# A Methodology for Programming Production Systems and its Implications on Parallelism

Alexander J. Pasik

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

Thesis Proposal

CUCS 362-88

Abstract

Production systems have been studied as a language for artificial intelligence programming for over a decade. The flexibility of a programming paradigm which allows for loosely structured, independent rules to represent knowledge is attractive. Unfortunately, two seemingly independent phenomena have hindered the ability to take full advantage of production systems. First, the performance of large production systems suffers due to the large amounts of computation required to run them. Second, the programming styles of individuals primarily accustomed to conventional programming has adversely affected the maintainability and performance of the resulting systems. The parallel execution of production systems has been studied in order to address the performance issues. Preliminary results have been interpreted pessimistically; production systems have been observed to contain only moderate to low levels of parallelism. By investigating the issue of programming style, however, it will be shown that the apparent lack of large-scale or massive parallelism is an artifact of this problem. Indeed, a set of programming guidelines and tools will be presented which yield more maintainable, understandable, and parallelizable production systems.

Is there a programming methodology or environment which will allow for the development of more maintainable and parallelizable production systems? This work will attempt to demonstrate that using a combination of several techniques, resulting production systems will more appropriately conform with the theory which supports their use. Production systems are not appropriate for encoding all problem solving tasks. They are appropriate when there is a clear separation of explicit control knowledge, tabular knowledge, and pattern-directed knowledge. This classification has been presented by many researchers in the field, often in order to advocate their separation. The issue has been addressed from a knowledge representation standpoint; here it will be one of several issues which, when addressed properly, will result in systems with improved performance in addition to their more adequate representation of the knowledge.

Substantially more parallelism can be extracted from these systems. In this regard, the techniques complement parallel match algorithms which provide the first step in the solution for mapping production systems onto parallel architectures. The techniques are *table-driven rules*, *creating constrained copies of culprit rules*, *multiple rule firing*, and *combining rule chains*. These methods are combined into a new way of viewing production system execution. Rather than assuming the sequentiality of production systems and trying to extract parallelism *explicitly*, the systems are assumed to be *implicitly* parallel and all necessarily sequential aspects are explicitly defined.

# Table of Contents

1. Introduction .......... 1

2. Control Knowledge .......... 3

3. Obstacles to Parallelization .......... 5

4. Rule Independence .......... 6

5. Using Macrorules .......... 7

6. Table-driven Rules .......... 9

7. Creating Constrained Copies of Rules .......... 10

8. Conventional Code Optimization .......... 13

9. Measurements .......... 20

10. Case Study: Homex .......... 21

11. Conclusions .......... 22

# A Methodology for Programming Production Systems and its Implications on Parallelism

Thesis Proposal

## 1. Introduction

Over 40 years ago, production systems were introduced as a general computational model [Post 1943]. As computer science and, in particular, artificial intelligence developed, the production system paradigm proved itself invaluable in a variety of frameworks. In the early 1970s, several prominent researchers established production systems as an adequate model for the expression of a wide range of problem-solving tasks [Newell 1973, Newell and Simon 1972, Nilsson 1971]. A definitive dissertation was prepared demonstrating production systems to be an ideal language for artificial intelligence programming [Rychener 1976]. All these results were commentaries on the model of knowledge representation provided by production systems. It was not until the development of the Rete pattern match algorithm that production systems could be realistically used for large scale artificial intelligence systems [Forgy 1982]. The Rete algorithm provided a mechanism for the rapid execution of production systems; the many pattern/many object match problem which Rete addressed was an obstacle preventing the use of production systems due to their poor performance.

The increasing availability of production system interpreters resulted in a proliferation of rule-based expert systems. These systems, though, were written primarily by individuals with extensive experience in conventional programming, as opposed to researchers intimately involved in the theory of production systems. In addition, many researchers subscribe to the belief that production systems could and should be used to represent all aspects of a given knowledge base or complete system. Due to these factors, many existing expert systems are encoded entirely in the production system paradigm, including explicit control knowledge, pattern-directed knowledge, and all other categories of information.

Many of the systems represent solutions to problems which have some degree of sequentiality to them. The explicit control knowledge being embedded in the production system's architecture has resulted in an inherent sequential nature in the execution of such systems. However, execution of production systems on parallel architectures intuitively was expected to provide massive performance improvements. Initial results and projections on parallelizing production system execution were disappointing. Unfortunately, researchers have pessimistically generalized these preliminary findings and have decided that only limited improvements on production system performance is available through parallelism [Gupta 1985]. Many of these results were obtained by analyses of a small sample of existing systems which were written with a sequential machine as their target. The lack of parallelism is sometimes due to the sequential nature of the problem, but may also arise from the programming style of the knowledge engineers who built the systems. In the former case, no implementation could provide a parallel solution, but in the latter, techniques could be applied which dramatically increase the level of parallelism.

Is there a programming methodology or environment which will allow for the development of more maintainable and parallelizable production systems? The fundamental nature of production systems is that of a collection of independent rules, each of which can be processed simultaneously. Programmers creating production systems in sequential environments, though, often write interdependent rules. This interdependency is most often due to the attempt to represent control knowledge in production rules, not due to a dependency in the pattern-directed knowledge of the domain. This introduction of sequentiality can be removed by providing a *control language* to represent the sequential aspects of the problem. This approach would invoke sets of production rules to solve subproblems whose solutions can be represented with a collection of independent rules that can be fired in parallel (commutative production systems [Nilsson 1980]). In the past, control languages have been suggested for separating control knowledge from domain knowledge in production systems, solely for knowledge representational advantages [Stolfo 1979, Georgeff 1982]. Here, a control language will provide the enriched knowledge representation as well as a mechanism for explicitly defining necessarily sequential aspects of the problem.

The remaining portions of the problem, with solutions expressed as sets of independent production rules, will be more easily maintained, understood, and parallelized. During each cycle, all satisfied rules should be able to be fired simultaneously. This would result in fewer cycles and more parallel processing at each cycle. This mechanism can only be used if the production systems satisfy certain criteria. The goal of this thesis is to outline these criteria and provide methods for generating alternative solutions to existing problems which can be executed in this fashion. The techniques described will be applied to a reimplementation of a set of existing systems. Also, two complete, commercial expert systems written with the aid of these techniques will be analyzed. The systems will be empirically analyzed to establish their *degree of parallelism* and execution time. The results will be compared to the original systems in order to demonstrate the validity of the techniques for improving performance and parallelism. A qualitative discussion will also be presented concerning the knowledge representational advantages of the new systems. The implications of this thesis include insights on:

- production systems as a more satisfying language for artificial intelligence programming by providing a methodology for rule-base design,
- the ability to take advantage of massive parallelism for the rapid execution of well-designed pattern-directed inference systems,

- the relationship between good knowledge representational models and parallel processing applicability in artificial intelligence systems.
- the generalization of the techniques applied to other formalisms such as frame- and object-based systems.

## 2. Control Knowledge

Production systems are certainly not appropriate for all programming tasks and, as will be discussed, also not for all knowledge representational tasks. Production system architecture provides a *production memory* in which an unordered set of rules are stored, a *working memory* in which the current problem state is represented as a set of assertions, and a *conflict resolution strategy* which, given a set of satisfied rules and the assertions matched, determines which instantiation to fire. Each rule is comprised of a set of conditions (abstractions on potential assertions) and a set of actions (predominantly new elements to be asserted and directives to remove existing elements). In a state-saving environment in which each production rule remembers its partial matches, a production system cycle begins with the match of the previous cycle's new assertions and removals against the conditions in all the rules resulting in a change to the conflict set. Conflict resolution then chooses the instantiation to fire and the actions feed the next cycle. According to this model, the control knowledge determining which rule fires is represented as reactions to the state described in working memory. In other words, a rule can be generally expressed as:

| | |
|---|---|
| **If** | a portion of the current problem fits this abstraction, |
| **Then** | alter the current problem description according to these directives. |

An example rule from an expert system for underwriting home owner's insurance policies which fits this description is:

| | |
|---|---|
| **If** | the home was built before 1950 and |
| | the application requests coverage for more than $150,000, |
| **Then** | assert that an inspection of the premises is needed and |
| | assert that the application should not be approved without referral to the head underwriter. |

Patterns detected in the problem state cause rules to be triggered and thus alter the state of the problem. A rule such as this does not explicitly refer to the next or previous rule in the firing sequence. These rules encode control as *pattern-directed knowledge.*

*Tabular knowledge* is also used to drive rule selection. Often, large tables of information can summarize knowledge about a subdomain of the problem. These tables can be represented in working memory and rules can be driven from their contents, specific entries in the table being selected by problem-state descriptive working memory elements. An example from the underwriting expert system is:

**In WM:** Rating table with fields town, county, state, and zip code.

**If** the address is in a given county, town, state, and zip code **and**
there is no entry in the rating table for the county or zip code **and**
the entry in the rating table for the town in the state has a minimum home value,
**Then** assert that the current address has the minimum home value from the table.

These rules which match on the tabular knowledge are further examples of using pattern-directed information as control knowledge. The patterns are satisfied partially from the assertions describing the problem and partially from the knowledge represented in the table.

Explicit control knowledge is often represented in production rules as well. These rules explicitly direct which rules should be selected next by asserting working memory elements which do not describe facts about the problem but just the current stage of problem-solving being performed. The next rules are triggered by the presence of these control elements. For example:

**If** the stage is calculating premium **and**
the factors are x, y, and z,
**Then** assert that the premium is x*y*z **and**
assert that the stage is underwrite risk.

This embedded explicit control knowledge differs from pattern-directed knowledge. It represents movement through different stages of problem solving, not reactions to problem descriptions. Explicit control knowledge in production systems can be viewed as an attempt to force conventional programming structures into the production system paradigm. Through its use, sequential blocks, conditional code segments, and iteration can be implemented. One of the goals of this thesis is to argue that explicit control information should not be represented in production rules or, if it must, it should be carefully separated from the remainder of the system. An external control language invoking production system executions in sequence, conditionally, or iteratively is a better model for expressing combinations of explicit control and pattern-directed knowledge.

This qualification will be supported both qualitatively by critically contrasting alternative encodings of a set of problems and quantitatively by empirically demonstrating performance and parallelization improvements.

## 3. Obstacles to Parallelization

According to Gupta [1986], the principal limitations to parallelizing production systems are the following:

1. a small number of affected productions per working memory change,
2. a large variation in processing time for these productions,
3. a small number of working memory changes per cycle.

In order to address these problems, the anatomy of the production system cycle is described. The production system execution cycle is often described as match-resolution-act. This is a misleading view; a better description for state-saving production system interpreters is {act-match}-resolution or, more accurately, {select-join}-resolve. The synchronization point is the conflict resolution: it cannot begin until the match is completed. The next select phase begins after resolution chooses an instantiation. Resolution is an operation computationally equivalent to finding the maximum of a set; parallel log *n* solutions exist. The {act-match}, on the other hand, presents several problems for efficient, balanced execution in a massively parallel environment.

The most fundamentally sequential aspect of production system execution is its cyclic nature. Thus to overcome this obstacle to parallelization, the overall number of cycles must be reduced, replacing them with fewer, more computationally intensive yet parallelizable cycles. By decreasing the number of cycles in this way, in Gupta's terms this will result in increasing the number of changes to working memory per cycle, thus addressing point 3. By effectively parallelizing this work, Gupta's point 2 is addressed. The techniques presented will also yield a greater number of affected productions (point 1). Two goals have therefore been identified: reduce the number of cycles and find solutions for the {act-match} phase which can take advantage of massive parallelism. The first goal will be addressed by the using the techniques of *rule independence* and *combining rule chains*. The latter has been investigated previously yielding the TREAT match algorithm [Miranker 1986]. This algorithm provided a better parallel match algorithm than parallel Rete. Nevertheless, load among processing elements was not well distributed and the algorithm alone did not demonstrate the usefulness of massive parallelism in the {act-match}, thus it did not directly address the second of Gupta's points.

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

The model of parallel match assumed is that a processing element is assigned to each rule. Thus, the set of selection tests for each rule is performed simultaneously, as is the set of join 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 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 will vary much more because it is dependent on how many assertions exist which match each condition independently. This can result in poor load balancing among processors. The technique of *creating constrained copies of culprit rules* will demonstrate the usefulness of massive parallelism by providing a mechanism to balance the load during the {act-match} phase.

## 4. Rule Independence

Intrinsic to the nature of production systems is the concept of rule independence. However, this characteristic is lost when explicit control knowledge is embedded in the rule-base. Productions should, ideally, represent independent knowledge chunks; the decision concerning when to apply the actions specified should be dependent on the problem description in working memory, not the presence of working memory elements placed there for control purposes alone.

In the ideal case, any production eligible for firing should be allowed to fire simultaneously with other eligible productions. Unfortunately, several reasons exist which make indiscriminate multiple rule firing in current systems difficult:

- many current systems were written with the assumption that only one production would fire per cycle and thus make use of that fact in their problem solving strategy,
- many systems implicitly rely on the conflict resolution strategy to select the single most specific rule in order to function properly,
- embedded control structures exist in the productions which often result in very few relevant productions eligible to fire in any cycle.

The productions nevertheless could be rewritten into rule sets which are controlled by an external mechanism. Each individual set can be made of rules which can fire simultaneously if they appear together in the conflict set. The productions in a set should be devoid of control knowledge and should not cause incorrect behavior if fired simultaneously. Existing production systems can be rewritten using this methodology so as to provide more clarity, more parallelism during the execution of any rule set, and explicit specification of sequentiality outside the rule formalism. The additional parallelism would arise from a larger number of working memory changes per cycle and a larger number of affected rules per cycle [Ishida and Stolfo 1984]. In addition, there would be fewer cycles overall. The greater degree of parallelism along with fewer cycles would result in much improved performance.

## 5. Using Macrorules

It is often the case that a given task requires a sequence of rule firings to accomplish its goal. Depending on the state of working memory before the task begins, a different traversal through the space of rule firings will take place, yielding a correspondingly altered working memory. In this model, a given execution of a production system segment can be replaced by a single *macrorule*. Theoretically, if all possible sequences of rule firings for a given rule set are known (and there are a finite number of these), one macrorule can replace each traversal through the system, thus causing the rule set's execution to complete in one cycle. This one cycle would consist of matching large sets of preconditions to the initial working memory and executing a large set of actions to alter it. All these operations, however, could be parallelized. Techniques of combining operations have been investigated in the realms of knowledge representation and learning [Rosenblum and Newell 1982], the development of search strategies [Fikes and Nilsson 1971], and standard code optimization [McKeeman 1965]. Similar techniques can be used in production systems in order to improve performance by increasing the amount of parallelism and, in addition, improving modularity by combining whole subtasks into one rule.

A set of macrorules would accomplish the same task as the original set of rules. The macrorule scenario would be superior in two respects: the rules representing the solution to the task would be independent of each other, each representing a different complete scenario, and the macrorule version would be more parallelizable because there would be more working memory changes in each of the fewer cycles and more rules affected by each of the changes (rules would contain more conditions as well as actions).

As a methodology for writing production systems, this technique will be demonstrated to provide superior systems by a qualitative argument as well as quantitative empirical performance and parallelization measurements.

The technique can also be applie [illegible]xisting systems. The systems can be rewritten after an analysis of their execution behavior. [illegible] obstacles to be overcome include the possibility of an infinite number of possible executions through a rule set due to loops in inference, the likelihood of an unmanageable number of executions even if no loops exist, and the problem of figuring out all possible executions even if the number is manageable. One approach to solving these problems which will be evaluated is the use of *execution expressions*. An execution expression is a regular expression. The terminal symbols are rule names and the operators are disjunction (+), sequence (,), and arbitrary execution (*). The form $(P_1+P_2)$ indicates that either $P_1$ or $P_2$ must fire. $(P_1,P_2)$ means that $P_1$ must fire and then $P_2$ must fire. $(P_1)^*$ means that zero or more firings of $P_1$ must occur. Although a regular expression is not descriptive enough to precisely define a given production system execution, it can be used to constrain the possible executions adequately. The interesting execution chunks are those which can be broken down into a finite number of possibilities; infinite sequences cannot be rewritten into a set of macrorules, so a more powerful description of infinite sets is not necessary.

Any production system can be described by the execution expression $(P_1+P_2+...+P_n)^*$. If it is known that $P_1$ always fires first and then never again, the expression can constrain the execution as follows $P_1,(P_2+P_3+...+P_n)^*$. Once a constrained execution expression for the production system is obtained, the portions of the expressions which do not contain * operations can be extracted. The rules in these pieces can be translated into a set of macrorules.

The derivation of the execution expression for a production system requires the specification of an abstract description of the possible initial working memory. From this and the set of rules, a finite state automaton is constructed where the states are these abstract working memory descriptions and the arcs are labeled with the rule names. The execution expression is then derived from the automaton. Other approaches to the derivation of execution expressions have been studied including the automatic learning of these control descriptions from sample execution traces [Stolfo 1979].

The execution expressions for three production systems have been derived. They illustrate three different scenarios, each of which can be parallelized in a different way. One system, MESGEN (a portion of the system ANA developed at the University of Pennsylvania by Kukich in 1983), resulted in an execution expression characterized by large strings of rules which could be combined, occasionally being cut by a single-rule tight loop (used for counting elements which matched some pattern). This system, when rewritten by combining the rule chains would execute in far fewer cycles.

Another system, Homex (developed at Fifth Generation Computer Corporation by Pasik and Lowry in 1986), had an execution expression which was quite different. This system was built with parallel execution in mind as a possibility. Therefore, there were few chains of rules to be combined. Nevertheless, the rules were mostly able to be fired in parallel and a complete execution on a parallel machine would require approximately 10 cycles, regardless of the size of the problem being solved.

Finally, the execution expression of a production system to label line drawings using Waltz constrain propagation was derived. This represented an intermediary between the two previous systems. There were several rule chains to combine and the resulting system would contain some new rules and some original ones which could be fired in parallel.

Other methods for deciding which rules to combine will be discussed as well. A run-time solution involves creating macrorules each time two rules fire in sequence. An upper limit on the total number of rules in the system will be maintained by discarding the most infrequently (or least recently) used rules. The criteria for the decision of which rules to discard will be compared to those involved in paging decisions in virtual memory operating systems. One of the issues which arises when dynamic methods are used is whether a macrorule replace the original rules or simply augment the rule set. If the macrorule is generated from rules which must fire in the order specified then they can be replaced. If, however, a heuristic method is used to control macrorule generation, the original rules must be maintained and an algorithm (such as least recently used rule removal) for the control of rule set size must be used. The scenarios requiring different solutions will be discussed and compared.

## 6. Table-driven Rules

Production rules and working memory elements are often categorized as representing long- and short-term memory respectively; the production rules represent the knowledge of the problem domain whereas the working memory describes the state of the particular problem being solved. Nevertheless, adherence to this distinction is not required. Table-driven rules can be used in conjunction with tables in working memory to represent long term domain knowledge [Pasik and Schor 1984]. The technique provides knowledge representational and system maintenance advantages. Often experts organize their knowledge in categories. Problem solving can involve the selection of a category and its associated information which best fits the current problem description. Thus, a table in working memory can be built representing a relation among various attributes of a problem. Rules can then select entries from this table according to other relevant assertions in working memory, extracting additional needed information.

In addition to the knowledge representational adequacy of table-driven rules, their use improves the maintenance of the system. Tuples in these tables are easily added and removed and few, concise rules can be written which are driven by the tables. The alternative is a large collection of rules which vary only according to a small number of parameters.

The disadvantage of using tables and table-driven rules is realized when attempts are made to parallelize the system. A table-driven rule will generate large selection relations to be processed in the join phase. This implies that, in non-shared memory architectures, the processor handling the table-driven rule will require much more memory than an average rule. In addition, the time require to match the rule will be substantially above average, thus slowing down the entire execution. However, the advantages provided are sufficient to warrant an investigation to determine methods of parallelizing this process. These same methods will be useful in general whenever a load balancing problem occurs in production system execution. The technique of creating constrained copies of culprit rules addresses this issue formidably.

## 7. Creating Constrained Copies of Rules

Whereas maintaining rule independence and using macrorules both improve parallel performance by reducing the number of cycles by creating more parallel work, neither technique addresses the issue of balancing the load over the processing elements. The problem is further accentuated when table-driven rules are used to encode tabular knowledge in production systems. Table-driven rules, however, are encouraged in the methodology due to their clear and maintainable representation of the knowledge. The technique of *creating constrained copies of culprit rules* [Pasik and Stolfo 1987, Stolfo *et al.* 1985] addresses the load balance problem elegantly and sufficiently both with and without table-driven rules.

Working memory elements are matched by the productions' conditions and created or removed by the actions of selected rules upon firing. The conditions of a given production 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 production is proportional to the number of conditions, $c$, and working memory elements, $w$: $O(c \times w)$. 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 join 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 as well as 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 join tests to determine rule 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 selection test phase, reducing the number of instantiations overall for join testing. Also, each of the copies can be selection- and join-tested 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 (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 conditions would each create selected relations containing $n$ tuples. Then, $n^2 = 10{,}000$ join tests would be required to create tuples (possibly very many of them) in the joined relation (all sets 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 $m = 5$ 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 $(n/m)^2$ join tests: a factor of $m$ fewer comparisons overall even on a sequential 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 removed 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 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 during join testing. According to Gupta [1985], 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 join test phase to the easily parallelizable selection test phase.

In addition to the speedup obtained, this technique also provides the advantage of smaller memory requirements for each rule to store its joined relations. On fine-grained parallel systems without shared memory (such as DADO), the number of tuples in the selection-generated relations created by certain conditions 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.

The selection of which attributes within which classes to constrain requires knowledge of the domain. The simple directive of selecting the classes of which there will be many working memory elements and the attributes of those classes with high variability will provide good results. Nevertheless, it may often be difficult to assess these parameters.

Initial tests have been performed estimating the amount of parallelism and speedup of three systems by creating constrained copies of culprit rules. They have demonstrated that as more copies of culprit rules are created, greater parallelism and faster execution time is achieved (see Figure 1).

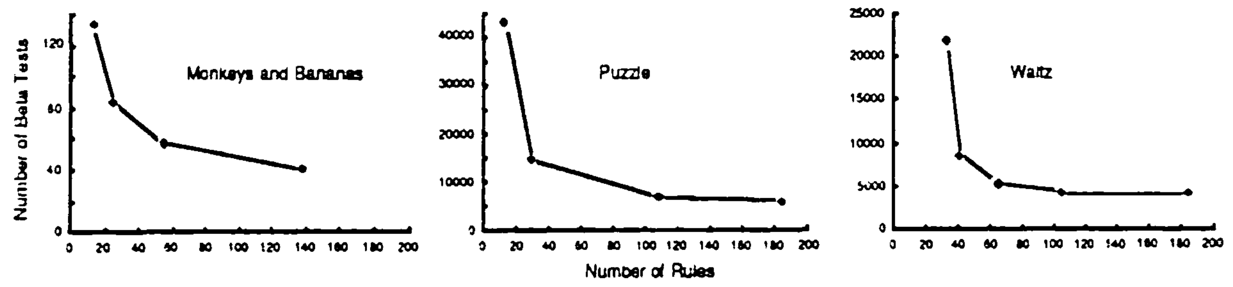

Figure 1. As more copies of culprit rules are created, the sum of the maximum number of intercondition tests per cycle during the whole execution is decreased. This indicates an overall speed improvement.

## 8. Conventional Code Optimization

A broad definition of conventional code optimization is the detection of certain patterns in code followed by the replacement of this code with more efficient constructs (Aho and Ullman 1977). Whereas optimization of conventional programs can yield up to a factor of 2 or 3 improved speed, optimization is more necessary in AI languages and can yield order of magnitude speed improvements (Earley 1975, Schwartz 1975a, Schwartz 1975b). The methodology described can be interpreted as a set of code optimization techniques; many conventional code optimizations were, after all, derived from programming guidelines. However, the techniques are specifically designed for production systems, and the scope of the code improvement includes both efficiency and maintainability. Conventional code optimization deals with automatic detection and replacement strategies applied to intermediate code produced by compilers. The production system methods are presented as code writing guidelines along with, in the case of copy and constraining, an automatic tool for the application of the method. For the sake of comparing conventional code optimization to the methodology described, however, these fundamental differences can be overlooked.

*Loop Optimization.*

According to the 90–10 theory (90% of a program's execution time is spent in 10% of its code) [Knuth 1975], the most important code optimization techniques are concerned with the improvement of inner loops. This loop optimization includes repositioning of loop-invariant code, elimination of induction variables (often accompanied by a reduction in operator strength), loop unrolling, and loop jamming.

The movement of loop-invariant computations in conventional code optimization improves the efficiency of code by reducing the amount of work being done in each iteration of the loop. Loop-invariant code is repositioned so as to be executed once before the body of loop is entered. A distant analogy can be made between this process and creating constrained copies of culprit rules. Just as the most compute-intensive portion of conventional programs is their inner loops, the intercondition test phase is the correspondingly intensive code in the production system execution cycle. Creating constrained copies of culprit rules involves identifying patterns in potential working memory configurations and adding constraints to copied rules so as to redistribute the work out of the intensive intercondition test phase and into the intracondition test phase. Although this "code movement" is not loop-invariant code, the process common to both optimization techniques is the offloading of work from an intensive region by adding work into a previous, less intensive program section.

The elimination of induction variables is a technique specific to conventional program's loop structures. A loose interpretation of the process is the detection of two (or more) variables which can be combined into one serving both (or all) purposes. In this sense, it is related to the local optimization technique of elimination of common subexpressions. There, an expression calculated twice is only evaluated once and the result used in both occurrences. Both of these optimizations result in a single operation replacing several. Using macrorules accomplishes a similar result by eliminating changes to working memory which would have been done and undone by the chain of rules being replaced. For example,

**If** A, B, and C
**Then** make D and E

and

**If** B and D
**Then** make F and remove D

could be replaced by

**If** A, B, and C
**Then** make E and F.

Thus, the operations of **make D** and **remove D** are eliminated. Although this process has nothing to do with induction variables, removing a data item and making one rule serve the purpose of both original rules is distantly related to the conventional code optimization process. This same example is more closely related to the peephole optimization process of removing redundant loads and stores.

Conventional loop unrolling is used to reduce the number of exit tests required in a loop's execution. Its is best explained by referring to the small example below. If a loop is known to execute a fixed number of times, the body of the loop can be duplicated so that the loop will perform its exit test 50% fewer times.

```
for (i=1;i<=100;) {
        a[i] = b[i]
        i = i+1
        }
```

becomes

```
for (i=1;i<=100;) {
        a[i] = b[i]
        i = i+1
        a[i] = b[i]
        i = i+1
        }
```

In effect, loop unrolling achieves the reduction in exit tests by replacing many small iterations with fewer large iterations. The process is similar to using macrorules and multiple rule firing in that these techniques also replace many small cycles with fewer large cycles. Both the conventional loop unrolling and the production system techniques reduce the overhead associated with the cycling: the number of exit tests and the number of conflict resolutions respectively. The production system techniques are also used to improve performance via increasing the available parallelism. It is interesting to note that conventional loop unrolling may also increase the parallelism in conventional code.

Similar to loop unrolling, loop jamming also replaces small loops with larger ones. Loop jamming refers to the combination of two separate loops into one. The analogy described above is even more appropriate for loop jamming. Macrorules represent the combination of two distinct pieces of code into one unit as opposed to the repetition of the same piece. The benefits of reducing overhead and increased potential parallelism are the same as those discussed for loop unrolling.

In conventional code, loops are easily detected because of the high level control constructs. Loops in production systems are hard or impossible to detect. The idiom corresponding to conventional code's loop constructs is

1. initialize working memory,
2. repeatedly fire a rule (or set of rules) over a number of matching working memory elements,
3. fire a loop exiting rule.

This is accomplished by a number of independent rules. Syntax provides little guidance since rules can be placed far apart in the production system program text. Indeed, the detection of such loops requires semantic knowledge applied to an analysis of the potential behavior of the rule base. Furthermore, there are no guarantees that a loop will be executed in its entirety. Since any rule can fire during the course of executing the body of a production system loop, the loop may be executed at unpredictable points in the computation. The conflict resolution strategies employed by OPS5 programs depend primarily on runtime behavior that cannot be determined at compile time.* Hence, the sharp control knowledge available at the compile time of conventional programs is only fuzzy or nonexistant in the production system case. This same argument holds when attempting to use other optimization techniques requiring data flow analysis.

The methodology proposed here puts the burden on the knowledge engineer to write code more efficiently in the first place in the absence of a very smart optimizing compiler. Production systems, after all, are declarative programming languages. The claimed advantages of production system programming of modularity and expressiveness afford perhaps the easy representation of human knowledge but at the expense of generally inefficient machine execution. There is need,

* OPS5's conflict resolution strategy favors instantiations of rules matching working memory elements asserted more recently than others. The order in which elements are asserted is not known at compile time since the initialization of working memory or the interactive behavior with the program falls victim to the vagaries of the user.

therefore, for a well structured programming methodology to balance the benefits of expres  on with the requirement of efficiency.

*Peephole Optimization.*

Peephole optimization refers to code optimizations resulting from an analysis of a small range of instructions (McKeeman 1985). An important characteristic of peephole optimizations is that an application of one such optimization often leads to one or more additional possibilities for further optimization. The construction of macrorules and the creation of constrained copies of culprit rules share this attribute of peephole optimization. When a macrorule is created, the set of possible sequences of rule firings changes. This may provide opportunities for further combination of rule chains. Also, the new macrorule may turn out to be a culprit rule requiring many intercondition tests. This would provide an opportunity for further parallelization and performance improvement by creating constrained copies of the macrorule. Creating constrained copies of one culprit rule may illuminate the presence of another culprit rule previously masked by the "worse culprit". In this case, the first constrained copy creation leads to a second.

Specific peephole optimization techniques can be compared to the production system techniques described. They are redundant loads and stores, jumps over jumps, unreachable code, and multiple jumps. Other peephole optimizations such as algebraic simplification, reduction in strength, and use of machine idioms are specifically related to conventional code optimization and have no apparent meaningful analogy to the production system environment. Production systems, after all, execute a relatively simple sequence of operators: test a set of symbolic structures, then add or remove a symbolic structure to or from memory.

As previously discussed in the context of induction variables, the elimination of redundant loads and stores is directly related to the elimination of temporarily used working memory elements during the creation of macrorules. Redundant loads and stores occur when conventional code generators naively create code such as

```
MOVE   R0, A
MOVE   A, R0.
```

Here, the second instruction can be removed because there is no label (thus it will only be executed immediately after the previous instruction) and R0 is guaranteed to already contain the value of A. The redundant instruction is analogous to the make followed by remove in two consecutive rules which can be combined. This combination allows for the removal of the actions.

Jumps over jumps, elimination of unreachable code, and rewriting multiple jumps (that is a jump to a jump should be replaced by a jump straight to the second jump's destination) are all examples of peephole optimization involving the removal of dead code which results from a naive code generation or the result of a previous application of another peephole optimization. This dead code removal is demonstrated in the production system environment by the example provided in the discussion of the elimination of induction variables. Creating macrorules results in dead code when a working memory element is made and removed in the sequence of rule firings being combined into one rule. The elimination of these actions provides an additional advantage to using macrorules: not only does the resulting system have more parallelism and better performance due to its fewer cycles and more effective work per cycle to be parallelized, unnecessary work is extracted from the system as well.

*Optimizing Conventional Code for Parallel Execution.*

Many code optimization techniques have been adapted for the improved implementation of code on parallel processors. Unfortunately, most of these optimization techniques rely heavily on data flow analysis which can be readily performed on convention code but which would be difficult or impossible on production systems [Kuck *et al.* 1980]. Data flow analyses have been applied to production systems [Moldovan 1984, Ishida and Stolfo 1984], but these studies have been used to determine the data dependency between a pair of rules. This is useful for determining whether two rules can be fired in parallel, but it cannot provide information about the overall sequence of rule firings necessary for the application of the code optimizations described. An important area of future research is the design of a knowledge engineering tool which would provide a mechanism for the representation of the data flow of a production system. This external data flow representation could be used to apply more optimization techniques to production systems. In addition, the techniques presented in this methodology could be automatically applied given the data flow description, instead of relying on the knowledge engineer to perform these programming techniques by hand.

Two methods of optimizing conventional code for parallel execution are *forall transformations* and *pipelining* [Padua *et al.* 1980]. These techniques are illustrative of the need for data flow analysis for the application of these automatic optimizations. Both these techniques are applied to detected for-loops in the source code. Forall transformations parallelize the code by method similar to the creation of constrained copies of rules. A copy is made for each iteration of the for loop constrained with its index variable being set to the value for one of the iterations. One copy is allocated to each processor. Code is inserted to synchronize the processors so that no statement is executed before another upon which it is dependent. Certain loops with little dependency within their bodies execute extremely quickly, achieving a linear speedup on the number of processors used. Others can achieve no speed up at all. For example, the following loop would require synchronization between the two statements and thus its parallel execution time would be the same as its sequential time.

```
for (i=1;i<=n;i++) {
        a[i] = b[i-1] + 2;
        b[i] = c[i] + 1;
        }
```

However, if the two statements are reversed (yielding the same functionality) distributing the $n$ iterations over $n$ processors, using the forall transformation would result in an execution time of 2 units (each statement requiring 1 unit time) thus achieving a linear speedup.

Pipelining for loops distributes the $m$ individual statements of the body of a loop over $m$ processors, placing synchronization code where appropriate. Using the above example, the two statements are not dependent on each other within one iteration. However, the first statement is dependent on the second statement of the previous iteration. Thus, pipelining this code into two processors would result in a speedup of slightly less than linear (due to the first cycle only executing the second statement and every subsequent cycle executing both the first statement of one iteration and the second statement of the previous iteration). However, this linear speedup is proportional to the number of statements in the loop (only 2) whereas the forall transformation with the statements reordered would result in a linear speedup proportional to the number of iterations (possibly very large). Pipelining is vaguely reminiscent of using macrorules in that operations from a set of distinct cycles are combined into one cycle. If these macrorules were themselves distributed, the analogy would be stronger.

Research on the parallelization of conventional code has provided a set of measurements which can be used to determine speedup, efficiency, utilization, and redundancy in parallel versions of programs [Kuck *et al.* 1974]. These techniques will be used and adapted in the measurments of parallelism in production systems.

*Algorithm Optimization.*

The conventioanl code optimization technique with the most impact is algorithm optimization. Using a better algorithm can provide improvements much greater than those achievable by a compiler. The replacement of an $n^2$ sorting routine with an $n \log n$ one provides a speed improvement far beyond that available from the conventional code optimization techniques described so far. The methodology for writing production systems and the tools for creating constrained copies of culprit rules can have effects on performance which are dramatic. This is mostly due to the techniques being more closely related to algorithm optimization than other code optimizations.

The algorithmic changes are the introduction of hash partitioning when using constrained copies of culprit rules, the use of parallelism to handle more matching concurrently, and the provision of more concurrently available matching by the collapse of the match of several cycles into one by multiple rule firing and using macrorules. These benefits provide substantial performance improvements and increased parallelism in production system execution.

## 9. Measurements

In addition to the qualitative discussions on the merits of the methodology and techniques, empirical studies will be performed in order to validate the techniques' effects on performance and parallelism. A series of measurements will be applied to the execution of several expert systems to assess the improvements due to the use of the techniques. Five systems written without the benefit of the methodology will be rewritten semi-automatically (see Figure 2) using the techniques described. Also, two large commercial systems written according to the guidelines for rule independence and external control structures will be tested with and without the additional use of -copy/constraining and using macrorules.

Each of the above systems will be run in an OPS5 environment altered so as to collect information on the number of tests (selection and join) performed per rule in each cycle. From this data, assuming one processor per rule, measurements will be taken to determine overall performance and amount of parallelism. Several measurements will be applied including the following two in order to support the desired results.

- The ratio of the maximum number of tests over all the rules in each cycle to the average number of tests per rule will provide insight on the degree of parallelism. The closer the value is to 1, the more balanced the execution is.
- The sum over all the cycles of the time required to perform the maximum number of tests over all the rules in the cycle plus conflict resolution time of the cycle will be an indicator of overall performance. Smaller values will indicate improved performance.

There are no existing standards for measuring the quantities described. A research topic itself is the development of set of measures for evaluating and comparing parallelism in programs. This issue will be addressed by providing several alternative measures and discussing the implications of each. Other measurements which will provide insight on parallelism and performance will be investigated, especially those which have been demonstrated as useful in determining the effectiveness of parallelizing conventional code.

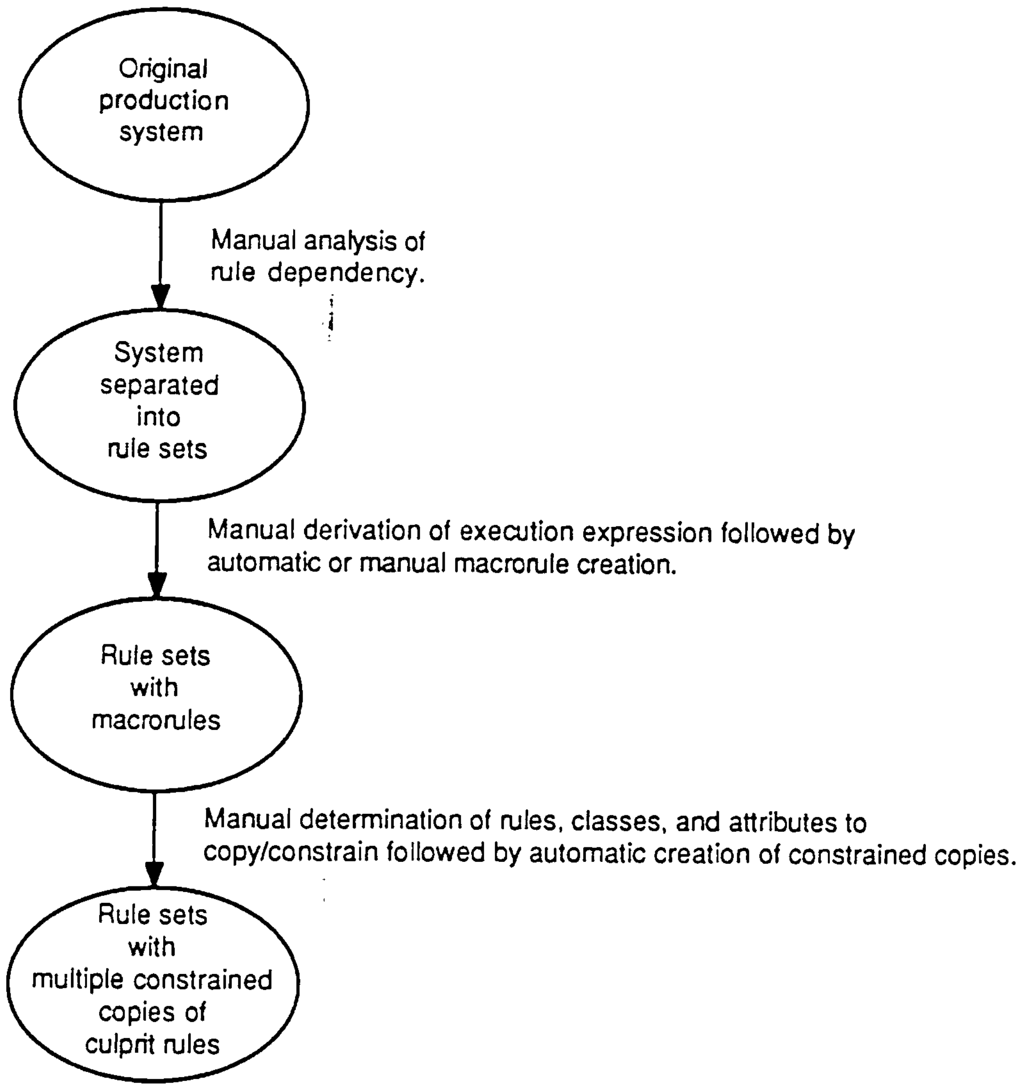
Original production system
Manual analysis of rule dependency.
System separated into rule sets
Manual derivation of execution expression followed by automatic or manual macrorule creation.
Rule sets with macrorules
Manual determination of rules, classes, and attributes to copy/constrain followed by automatic creation of constrained copies.
Rule sets with multiple constrained copies of culprit rules

**Figure 2.** Rewriting existing systems is accomplished by manual analysis and the application of automatic tools.

## 10. Case Study: Homex

Homex is an expert system developed by Alexander Pasik and Andrew Lowry at Fifth Generation Computer Corporation for Home Insurance Company. Homex underwrites (evaluates risk) of new home owner insurance policies specifying whether the company should insure the home and, if so, whether certain additional criteria are required. The system was written using the methodology described: rule independence was maintained when possible, table-driven rules were used, explicit control knowledge among rule sets was represented separately (albeit still in OPS5), and rules which could be combined into macrorules or copy/constrained have been identified. The architecture of Homex, the experiences which concluded in its completion, and the empirical analysis of its performance will be discussed. Homex will also be compared to other systems built with and without the methodology described.

## 11. Conclusions

Although production systems were originally introduced for artificial intelligence programming because of the flexibility they provide for incremental development, many systems have specifically voided this advantage by embedding explicit control knowledge. This has yielded systems which are very difficult to maintain and debug because of the extensive interdependency of the rules. No longer can a developer add new rules at will without completely understanding all of the embedded control structures in these interdependent rule systems. In addition, this style has contributed greatly to the apparent minimal amount of parallelism in the systems. The methodology which will be described and evaluated will provide a mechanism for the construction of better production systems. Also, methods and tools will be developed which will help in the construction of these systems as well as the rewriting of existing systems.

The substantial pessimism concerning the parallelization of production systems will be addressed by providing solutions to the problems of few affected rules, poor load balance, and few changes in working memory per cycle. 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 join tests performed, maximum number of join tests per cycle, and the decrease in the variability between rules of the number of join tests required. Overall, many more selection tests are performed because of the proliferation of new rules, but each can be processed in parallel. This parallelization reduces the selection test overhead. Even on sequential implementations, however, systems plagued with large numbers of required join tests exhibit improved performance in spite of the added selection tests.

The low-level parallelism provided by match algorithms and enhanced by creating constrained copies of culprit rules is the first step in extracting more parallelism from production systems. The techniques of multiple rule firing of independent rules enables even more work to be performed in parallel. Also, by analyzing possible execution paths of production systems, rules can be rewritten into macrorules which provide an even greater degree of parallelism by reducing the number of execution cycles while increasing the amount of parallelism available in each cycle.

Although more methods may exist for the optimization of production systems for parallel execution, the techniques described each address the issues as presented by Gupta. Few changes to working memory per cycle is addressed by multiple rule firing and using macrorules; few affected productions per working memory change is addressed by creating more rules via copy/constrain and increasing the number of selection tests by using macrorules; balancing the load among processors is addressed by creating constrained copies of culprit rules.

## References

Aho A.V. and Ullman J.D. (1977) *Principles of Compiler Design.* Addison-Wesley Publishing Company: Reading, Massachusetts.

Earley J. (1975) High Level Iterators and a Method of Data Structure Choice. *Journal of Computer Languages 1(4)*: 321–342.

Fikes R.E. and Nilsson N.J. (1971) STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2: 189-208.

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

Georgeff M.P. (1982) Procedural Control in Production Systems. *Artificial Intelligence 18(2)*: 175-201.

Gupta A. (1985) *Parallelism in Production Systems: The Sources and Expected Speed-up.* Fifth International Workshop on Expert Systems and Applications.

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

Ishida T. and Stolfo S.J. (1984) *Simultaneous Firing of Production Rules on Tree-structured Machines.* Technical Report, Department of Computer Science, Columbia University.

Knuth

Kuck D.J., Budnik P.P., Chen S., Lawrie D.H., Towle R.A., Strebendt R.E., Davis E.W., Han J., Kraska P.W., and Muraoka Y. (1974) Measurements of Parallelism in Ordinary FORTRAN Programs. *Computer 7(1)*: 37-46.

Kuck D.J., Kuhn R.H., Leasure B., and Wolfe M. (1980) The Structure of an Advanced Retargetable Vectorizer. *The Proceedings of COMPSAC '80.*

McKeeman W.M. (1965) Peephole Optimization. *Communications of the ACM 8(7)*: 443-444.

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

Moldovan

Newell A. (1973) Production Systems: Models of Control Sructure. In W. G. Chase (ed.), *Visual Information Processing.* Academic Press: New York.

Newell A. and Simon H.A. (1972) *Human Problem Solving.* Prentice-Hall: Englewood Cliffs, New Jersey.

Nilsson N.J. (1971) *Problem-solving Methods in Artificial Intelligence.* McGraw Hill: New York.

Nilsson N.J. (1980) *Principles of Artificial Intelligence.* Tioga Publishing Company: Palo Alto, California.

Padua D.A., Kuck D.J., and Lawrie D.H. (1980) High Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers C-29(9)*: 763-766.

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

Pasik A.J. and Stolfo S.J. (1987) Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Rules. Technical Report, Department of Computer Science, Columbia University.

Post E.L. (1943) Formal Reductions of the General Combinatorial Decision Problem. *American Journal of Mathematics 65.*

Rosenblum P. and Newell A. (1982) *Learning by Chunking: Summary of a Task and a Model.* AAAI-82.

Rychener M.D. (1976) *Production Systems as a Programming Language for Artificial Intelligence Applications.* Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University.

Schwartz J.T. (1975a) Optimization of Very High Level Languages. Part I: Value Transmission and its Corollaries. *Journal of Computer Languages 1(2)*, 161–194.

Schwartz J.T. (1975b) Optimization of Very High Level Languages. Part II: Deducing Relationships of Inclusion and Membership. *Journal of Computer Languages 1(3)*, 197–218.

Stolfo S.J. (1979) *Automatic Discovery of Heuristics for Nondeterministic Programs from Sample Execution Traces.* Ph.D. Thesis, Courant Institute of Mathematical Sciences Computer Science Department, New York University.

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.

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.