From: AAAI-90 Proceedings. Copyright ©1990, AAAI (www.aaai.org). All rights reserved.

On the Performance of Lazy Matching in Production Systems

Daniel P. Miranker*, David A. Brant, Bernie Lofaso**, David Gadbois***

*Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

**Applied Research Laboratories
The University of Texas at Austin
P.O. Box 8029, Austin, TX 78713

ABSTRACT

Production systems are an established method for encoding knowledge in an expert system. The semantics of production system languages and the concomitant algorithms for their evaluation, RETE and TREAT, enumerate the set of rule instantiations and then apply a strategy that selects a single instantiation for firing. Often rule instantiations are calculated and never fired. In a sense, the time and space required to eagerly compute these unfired instantiations is wasted. This paper presents preliminary results about a new match technique, lazy matching. The lazy match algorithm folds the selection strategy into the search for instantiations, such that only one instantiation is computed per cycle. The algorithm improves the worst-case asymptotic space complexity of incremental matching. Moreover, empirical and analytic results demonstrate that lazy matching can substantially improve the execution time of production system programs.

1.0 Introduction

There is a large and growing body of research directed toward the integration of relational database and expert system technologies (Kerschberg 1987, Kerschberg 1988). Our work focuses on the problem of using the production system paradigm as the deductive component of an expert database system. The use of simple rules in databases is well known for enforcing integrity constraints and spontaneously triggering daemons if certain patterns appear in the data (Bunemann 1979, Astrhan 1976). The database problem of maintaining a view in the presence of updates to a database is very similar to the problem of incrementally evaluating the rules in a production system (Blakeley 1986). Even though some database systems incorporate a portion of the power of pattern directed inference systems, the number and form of the rules that can be effectively included in these systems is very limited. On the other side of the problem, expert systems that require information from existing databases do not access the data directly but maintain a small separate subset of the data by periodically issuing queries. Rule systems on the scale of an accredited expert system have not been tightly integrated with large databases. This is due to the extraordinary time and space demands one can expect from inferencing on large databases.

One of the fundamental issues is the exponential worst-case time and space requirements inherent in existing production system match algorithms (Raschid 1988, Miranker 1987, Forgy 1982). The worst-case asymptotic time and space requirements of both the RETE (Forgy 1982) and TREAT (Miranker 1987) match algorithms are O(wm^c) where wm is the size of the working memory and c is the maximum number of condition elements. While the average space requirements do not approach worst case, the variance in both time and space demonstrated over the life of a system is very volatile. Figure 1 shows rule firings (x-axis) versus number of instantiations (y-axis) for four OPS5 test applications[1]. These test programs have previously appeared in the literature (Gupta, Forgy & Newell1989, Miranker 198, Lofaso 1989). Some summary statistics about these systems are presented in Table 1. The erratic behavior seen in these graphs illustrates the time and space wasted in the eager evaluation of rules. Although the worst case is rarely achieved, it is clear from the graphs that very bad behavior may appear at any time. For database applications it is entirely possible for such algorithms to unexpectedly exhaust all of the available storage in large virtual memory computer systems (Bein, King & Kamel 1987).

TABLE 1. OPS5 Program Statistics

Program	Rules	Avg. WM Size	Inst-antiations	Rule Firings	Unused Inst-antiations	% Unused
WALTZ	33	42	151	70	81	54
TOURNEY	17	123	2324	528	1796	77
JIG25	6	50	205	58	147	72
WEAVER	637	152	1331	751	580	44

Therefore, we have developed an algorithmic basis for matching that is fundamentally better than current match algorithms in its space requirements. The first obstacle we observed is that all presently used algorithms for evaluating production systems enumerate the entire *conflict set*. The conflict set consists of rule instantiations where an instantiation is a rule name and an ordered set of working memory elements that satisfy that rule. The conflict set by itself has worst-case space complexity of O(wm^c). Thus, to

1. The applications used in our study are:
(a) JIG25 - solves a simple jigsaw puzzle,
(b) TOURNEY - schedules a bridge tournament,
(c) WALTZ - interprets three-dimensional line drawings, and
(d) WEAVER - routes a VLSI channel.

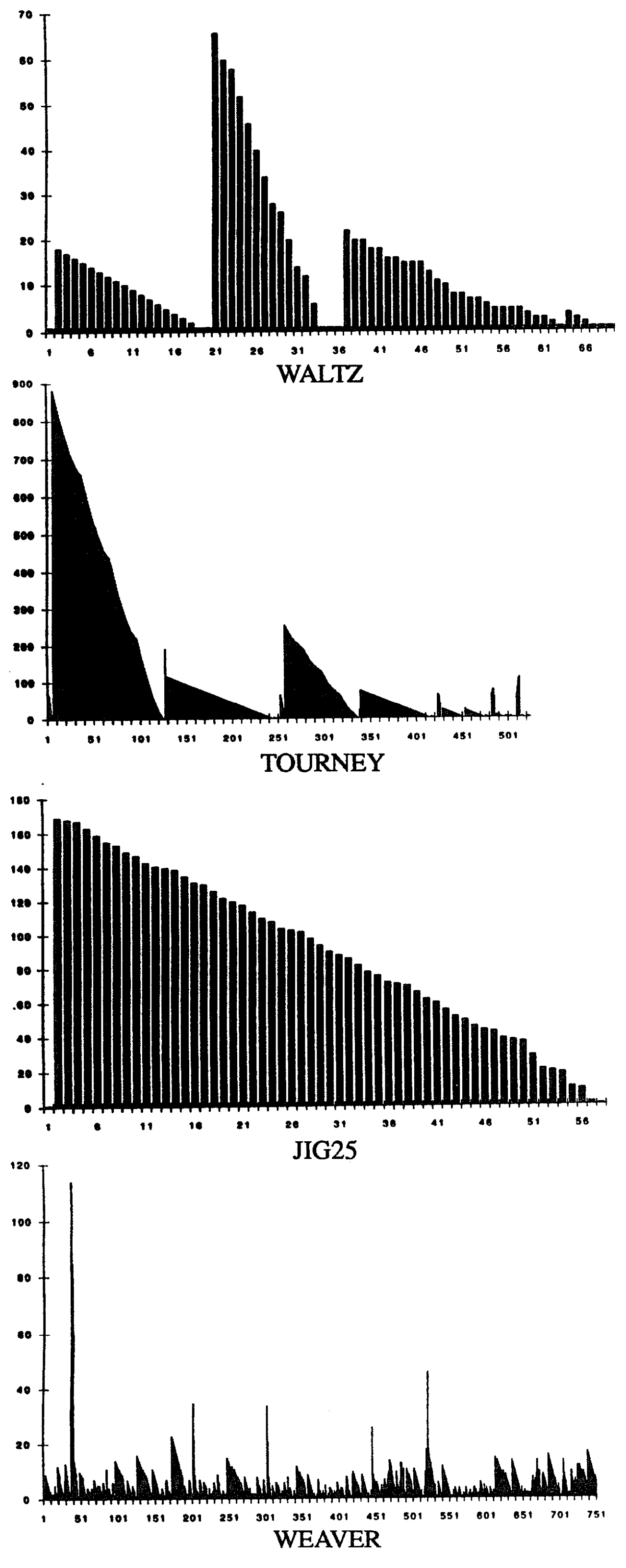

Figure 1. Conflict Set Instability

asymptotically improve the space complexity of the matching problem, it is necessary to avoid enumerating the conflict set. We have developed a new incremental match algorithm, the lazy match, that computes a single rule instantiation per cycle, yet may maintain the present execution semantics of existing production system languages. The lazy match is described in section 3 and has worst-case space complexity that is O(max(*ts*)**c*). Where *ts* is a timestamp and max(*ts*) is therefore bounded by the total number of updates to working memory. Further, it is often the case that instantiations are computed and placed in the conflict set and never fired (see Table 1). The lazy match never computes these "wasted" instantiations. Time is wasted not only in the eager computation of unused instantiations, but recent results have also shown that memory management is a dominant factor in the performance of these systems (Lofaso 1989). If memory requirements can be reduced then we might also expect performance to improve. Section 4 presents preliminary results of an OPS5 implementation based on the lazy match. These results show that the lazy match may substantially improve the performance of a production system program and eliminate the need to avoid certain troublesome constructs when writing rules. In section 2 we define production systems. Throughout the a paper we will draw examples using the OPS5 production system language. It is assumed that the reader is familiar with either the RETE of TREAT incremental match algorithms.

2.0 Production Systems and Eager Matching

In general, a production system is defined by a set of rules, or productions, that form the production memory, together with a database of current assertions, called the *working memory* (WM). Each production has two parts, the *left-hand side* (LHS) and the *right-hand side* (RHS). The LHS contains a conjunction of pattern elements, or *condition elements* (CEs), that are matched against the working memory. The RHS contains directives that update the working memory by adding or deleting facts, and directives that carry out external side effects such as I/O. In operation, a production system interpreter repeats the following recognize-act cycle:

1. Match. For each rule, compare the LHS against the current WM. Each subset of WM elements satisfying a rule's LHS is called an instantiation. All instantiations are enumerated to form the conflict set.
2. Select. From the conflict set, chose a subset of instantiations according to some predefined criteria. In practice a single instantiation is selected from the conflict set on the basis of the recency, specificity, and/or rule priority of the matched data in the WM.
3. Act. Execute the actions in the RHS of the rules indicated by the selected instantiations.

An OPS5 *working memory element* (WME) forms the user's conceptual view of an object and consists of a class name followed by a list of attribute-value pairs (Forgy 1981). A class name identifies an object and the attribute-value pairs describe a particular instance of that object. Each WME has a unique identifier (ID) associated with it.

IDs are often implemented as a strictly increasing sequence of integers assigned when the WME was created or last modified. They may be construed as timestamps or as logical pointers to individual WMEs. In most production systems, IDs are used in the conflict set resolution criteria. Consider the WME, shown below, used to describe a red cube named c_1, with a mass of 100, and having a length of 10 (attributes names are distinguished by a preceding ^ operator).

(cube ^name c_1 ^color red ^mass 100 ^len 10)

A production's LHS consists of a conjunction of CEs. It contains one or more non-negated CEs and zero or more negated CEs. Negated CEs are distinguished by a preceding negative sign. The LHS is said to be satisfied when:

1. for each non-negated CE, there exists at least one matching WME, and,
2. for all negated CEs, there do not exist any matching WMEs.

Each CE consists of a class name and one or more terms. Each term specifies an attribute within the class and a predicate to be evaluated against the values of that attribute. A CE need not reference all of the attributes contained in its corresponding class. The class is projected onto the named attributes in the CE. Those not named do not affect the match criteria. Predicates consist of a comparison operator ($<,>,=,\leq,$, or$\neq$) followed by a constant or variable. A predicate containing a constant is true with respect to a WME if the corresponding attribute value in the WME matches the predicate. For example, consider the CEs and corresponding WMEs shown in Fig. 2. CE (a) matches WMEs (1) and (3), while CE (b) matches only WME (1).

		"cube" WMEs			
CEs		name	color	mass	len
a) (cube ^mass<10)	1)	c_1	red	6	8
b) (cube ^mass<10 ^len>5)	2)	c_2	blue	11	5
	3)	c_3	red	1	3

Figure 2. Predicate Matching

The scope of a variable is the production in which it appears, and, therefore, all occurrences of a variable within a given LHS must be bound to the same value in working memory for the LHS to be satisfied. For condition elements containing variables, a mapping can be made to a relational join operation. The join operator will ensure that a given variable is consistently bound for all of its occurrences within a LHS.

2.1 Eager Matching

A critical component of any production system is the match algorithm that computes the instantiations. Currently used match algorithms are eager in nature. If a new WME is entered into the system they will perform a search for all instantiations containing that WME. These instantiations are then added to the conflict set (CS). Thus, from one rule firing to the next, the set of all valid instantiations is preserved by making incremental changes to the CS. In order to analyze the behavior of production systems and their match algorithms we have characterized them in terms of events that can change the conflict set and the operations performed by the system in response to those events. There are five events that may result in changes to the conflict set. Two may add instantiations. They are:[2]

1. make(WME^+) add a WME to a class corresponding to a non-negated CE
2. remove(WME^-) remove a WME from a class corresponding to a negated CE

Three events may remove instantiations from the CS. They are:

3. make(WME^-) add a WME to a class corresponding to a negated CE
4. remove(WME^+) remove a WME from a class corresponding to a non-negated CE
5. fire(I) fire instantiation I.

Current implementations of production systems perform several basic operations in response to these five events. For events that involve the computation of new instantiations,i.e., (1) and (2), the match algorithm effectively computes a relational database join for each rule containing the class associated with the specified WME. The WME is used as a seed to root the join and the join path branches out from the seed to the other classes of the CEs for that rule based on its join query graph. Nodes in the graph represent classes and arcs prescribe the join order. A failed search results in a backtrack to the previous class. The searches that succeed at the lowest level (leaf nodes of the query graph) indicate that an instantiation was found. A given instantiation can be represented by the timestamps of the WMEs along the path leading from the root to the leaf.

As new instantiations are produced the conflict set must be resorted according to the resolution strategy. Event (3) also results in a seed join, the results of which are removed from the CS. Event (4) produces a search of the conflict set for instantiations containing the specified WME, which are then removed. Event (5) simply removes the fired instantiation from the CS.

3.0 A Lazy Matching Algorithm

The following describes a method for computing production instantiations in a lazy manner. This is accomplished by executing a best-first search for instantiations. After one instantiation is found, the search pauses to allow the corresponding rule to be fired. Since the rule firing may change WM, the best-first search must be capable of responding to a dynamic search space. This is accomplished by maintaining a stack of best-first search pointers. As searches are superceded by changes to the WM, their state is pushed onto the stack. When a search is exhausted, the next set of pointers is removed from the stack. The top of

2. Note that these events do not necessarily have a one-to-one correspondence to the makes and removes specified in an OPS5 rule's RHS, e.g., a given WME may be applicable to many CEs and therefore an OPS5 "make" could result in numerous make(WME+) events.

stack always contains the state information for the next search.

The correctness of lazy match is dependent upon being able to enforce a total ordering in the generation of instantiations. If this is not done, duplicate instantiations may be computed and fired. If the total ordering is by timestamp (i.e. ID), a search heuristic based upon firing the production with the most recent instantiation (McDermott & Forgy 1978) can be employed. However, it is important to note that any total ordering of instantiations for a given rule will work. Adding additional criteria for instantiations of different rules, such as specificity and/or rule priority, can also be accommodated in a straightforward manner[3].

3.1 Conflict Set Resolution and Lazy Matching

The challenge of the lazy matching algorithm is in controlling a best-first search for instantiations through a WM that may change after each instantiation is found and fired. The criteria for "best" in this case is based upon the conflict set resolution strategies.

Lazy matching uses the selection strategy as an evaluating function to direct the search for a fireable instance. This is done by using that criteria to direct the search for matching WMEs from the *alpha-memories*[4]. On any given cycle, the search for an instantiation will stop after the first one is found, with the search being conducted so as to preserve recency. Of course, additions to, and deletions from, the WM will affect the search, and we must ensure that a given instantiation is fired at most once. To do so, state information is saved on a stack in order to continue the correct computation of instantiations.

3.2 Computing Instantiations Using Lazy Matching

Elements of the stack consist of a sets of pointers representing the state of a best-first search for instantiations. For convenience we use the timestamps of the WMEs to represent both an instantiation and the search state. For simplicity of presentation we assume a single rule system. We define an instantiation as a tuple containing one timestamp from each non-negated CE. Thus for a rule containing *n-1* non-negated CEs, stack entries and instantiations have the form «ts_0,...,ts_{n-1}», where ts_i is a timestamp. As each WME is entered into the an alpha-memory, a corresponding initial search state is pushed onto the stack. The initial state is «ts_0-1,...,ts_i,...,ts_{n-1}-1», where ts_i is the timestamp of the newly added WME.

The concept of a dominant timestamp (DT) is introduced to control the lazy computation of instantiations. For any stack entry, the DT is the most recent timestamp. Figure 4(a) shows the initial system state for the production appearing at the top of the figure. Note that the timestamp is denoted as the attribute "ts".

The computation of an instantiation begins with popping the top of stack and selecting the DT. This is followed by a best-first search for an instantiation rooted at the WME referenced by the DT. To ensure that instantiations are produced only once, alpha-memories have a fixed ordering (by timestamp in this example), and the best-first search computation restricts the WMEs joining with DT to

```
(P Example
 (C0 ^A<x>)
 (C1 ^A<x> ^B<y>)
 (C2 ^B<y>)
 ---> actions...)
```

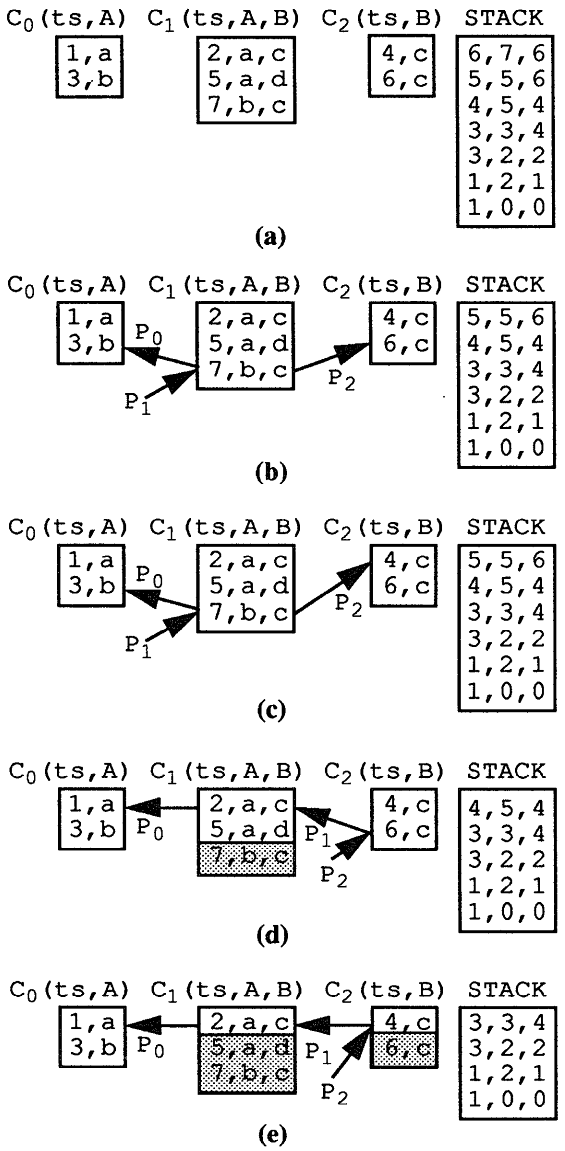

Figure 3. Lazy Computations of Instantiations

3. In general, the only form of conflict set resolution strategies that cannot be done lazily are those that demand an enumeration of the conflict set, e.g., fire the rule having the most instantiations.

4. Same as the alpha-memories described by Forgy and used in both RETE and TREAT.

those having timestamps less than DT. As soon as a matching set of WMEs (i.e., an instantiation) for DT is found, the computation pauses and the result is fired. If an instantiation containing DT cannot be found, then the next stack element is popped, and a new best-first search is begun. When an attempt is made to pop from an empty stack the system halts.

Figure 3(b) shows the initial state of the best-first search pointers (P_i) after the top stack entry has been popped and the corresponding search has found an instantiation. The best-first search is rooted at ts=7 in alpha-memory C_1 and proceeds outward in join order, most recent to least recent WME in each alpha-memory. Thus, for Fig. 3(b) the search state is «3,7,6». These WMEs satisfy the production and become the first instantiation. Next, the rule is fired, and, assuming for now that no WMEs are added to, or removed from, the WM by firing the rule, the search resumes to find the next instantiation. Figure 3(c) shows the state of the search after finding the next instantiation — «3,7,4». Before finding «3,7,4» the search would have tried «1,7,6», failed, backtracked, advanced the P_2 pointer, and succeeded (we arbitrarily chose to search the left alpha-memory first). The next time the search is performed «1,7,4» will be tried and will fail. That will exhaust the search rooted at the DT with ts=7. At that time the next stack entry is popped. In this case it is the WME with ts=6. The shaded area in Fig. 3(d) contains WMEs that have timestamps greater than that of the DT and therefore are not considered in the search. The next instantiation to be found is «1,2,6», after unsuccessfully trying «3,5,6», «1,5,6», and «3,2,6». After that, the stack is again popped with DT=5. Since no instantiations can be found for DT=5, another pop is performed and the WME with ts=4 is chosen as DT. The instantiation «1,2,4» is found (Fig. 3(e)) after trying «3,2,4». «1,2,4» is the final instantiation that can be produced. After it is fired, all the remaining stack entries are popped and their searches exhausted. Finally an attempt is made to pop from the empty stack and the system halts.

We now consider the effects of adding and deleting WMEs after each rule firing. When a new element is added to the WM a set of initial pointers is pushed onto the stack, but first, the current search is suspended and its state pushed onto the stack. That search may be resumed at a later time when it is popped off the top of stack. Since deletions may affect the state of a suspended search by removing WMEs that have pointers to them on the stack, each time a search state is popped from the stack, its pointers must be verified by the best-first search, backtracking if necessary. Figure 4(a) is the same as Fig. 3(b). Assume that the instantiation referenced by «3,7,6» fires and adds the WME <8,d> to C2. This causes

1. the search state «3,7,6» to be pushed to the stack,
2. «7,7,8» is pushed onto the stack and subsequently popped,
3. a best-first search is started with DT=8, and
4. the next instantiation is found, i.e., «1,5,8» (Fig. 4(b)).

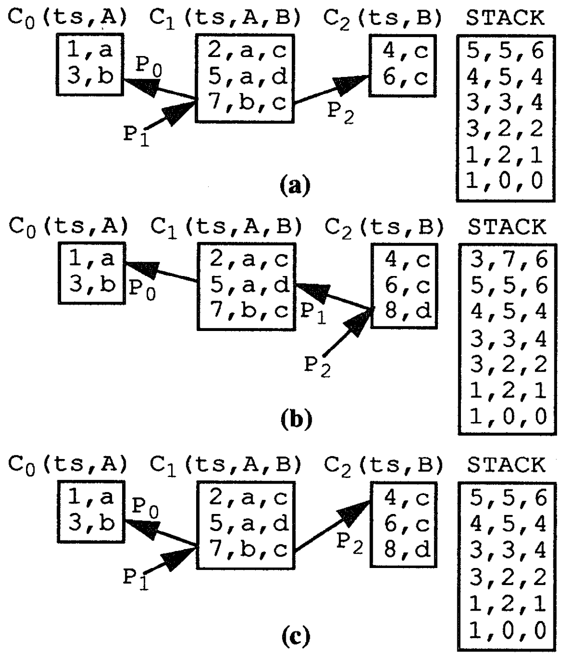

Figure 4. Dynamic Search Space

Assume that firing «1,5,8» does not change the WM. On the next cycle the search rooted at DT=8 will be exhausted and the top of stack popped. Thus the search that was suspended, «3,7,6», is resumed. The next instantiation found will be «3,7,4». The pseudocode in Fig. 5 should help elucidate the algorithm.

```
program Lazy Match;
p1,...,pn-1: WME timestamps;
begin
   initialize stack;
     {The top level makes that form the ini-
     tial WM are added here. An entry for
     each is placed on the stack.}
   loop while stack not empty
     pop_stack(p1,...,pn-1,empty);
     if not empty then
       best_first(p1,...,pn-1,found);
       if found then
         push_stack(p1,...,pn-1);
         fire(p1,...,pn-1);
   end loop;
end Lazy Match;
function pop_stack(p1,...,pn-1,empty);
   {If the stack is not empty it returns
   the top element and sets empty=FALSE,
   else empty=TRUE}
function best_first(p1,...,pn-1,found);
   {Performs a best-first search for an in-
   stantiation by working backwards from an
   ordered list of timestamps. The search
   first validates the pointers then searches
   using the DT as the root. If an instanti-
   ation is found then found=TRUE and the
   new instantiation is returned in the
   p1,...,pn-1, else found=FALSE.}
function push_stack(p1,...,pn-1);
   {Pushes the pointers onto the stack.}
function fire(p1,...,pn-1);
   {Fires the instantiation referenced by
   the p1,...,pn-1. This may alter the WM. A
   make will place an entry on top of the
   stack. A remove may delete an entry from
   the stack.}
```

Figure 5. Lazy Match Pseudocode.

There are pathological cases where the best-first search strategy will not produce the identical sequence of instantiations as OPS5. It is possible to avoid these cases by imposing a strict LEX ordering on lazy match, but doing so is computationally expensive. Nevertheless, the criteria used in lazy matching is in keeping with the general concept of recency as presented in (McDermott & Forgy1978), and has not yet posed a problem.

3.3 Handling Negated Condition Elements

We have discovered three different methods of lazily handling negated condition elements (NCEs). Only one will be described here. The methods for dealing with NCEs are closely related to the method developed for the TREAT match algorithm (Miranker 1987). If a search for an instantiation consistently binds with a WME that matches an NCE, then the search fails at that point and must backtrack. We say that that WME blocked the search. When a blocking WME is removed from the system, some instantiations may become unblocked and allowed to compete for firing. Those instantiations that become unblocked are those that would have been computed had the condition element been positive instead of negative, and had the WME been added to the system instead of removed.

To handle NCEs, for each negated condition add a second alpha-memory which will shadow the first. Rename the original alpha-memory from C_i to C_i^-. Call the shadow alpha-memory C_i^S. When a WME that has blocked a search is removed from a C_i^- alpha-memory it is inserted into C_i^S, given the next available timestamp, and an entry is pushed onto the stack. Note that this requires the stack to accommodate another timestamp in its elements, one for each shadow alpha-memory. The newly added WMEs to C_i^S can then be allowed to root a best-first search for those instantiations that they had blocked.

A problem arises when a search leads to an instantiation that has already been derived from a search rooted by a WME in C_i^S. This is solved by requiring best-first search to examine all of C_i^- and the portion of C_i^S such that a search that starts with a DT=ts_1 and binds consistently with a WME in C_i^S with timestamp ts_2>ts_1 fails. The idea is that once a WME enters CSi, only it may generate instantiations with older WMEs. Such a WME will be able to root the search for all instantiations older than itself, whether they were blocked or not.

We can now summarize the operations performed by lazy match and TREAT in response to the five conflict set events (see Table 2).

TABLE 2. Events and Operations

Event	TREAT Operation	Lazy Operation
make(WME^+)	Seed Join	Stack Push
remove(WME^-)	Seed Join	Stack Push
make(WME^-)	Seed Join & Delete from CS	None
remove(WME^+)	Delete from CS	None
fire(I)	Delete from CS	Best-first Search

4.0 Preliminary Results

4.1 Space and Time Complexity

Each alpha-memory is proportional to the size of the WM. In the most adversarial scenario, every WME can be added to a shadow-memory and never removed. Thus, in the worst-case the shadow-memories are bounded by the maximum timestamp and the worst-case space complexity of the lazy match is O(max(ts)*c). Although the worst-case space requirements for a nonterminating program based on this version of the lazy match are unbounded, the worst-case is very unlikely and the space requirements of the lazy match are not at all volatile.

We have identified several techniques that filter and reduce the size of the shadow-memories. The most aggressive of these filtering techniques bounds the size of the shadow memories to O(wm^v) where v is the number distinct positive condition elements needed to bind the variables in the shadow memory. This filter results in a worst-case space complexity of O(Min(wm^v, Max(ts))*c).

A simple filter, invoked when a rule becomes inactive, completely purges a rule's shadow memory. A rule is active when each of its positive alpha-memories contains at least one entry. The first filter is expensive, but effective. The second is very inexpensive, but for some rules in a nonterminating program it may never be invoked. Since the shadow memories must be searched as well as the negated alpha-memories, and since there is no analog of shadow memories in either RETE or TREAT, the actual execution time of the lazy match must be evaluated empirically.

4.2 Implementation

To evaluate the effectiveness and the trade-off's with respect to the variants of the lazy match we have reworked the back-end of the OPS5c compiler to use the lazy match. OPS5c is a portable C/Unix based OPS5 compiler originally based on the TREAT match algorithm (Miranker et al. 1990). OPS5c produces in-line matching code for each rule. Its target is C code which must then be compiled for the target machine.

The recently completed current version only implements the simple purging filter on the shadow memories. The above presentation of the lazy match considered the generation of an instantiation by the best-first search as a computation involving a single rule. The current implementation was extended to multiple rules by first selecting the DT and then considering each rule/alpha-memory that contained that DT in the order determined by the remaining conditions of the OPS5 lex strategy.

Table 3 shows the performance of the lazy match implementation with respect to OPS5c tests for three test programs. Lazy matching generally resulted in 2-3 times fewer WME tests.

TABLE 3. WME Tests

	WME Tests	
Program	TREAT	LAZY
JIG25	35,780	11,113
TOURNEY	1,107,259	513,600
WALTZ	23,890	14,967

OPS5c has reduced the match time for these programs to below the 90%. Therefore, speed-up may not be as high as WME tests might indicate. The test programs in Table 3 execute very quickly and do not provide a good measure of execution time. However, the The WALTZ program can be scaled up by inputting larger line drawings. The original data describe a drawing consisting of 18 line segments. To demonstrate scaling and the effectiveness of the algorithm on large problems, we gave it a 10,000 WME waltz problem. This resulted in a 4-fold reduction in the number of WME tests and reduced the run time by more than 50%.

These results are much better than we expected, especially when compared to the table of unused instantiations (Table 1) which we had thought was an optimistic measure of pruning. Detailed examination of the programs and their performance has revealed that lazy evaluation of certain programming constructs commonly used in rule systems can result in improved time complexity for the evaluation of those constructs.

We start with an illustrative example (see Fig. 6). The rule represents a naive one rule solution to a jigsaw puzzle problem. This rule is typical in structure of many of the rules in all systems we tested. This rule says, "compare all edges to all other edges and if two have the same shape place them next to each other". If there are n edges, then the TREAT algorithm will perform n^2 operations.

```
(p one-rule-jigsaw-solution
   (edge ^piece-id <pid1> ^edge-id <eid1>
     ^shape <s> ^matched F)
   (edge ^piece-id {<><pid1> <pid2>} ^edge-
     id <eid2> ^shape <s> ^matched F)
   -->
   (write "Place puzzle piece" <pid1> "next
     to piece" <pid2>")
   (modify 1 ^matched T)
   (modify 2 ^matched T))
```

Figure 6. Jigsaw Rule

The execution of the lazy match first picks an edge matching the first condition element and then takes an average of $n/2$ operations to find the matching piece. The rule would then fire and these two pieces would be removed from consideration. On the next cycle the lazy match would again pick an edge matching the first condition element and then take an average of $(n-2)/2$ operations to find the matching piece. Computing the sum, from n until all pieces are exhausted, for this rule shows that the lazy match would execute $(n^2+2n)/8$ operations.

We can generalize this type of problem to rules that pick loosely restricted subsets of size k from n objects, where loosely restricted means that once $j<k$ objects are chosen it will always be possible to fill the requirements for the $j+1$ object without backtracking. An eager evaluation of such a rule requires $O(n^k)$ time. A lazy evaluation will take $O(n)$ time to pick each of k objects or $O(n*k)$. Many systems, as in the jigsaw puzzle, will fire such a rule until all n of the objects have been chosen forming n/k subsets.

Theorem:
Given n WM elements. To choose n/k disjoint subsets of size k by executing n/k cycles an eager evaluation will take $O(n^k)$ operations. A lazy evaluation will take $O(n^2)$.

Proof:
Let σ be the join selectivity. Join selectivity is the probability that the values tested for a WME will be consistent with the values bound up to that point. For example, for the jigsaw puzzle rule if the shape of each edge is unique then $\sigma = 1/n$.

Eagerly evaluating a rule with k conjuncts and having each alpha-memory having n elements takes:

$$\sum_{i=1}^{k} \sigma^{i-1} n^i = O(n^k) \qquad \text{(EQ 1)}$$

A Lazy evaluation takes:

$$\sum_{i=0}^{(n/k)-1} \frac{n-ik}{\sigma n+1}(k-1) = O(n^2) \qquad \text{(EQ 2)}$$

Notice the constants greatly favor lazy evaluation.

How common are such rules? Rules that are completely of this type are probably not that common. But there are rules in nearly all systems which pairs or triple of CEs represent one of the above constructs. Any rule in any program that refers to the same class in more than one CE is a candidate for this reduction. We have found rules of this form in WEAVER,TOURNEY, and WALTZ. In WEAVER there are many rules where 5 or more condition elements refer to the same class. Our conjecture is that, using lazy matching, there are many rules in most systems whose time complexity will improve by one or more degrees.

5.0 Conclusions and Current Work

The idea of lazy matching is necessary to improve the asymptotic space complexity of the incremental match problem. Preliminary results show that for several application programs Lazy matching substantially improves execution time as well as the space requirements. Investigation of the applications revealed that Lazily matching certain commonly used and expensive rule constructs leads to asymptotic improvement in the execution time of those rules. In the near future we will consider rule-parallel implementations that compute one instantiation per rule. We will also investigate other filtering techniques for the shadow memories, including a technique that eliminates shadow memories completely but whose worst-case space complexity is

O(Max(wm^v, ts)*c), and whose average space requirements are potentially volatile.

Our current goals include the development of an integrated expert-database system. By an integrated expert-database system we mean a system where working memory encompasses a large disk resident database and conventional database transactions may occur concurrently with the inferencing tasks. As a prototype, we are integrating the OPS5c compiler and the Genesis extensible database management system (Batory et al. 1988). We are exploring the use of appropriate data structures and memory hierarchy to support this type of system. This research is being conducted in the context of the behavior of the Lazy match as the size of working memory is scaled to the size typical of existing commercial databases.

REFERENCES

Astrhan, M., et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976.

Batory, D., et. al., "GENESIS: An Extensible Database Management System," IEEE Transactions on Software Engineering, Nov., 1988.

Bein, J., R. King, and N. Kamel, "MOBY: An Architecture for Distributed Expert Database Systems," Proceedings of the 13th VLDB Conference, Brighton, 1987.

Blakeley, J. A., et. al., "Efficiently Updating Materialized Views," Proceedings of the 1986 ACM-SIGMOD International Conference on Management of Data, Washington, DC, June 1986.

Bunemann, P. and E. Clemons, "Efficiently Monitoring Relational Data Bases," ACM-TODS, Sept. 1979.

Forgy, C., "OPS5 User's Manual", Tech Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

Forgy, C., "RETE: A Fast Match Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, no. 19, pp. 17-37, 1982.

Gupta, A., C. Forgy, and A. Newell, "High-Speed Implementations of Rule-Based Systems," ACM TOCS, June, 1989.

Kerschberg, L., "Proceedings of the First International Conference on Expert Database Systems", Benjamin/Cummings Publishing Company, Inc.,Menlo Park CA, 1987.

Kerschberg, L., "Proceedings of the Second International Conference on Expert Database Systems", Benjamin/Cummings Publishing Company, Inc.,Menlo Park CA, 1988.

Lofaso, B. J., "Join Optimization in a Compiled OPS5 Environment," Tech. Report No. ARL-TR-89-19, Applied Research Laboratories, The University of Texas at Austin, April, 1989.

McDermott J., and C. Forgy, "Production System Conflict Resolution Strategies," In Pattern-directed Inference Systems, D. Waterman and F. Hayes-Roth (eds.), Academic Press, 1978.

Miranker, D., "TREAT: A Better Match Algorithm for AI Production Systems," Proceedings of the 1987 National Conference on Artificial Intelligence, Seattle, 1987.

Miranker, D., "TREAT:A New and Efficient Match Algorithm for AI Production Systems", Pittman/Morgan Kaufman, 1989.

Miranker, D., B.J. Lofaso, G. Farmer, A. Chandra, and D. Brant, "On a TREAT Based Production System Compiler", Proceedings of the 10th International Conference on Expert Systems, Avignon, France, 1990.

Raschid, L., T. Sellis, and C-C Lin, "Exploiting Concurrency in a DBMS Implementation for Production Systems," Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, 1988.