drawings according to the Waltz constraint propagation method. The original systems were composed of 13, 13, and 33 rules respectively. The average number of working memory elements per cycle in each of the systems was 10, 63, and 42 respectively. By creating constrained copies of culprit rules, the number of rules was increased for each system as follows:

Monkeys and Bananas	13	25	55	139	
Puzzle	13	29	109	185	
Waltz	33	41	65	105	185

Each of these system configurations was executed and the number of α and β tests per rule per cycle was recorded. The maximum number of α (β) tests per cycle indicates how long the α (β) test phase took in that cycle, because each rule is processed simultaneously but all must wait for the slowest. The average of this maximum over all the cycles serves as an indicator of the average time spent in that phase in any cycle. Plotting the average maximum α and β tests against the number of rules in the system shows that as more constrained copies are created, the α test time is essentially unaffected whereas the β test time is substantially decreased (see Figure 1).

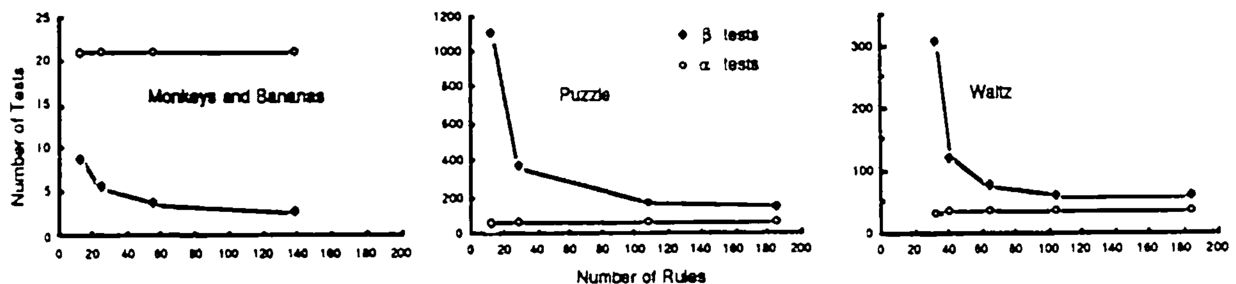

Figure 1. As more copies are created, the average maximum number of α tests per cycle remains constant whereas the average maximum number of β tests decreases. This indicates that the time spent in each cycle is reduced.

The worst cycle is that one which requires the maximum of the maximum β tests per rule over all the cycles. Plotting this against the number of rules shows that with more constrained copies, the worst case cycle becomes much better (see Figure 2).

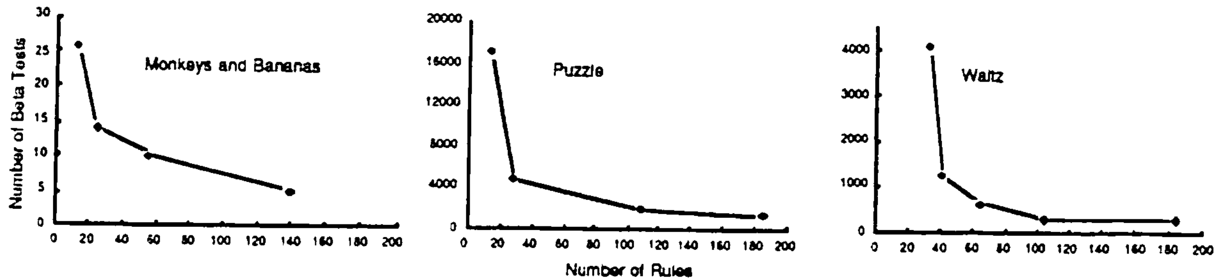

Figure 2. As more rules are created, the maximum over all the cycles of the maximum number of β tests per rule decreases. This indicates that the worst case cycle is improved.

The standard deviation of the number of β tests per rule for a given cycle provides a measurement of the load balance among the processors for that cycle (recalling that α tests are close in number). The average of the standard deviations over all the cycles indicates an approximate measure of the load balance during the entire system execution. Again, plotting this value against the number of rules displays that as more constrained copies of culprit rules are created, the standard deviation of the β test time per rule decreases dramatically (see Figure 3).

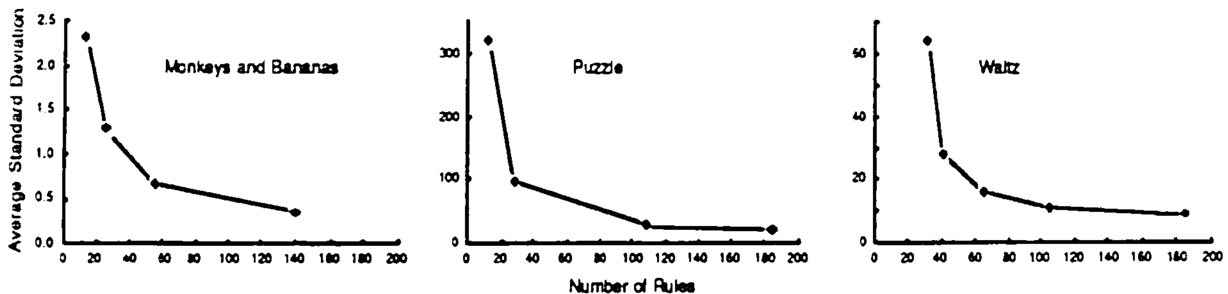

Figure 3. As more rules are created, the average over all the cycles of the standard deviation of the number of β tests per rule decreases. This indicates improved load balance.

The overall time for system matching can be estimated to be proportional to the sum of the maximum number of β tests per rule over all the cycles. This proportionality is supported by the results obtained when plotting the timings for running the *monkeys and bananas* system against the sum of the maximum number of β tests (see Figure 4). The timings were achieved

by running the system on the DADO machine, distributing one rule per processor [Stolfo and Miranker 1986]. The remaining systems could not be run on DADO in their original form due to the large relations formed by the α testing overflowing the limited memory of the processing element. This unfortunate situation, however, demonstrates the additional advantage of creating the constrained copies: the larger rule systems were able to run on the fine-grained machine.

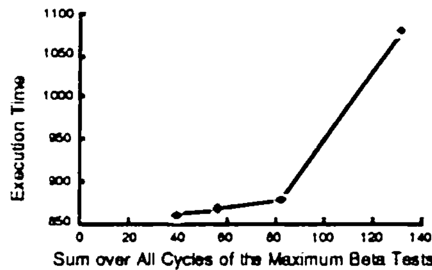

Figure 4. Execution time increases as the sum of the maximum number of β tests increases.

Having established the relationship between execution time and the sum of the maximum number of β tests per rule, these values are plotted against the number of rules in each system configuration. Again, it demonstrates that by creating constrained copies of culprit rules, execution time is dramatically decreased (see Figure 5).

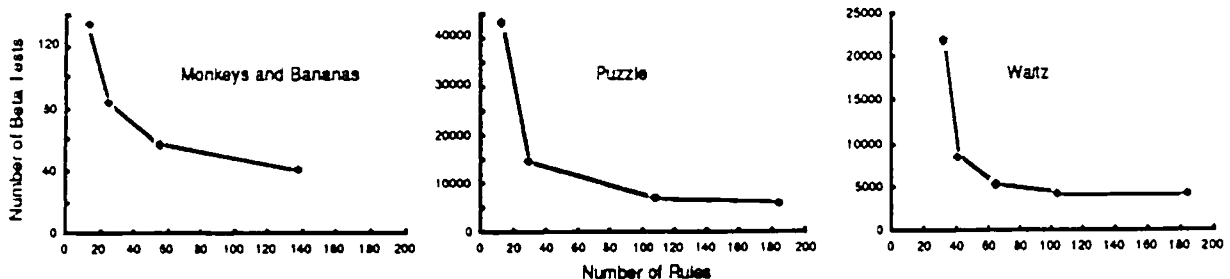

Figure 5. As more copies of culprit rules are created, the sum of the maximum number of β tests per cycle during the whole execution is decreased. This indicates an overall speed improvement when this method is used.

7. Conclusion

Although there has been substantial pessimism concerning the parallelization of production 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 production systems. The speed improvements obtained using this method alone were measured over eight-fold. The advantages of this technique stem from the reduction in both total number of β tests performed, maximum number of β tests per cycle, and the decrease in the variance between rules of the number of β tests required. Overall, many more α tests are performed because of the proliferation of new rules, but each can be processed in parallel. This eliminates the α tests overhead. Even on sequential implementations, however, systems plagued with large numbers of required β tests exhibit improved performance in spite of the added α tests.

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.

Creating constrained copies of culprit rules by hashing provides a relatively domain-independent mechanism for extracting additional parallelism from production systems. The resulting speed improvements are encouraging. Driven by these results, other methods are being investigated for extracting additional parallelism from production systems.

Acknowledgements

The authors wish to thank Andrew R. Lowry and Richard L. Reed and the rest of the DADO project team for their intellectual and technical assistance. Their insight and expertise were essential to the development of the ideas and software required to successfully complete this stage of the research.

References

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

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

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

Pasik A.J. and Schor M.I. (1984) Table-driven Rules in Expert Systems. *SIGART Newsletter 87*: 31-33.

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

Stolfo S.J., Miranker D.P., and Mills R.C. (1985) *A Simple Preprocessing Scheme to Extract and Balance Implicit Parallelism in the Concurrent Match of Production Rules*. IFIP Conference on Fifth Generation Computing.

van Biema M., Miranker D.P., and Stolfo S.J. (1986) *The Do-loop Considered Harmful in Production System Programming*. First International Conference on Expert Database Systems.

Vesonder G.T., Stolfo S.J., Zielinski J., Miller F., and Copp D. (1983) *ACE: An Expert System for Telephone Cable Maintenance*. Eighth International Joint Conference on Artificial Intelligence.