Collection Oriented Match

Anurag Acharya and Milind Tambe
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
{acharya, tambe}@cs.cmu.edu

Abstract

Match algorithms that are capable of handling large amounts of data, without giving up expressiveness are a key requirement for successful integration of relational database systems and powerful rule-based systems. Algorithms that have been used for database rule systems have usually been unable to support large and complex rule sets, while the algorithms that have been used for rule-based expert systems do not scale well with increasing amounts of data. Furthermore, these algorithms do not provide support for collection (or set) oriented production languages.

This paper proposes a basic shift in the nature of match algorithms: from *tuple-oriented* to *collection-oriented*. A collection-oriented match algorithm matches each condition in a production with a collection of tuples and generates *collection-oriented instantiations*, i.e., instantiations that have collection of tuples corresponding to each condition in the production. This approach shows great promise for efficiently matching expressive productions against large amounts of data. In addition, it provides direct support for collection-oriented production languages. We have found that many existing tuple-oriented match algorithms can be easily transformed to their collection-oriented analogues. This paper presents the transformation of Rete to Collection Rete as an example and compares the two based on a set of benchmarks. Results presented in this paper show that, for large amounts of data, a relatively underoptimized implementation of Collection Rete achieves orders of magnitude improvement in time and space over an optimized version of Rete. The results establish the feasibility of collection-oriented match for integrated database-production systems.

1. Introduction

The integration of relational database systems and production systems (forward-chaining rule systems) promises a number of benefits. Production rules have long been known to be a natural mechanism for enforcing integrity constraints, performing authorization checks and maintaining *views* (derived data) [10]. They have also been used to implement *alerters* that monitor conditions and *triggers* that conditionally initiate actions [8, 29]. A number of prototype relational database systems — STARBURST [35], POSTGRES [29], ARIEL [17] and RPL [9] — use production rules for some or all of these purposes. Commercial relational database systems like INGRES [18] and Sybase [30] also support production systems. Most of these systems use fairly simple match algorithms to determine the set of production instantiations (ARIEL is an exception). This effectively limits the complexity of the rules that can be efficiently supported in the presence of large amounts of data [29]. Typical rules supported in these systems have a small number of simple tests. Supporting powerful rules in a database environment requires match algorithms that can efficiently handle complex tests in the presence of large amounts of data.

On the other hand, expert systems have long supported powerful rules. These systems have traditionally used more powerful match algorithms like Rete and its derivatives [4, 11] and Treat [22]. These algorithms, however, have not been designed for matching large amounts of data and do not scale well [24, 34]. This limits the the amount of data that expert systems can deal with and forces the expert systems that have been coupled with database systems to either use only simple rules or maintain a small separate subset of data by periodically issuing queries [24]. Extending the scope of expert systems to include data-intensive tasks, without giving up expressiveness, requires powerful match algorithms that can efficiently match large amounts of data.

The question then is: can production match algorithms support a large number of powerful match operations and yet scale well with increasing amounts of data ? In this paper, we attempt to answer this question by investigating an approach that avoids the major limitation on the scalability of traditional match algorithms.

The primary reason why traditional match algorithms used in expert systems don't scale well is that they generate a large number of combinations of individual tuples [14, 24, 32]. These combinations are generated during the match procedure as intermediate results and as production instantiations. The total number of such combinations can be a high order polynomial function of the number of tuples [24], which leads to a combinatorial explosion as the number of tuples grows. For the sake of efficiency, almost all existing match algorithms that are capable of matching powerful rules maintain some of these combinations as intermediate *match state* [6, 15].

Research efforts at developing match algorithms with better scaling characteristics have focussed either on maintaining less state [6, 17] or on efficiently maintaining the state on secondary storage [5, 27, 34]. These algorithms retain the *tuple-oriented* nature of the traditional algorithms. That is, they match individual tuples, and generate combinations of individual tuples for intermediate results and instantiations.

We take a different approach. We propose a basic shift in the nature of the match process: from tuple-oriented to *collection-oriented*. In collection-oriented match, collections of tuples that match individual conditions are the unit of matching, rather than individual tuples. The intermediate results and instantiations of collection-

oriented match are combinations of these collections, instead of combinations of individual tuples. For large amounts of data, the number of tuples that match individual conditions is likely to be large. In such situations, the number of collections will be much smaller than the number of tuples. This allows collection-oriented match to tame the combinatorial explosion, since it generates combinations of collections instead of combinations of tuples. Thus, it promises substantial improvements in space and time requirements over tuple-oriented match algorithms without giving up expressiveness.

Collection-oriented match also addresses another problem with tuple-oriented match algorithms. These algorithms do not provide efficient support for collection-oriented production languages. Collection-oriented production languages are arguably more suitable for integration with relational databases than tuple-oriented languages, since the unit of operation in relational databases is a relation rather than a single tuple. Collection-oriented languages also make it possible to specify aggregate operations like count, sum, statistical operations like mean and variance, data-fitting operations etc which are important for database-based tasks [9, 13, 35]. While such operations *can* be expressed in tuple-oriented languages, they cannot take advantage of optimized procedures for these operations. Collection-oriented match provides direct support for collection-oriented production languages. In addition, if needed, it can also be used to implement a tuple-oriented production language such as OPS5 [7]. The tuple-oriented semantics of languages would, however, limit the gains that would be achieved.

We have found that many of the tuple-oriented match algorithms can be easily transformed to their collection-oriented analogues. To illustrate this, we describe the transformation of Rete to *Collection Rete*, its collection-oriented analogue. To investigate the efficiency and scalability of Collection Rete, we used it to implement a collection-oriented extension of OPS5, called COPL. We compared the performance of this implementation with that of an optimized Rete-based OPS5 implementation on a set of scalable benchmarks. Results show that, on these benchmarks, a relatively underoptimized implementation of Collection Rete achieves between one and four orders of magnitude improvement in time *and* up to one order of magnitude improvement in space over an optimized Rete implementation.

These results clearly illustrate the efficiency and scaling characteristics of tuple-oriented and collection-oriented match approaches. However, more importantly, we view them as establishing the feasibility of collection-oriented match for matching large amounts of data and, therefore, for its use in integrated database-production systems. In several of the experiments, the Collection Rete implementation (with its limitations) matched over a million tuples within a reasonable time period. To the best of our knowledge, this is approximately two orders of magnitude larger than the largest working memory previously dealt with. Clearly, these results are specific to our benchmarks and more research on collection-oriented match algorithms will be required before such large working memories can be routinely dealt with.

The rest of this paper is organized as follows. Section 2 introduces OPS5 and the terminology that we will use in the rest of the paper. Section 3 presents collection-oriented match, and describes how it supports collection-oriented production languages. Section 4 describes Rete, and its transformation into Collection Rete. Section 5 describes our benchmarks, experimental methodology, results and analyses. Section 6 addresses the issue of validity of these results. Section 7 discusses related work and the implication of the collection-oriented approach for other match algorithms. Finally, Section 8 presents conclusions and discusses issues for future work.

2. Background

Various languages have been proposed for integrated database-production systems. Many researchers have focused on using OPS5 or OPS5-style production system languages for this integration [6, 9, 13, 17, 28]. This section introduces OPS5 terminology using the simple production system in Figure 2-1.

Figure 2-1-a shows the working memory in the system, essentially a relational database. The working memory contains nine tuples (or working memory elements): W1,W2...W9. The symbols GOAL and EMPLOYEE are called the classes of the tuples, and correspond to relations. The up-arrows (^) in the tuples indicate attribute names, and correspond to the fields in a relation. These tuples are to be matched with the production MAKE-TEAM, shown in Figure 2-1-b. The production has three conditions on its condition-side or LHS, and one action on its action-side or RHS. The symbols in the conditions are either constants, e.g., HARDWARE, that test if these constants appear in specific fields of the tuples, or variables (enclosed in <>) that bind to values appearing in identical fields in the tuples. The production MAKE-TEAMS teams up a pair of employees, who have worked together on a previous project, but have different areas of expertise. The *make* command on the action side actually creates a new tuple of class TEAM that includes the two members.

```
W1: (GOAL  ^TYPE CREATE-TEAM)

W2: (EMPLOYEE  ^NAME A  ^PREVIOUS-PROJECT WARP  ^EXPERTISE HARDWARE)
W3: (EMPLOYEE  ^NAME B  ^PREVIOUS-PROJECT WARP  ^EXPERTISE HARDWARE)
W4: (EMPLOYEE  ^NAME C  ^PREVIOUS-PROJECT PSM  ^EXPERTISE HARDWARE)
W5: (EMPLOYEE  ^NAME D  ^PREVIOUS-PROJECT PSM  ^EXPERTISE HARDWARE)

W6: (EMPLOYEE  ^NAME E  ^PREVIOUS-PROJECT WARP  ^EXPERTISE COMPILERS)
W7: (EMPLOYEE  ^NAME F  ^PREVIOUS-PROJECT WARP  ^EXPERTISE COMPILERS)
W8: (EMPLOYEE  ^NAME G  ^PREVIOUS-PROJECT PSM  ^EXPERTISE COMPILERS)
W9: (EMPLOYEE  ^NAME H  ^PREVIOUS-PROJECT PSM  ^EXPERTISE COMPILERS)
```

(a)

```
(PRODUCTION  MAKE-TEAM
     (GOAL  ^NAME CREATE-TEAM)
     (EMPLOYEE  ^NAME <N1>  ^PREVIOUS-PROJECT <P>  ^EXPERTISE HARDWARE)
     (EMPLOYEE  ^NAME <N2>  ^PREVIOUS-PROJECT <P>  ^EXPERTISE COMPILERS)
 -->
     (MAKE  TEAM  ^FIRST-MEMBER <N1>  ^SECOND-MEMBER <N2>))
```

(b)

Figure 2-1: A simple production system: (a) Working memory, and (b) A production.

Tuple-oriented match in a production system involves

finding all possible tuple-oriented instantiations of the production given the tuples. A tuple-oriented instantiation is a combination of tuples that provide consistent bindings for the variables in the production. In Figure 2-1-a, the instantiation (W1, W2, W6) provides one such consistent binding WARP for the variable <P>. Seven other production instantiations are also generated: (W1, W3, W6), (W1, W2, W7), (W1, W3, W7), (W1, W4, W8), (W1, W5, W8), (W1, W4, W9), and (W1, W5, W9). When instantiations *fire*, action side is executed in the context of its variable bindings, updating the working memory of the system.

3. Collection-Oriented Match

Collection-oriented match treats collections of tuples, rather than individual tuples as the primary objects to be matched. A collection-oriented match algorithm matches each condition in the productions with a collection of tuples and generates *collection-oriented instantiations*, instantiations that have collection of tuples corresponding to each condition in the production. All tuples in the collections are *guaranteed to be mutually consistent*. The following example clarifies this point.

Consider the production system shown in Figure 2-1. Here, collection-oriented match results in two collection-oriented instantiations. The first is ({W1}, {W2,W3}, {W6,W7}). Here, {W1} matches the first condition, {W2, W3} matches the second condition and {W6, W7} matches the third condition. The tuples in these three collections are mutually consistent with each other, i.e. they have consistent values for the variable <P> — W1 is consistent with W6 and W7, W2 is consistent with W6 and W7, and so on. Similarly, the second collection-oriented instantiation is ({W1}, {W4,W5}, {W8,W9}).

A comparison of these collection-oriented instantiations with the tuple-oriented instantiations presented earlier illustrates two useful points. First, the two types of instantiations contain identical information about consistency of matching tuples. Thus, the tuple-oriented instantiations can be easily generated from collection-oriented instantiations by creating a cross product of its component collections. For instance, a cross product of {W1}, {W2,W3}, {W6,W7}, generates the first four tuple-oriented instantiations.

Second, the comparison illustrates the source of execution space and time efficiency in collection-oriented match. As mentioned earlier, the primary cause of high space and time costs in tuple-oriented algorithms is the generation of a large number of combinations of individual tuples. Collection-oriented match cuts down on the number of such combinations. For instance, suppose each of the three collections in one of the collection-oriented instantiations above contained N elements — the instantiation would consume (N + N + N =) O(N) space. On the other hand, tuple-oriented match would create a cross product of ($N \times N \times N$) tuple combinations as its instantiations; and consume $O(N^3)$ space. If these instantiations are maintained as part of the match state, as they are in many match algorithms, then the space savings from collection-oriented match will be $O(N^3)$. For a production with K conditions, the asymptotic savings are $O(N^K)$. Actual savings are even greater for algorithms that maintain intermediate products — since the constant factor is larger for them.

A reduction in the number of combinations, either intermediate products or instantiations, leads to corresponding speedups in execution time. For instance, avoiding the generation of $O(N^K)$ instantiations will lead to correspondingly large speedups. Furthermore, the overheads of updating and maintaining combinations as part of the match state are also reduced dramatically.

The preceding arguments establish two factors as influencing the speedup and match state reduction in collection-oriented match: (i) the size of the collections that match individual conditions (N in the preceding paragraph), and (ii) the number of conditions in the productions (K in the preceding paragraph). A third such factor is the amount of *fragmentation* in the component collections. The example in Figure 2-1 shows a simple case of fragmentation. Suppose all of the EMPLOYEE tuples in the figure had an identical value for the PREVIOUS-PROJECT field, say MACH. Then matching the production MAKE-TEAM using collection-oriented match would have resulted in a single collection-oriented instantiation ({W1}, {W2,W3,W4,W5}, {W6,W7,W8,W9}). However, the EMPLOYEE tuples in Figure 2-1-a have two different values for the PREVIOUS-PROJECT field. Therefore, two different collection-oriented instantiations, with smaller component collections, are formed. It is as though the collection-oriented instantiation with the larger collections has fragmented. In general, fragmentation may lead to the formation of many collection-oriented instantiations with smaller component collections. This increases both space and time requirements since more match state has to be generated and maintained. However, fragmentation occurs because of the necessity to maintain consistency between collections. It is thus a feature of the program and not the implementation.

Collection-oriented match does not improve the worst-case space and time complexity of production match. It is still possible to encode NP-complete problems such as hamiltonian circuit or subgraph isomorphism within the match of a single production.

3.1. Collection-oriented languages

As discussed earlier, tuple-oriented instantiations can be easily generated, as needed, from the collection-oriented instantiations. This allows collection-oriented match to function simply as an efficient match implementation for tuple-oriented production systems such as OPS5.

More importantly, collection-oriented match provides direct and efficient implementation support for collection-oriented production languages [13, 35]. At the core of such languages is the capability to directly manipulate collections (or sets) as single entities, instead of manipulating them on an element by element basis. These languages directly support collection-oriented operations such as counting, mean, variance etc. In collection-oriented match, a single instantiation packages together collections

that are consistent with each other. This allows the action side of a production to be executed in the context of values from a collection of tuples, rather than values from an individual tuple. For instance, in Figure 3-1, <EMP> will be bound to the entire collection of EMPLOYEEs with an expertise in COMPILERS. Thus, given the working memory from Figure 2-1-b, <EMP> will be bound to {W6,W7,W8,W9}. The function CARDINALITY on the action side then counts these employees, and creates the tuple (COMPILER-EXPERTS ^COUNT 4).

```
(PRODUCTION  COUNT-COMPILER-EXPERTS
     (GOAL  ^TYPE COUNT-COMPILER-EXPERTS)
   { (EMPLOYEE  ^NAME <X>  ^EXPERTISE COMPILERS) <EMP> }
  -->
     (MAKE COMPILER-EXPERTS  ^COUNT (CARDINALITY <EMP>) ))
```

Figure 3-1: A simple example of a collection-oriented acti

Another language-related issue is the role of negated conditions in production systems. Negated conditions are commonly used to restrict the match and sequence the firing of instantiations. In many cases, the collection-oriented approach obviates such usage. A example is iterating over a collection of tuples. Figure 3-2 shows how the counting operation performed by the collection-oriented production in Figure 3-1 would be done in a tuple-oriented language.

```
(PRODUCTION  COUNT-COMPILER-EXPERTS
     (GOAL  ^TYPE COUNT-COMPILER-EXPERTS)
     { (EMPLOYEE  ^NAME <X>  ^EXPERTISE COMPILERS)  <EMP> }
     { (COMPILER-EXPERTS  ^COUNT <VAL>)  <C> }
     - (EMPLOYEE  ^NAME <X>  ^EXPERTISE COMPILERS  ^COUNTED YES)
  -->
     (MODIFY <C>  ^COUNT (COMPUTE <VAL> + 1))
     (MODIFY <EMP>  ^COUNTED YES))
```

Figure 3-2: Counting in a tuple-oriented production syste

We have developed a language called COPL (Collection-Oriented Production Language) with collection-oriented semantics. COPL extends OPS5 in three ways. First, the instantiations of COPL productions are collection-oriented, i.e., the variables are bound to collections of values instead of individual values. Second, COPL actions are collection-oriented. For instance, the *make* action creates a collection of tuples instead of a single tuple, the *remove* action removes a collection of tuples, and so on. Third, as part of its actions, COPL supports calls to functions that operate on collections of values. These functions perform collection-oriented operations (such as count, sum, etc.) or element-wise operations, and return collections of values. Work on COPL is currently in preliminary stages, and many issues need to be resolved. Nonetheless, we have used this version of COPL for our experiments.

4. Transforming a Match Algorithm

To obtain a concrete basis for investigating collection-oriented match, we elected to transform the Rete match algorithm [11] to its collection-oriented analogue: Collection Rete. The decision to transform Rete was based on three reasons. First, Rete is the most commonly used algorithm in production system implementations. Second, Rete is intended for systems with a relatively slow rate of change of tuples. Database systems are expected to have a slow rate of change. Third, an implementation of the Rete algorithm was available to us. (As Section 7 shows, other algorithms can be similarly transformed.) In order to understand Collection-Rete, it is first useful to understand Rete itself. Section 4.1 describes Rete; subsequently, Section 4.2 will describe Collection Rete.

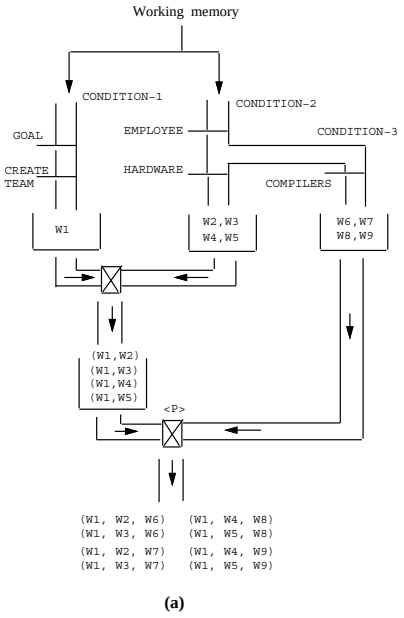
4.1. Rete

Rete employs two main optimizations: (i) it maintains match state from previous computations and (ii) it shares common parts of conditions in a single production or across productions to reduce match effort. We will use the simple production system shown in Figure 2-1 to explain Rete. In Figure 2-1, tuples W1,W2...W9 are to be matched with the production MAKE-TEAM. Rete's operation in matching this production can be understood using the analogy of water-flow through pipes. As shown in the upper part of Figure 4-1-a, each condition of the production can be considered as a pipe, ending in a bucket. The tuples flow through these pipes. Each pipe has filters associated with it, which correspond to the constant tests in the condition, and allow only particular tuples to pass through. For example, the filters in the first pipe (corresponding to the first condition) check if the tuple is of class GOAL, and has the value CREATE-TEAM for its TYPE field. Therefore, only W1 passes through the first pipe and appears in the bucket for the first pipe. Note that since the filter EMPLOYEE for the second and third pipes tests an identical field of the tuples, this filter is shared between the two pipes, illustrating the sharing optimization.

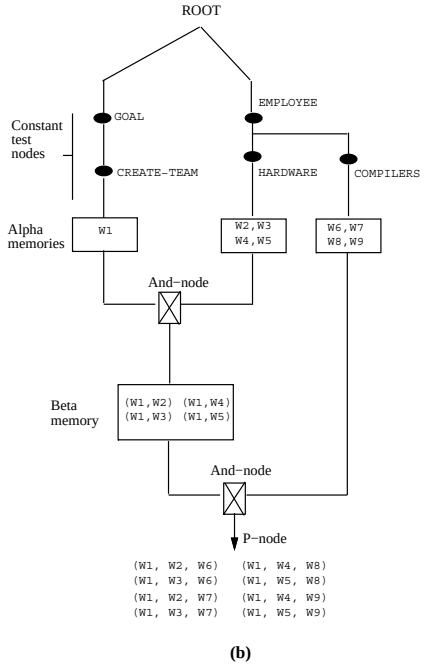
Next, the small boxes with Xs inside them check for consistency between tuples. Since there is no variable test between the first two conditions, the box that joins the first two conditions does not perform any consistency checks. The tuple W1 is therefore found consistent with each of the tuples W2,W3,W4, and W5. This leads to the creation of the four tuple combinations: (W1,W2), (W1,W3), (W1,W4) and (W1,W5), which are stored in the following bucket. Now the second small box checks the consistency of each of these four combinations against the tuples matching the third condition: W6, W7, W8 and W9. This involves testing the consistency of bindings for variable <P>. Eight different combinations of tuples (W1,W2,W6),...(W1,W5,W9) succeed and form instantiations of this production. These instantiations are stored in an *instantiation set.*

The buckets and the instantiation set in Figure 4-1-b contain the match state of Rete. If a new tuple, say W10, is added to the system, then only W10 will be matched; Rete will avoid re-matching W1,W2...,W9. The penalty for maintaining this state is that if a tuple is deleted, it has to be deleted from all the combinations that contain it.

In a Rete implementation, the productions are compiled to a dataflow network as shown in Figure 4-1-b. Tuples travel down from the ROOT node. The filters that test for constants are called constant test nodes. The buckets that



(a)



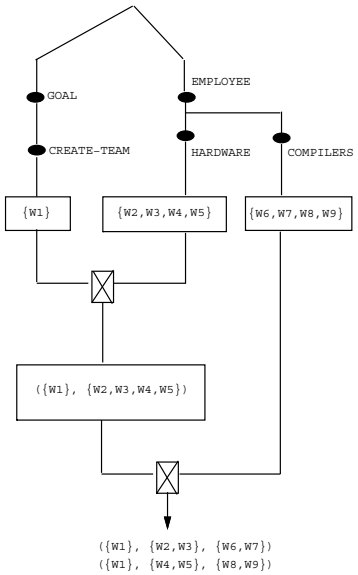
(b)

Figure 4-1: Rete algorithm: (a) Analogy of water-flow through pipes, and (b) Dataflow network.

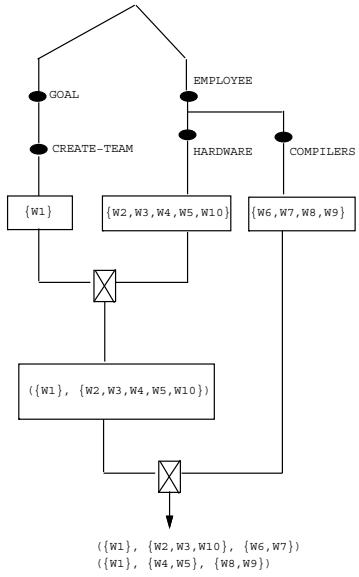
store individual tuples are called alpha memories. The tuples stored in alpha memories are called *right tokens*. Combinations of tuples, e.g., (W1, W4), are called *left tokens* and are stored in beta memories. And-nodes perform consistency-checks, while P-nodes add (and delete) instantiations to the instantiation set. The tuple combinations mentioned as the cause of the poor scalability of tuple-oriented algorithms are the left tokens and the instantiations.

4.2. Collection Rete

We will use the example in Figure 2-1 to describe Collection Rete. Figure 4-2-a shows the transformation of the Rete from Figure 4-1 into Collection Rete. (For the sake of brevity of this paper, presentation of the detailed algorithm, including description of how the algorithm deals with negated conditions, has been deferred to [3].)



(a)



(b)

Figure 4-2: Collection Rete: (a) Transformation of Figure 4-1, and (b) Adding a tuple W10.

In Collection Rete, each condition matches a collection of tuples. Therefore, alpha memories in Collection Rete store collections of tuples that match particular conditions. The first alpha memory stores {W1}. Since there is no

consistency test between the first two conditions, {W1} is found consistent with the collection {W2,W3,W4,W5} from the second alpha memory, forming a single *left collection-token* ({W1}, {W2,W3,W4,W5}). In forming such a token, two constraints are observed. First, all the tuples in the component collections of the token are guaranteed to be mutually consistent. Second, to obtain maximum benefit from collection-oriented match, largest possible left collection-tokens are formed. For instance, two separate left collection-tokens ({W1}, {W2,W3}) and ({W1}, {W4,W5}) could potentially be formed instead of the single ({W1}, {W2,W3,W4,W5}), but this fragmentation has been avoided.

At this point, the left collection-token ({W1}, {W2,W3,W4,W5}) formed is stored in the beta memory as shown. It is then checked for consistency with {W6,W7,W8,W9} in the third alpha memory to form new tokens. Given the consistency test for variable <P> at this point, forming left collection-tokens becomes more complex. A simple method for forming such tokens is as follows. The left collection-token is first sequentially compared with each element of the collection in the third alpha memory — W6,W7,W8, and W9. During this process, a subpart ({W1}, {W2,W3}) of the left collection-token is found consistent with the tuple W6, forming a new left collection-token ({W1}, {W2,W3}, {W6}). The subpart is also found consistent with the tuple W7, forming ({W1}, {W2,W3}, {W7}). Two other new left collection tokens are also formed: ({W1}, {W4,W5}, {W8}) and ({W1}, {W4,W5}, {W9}).

While these four newly formed left collection-tokens are consistent, they do not contain the largest collections possible. For this, a *merging* step is required. This step maintains the consistency of component collections. For two tokens to be merged, they must differ in exactly one component. For instance, ({W1}, {W2,W3}, {W6}) and ({W1}, {W2,W3}, {W7}) can be merged together to form ({W1}, {W2,W3}, {W6,W7}). Similarly, the remaining two tokens may be merged to form: ({W1}, {W4,W5}, {W8,W9}). After this, no further merging can take place. Since this is the final condition of the production, the two tokens ({W1}, {W2,W3}, {W6,W7}) and ({W1}, {W4,W5}, {W8,W9}) are sent to the P-node as collection-oriented instantiations.

Figure 4-2-b shows the operation of Collection Rete when a new tuple W10 of the form (EMPLOYEE ^NAME I ^PREVIOUS-PROJECT WARP ^EXPERTISE HARDWARE) is added to the working memory. W10 becomes a member of the collection matching the second alpha memory. As in Rete, W10 is then checked for consistency with the contents of the previous alpha memory. Since there is no consistency test, a left collection-token ({W1}, {W10}) is formed. This token is then merged with the token in the following beta memory to form ({W1}, {W2,W3,W4,W5,W10}). While ({W1}, {W10}) is merged, it is also compared with the collection in the third alpha memory. The token ({W1}, {W10}, {W6,W7}) results from this comparison. This token is merged with an existing collection-oriented instantiation that differs in only one slot, giving rise to the instantiation ({W1}, {W2,W3,W10}, {W6,W7}) as shown.

If a tuple is deleted, then it follows a course symmetrical to its addition. Suppose W10 is deleted from the Collection Rete in Figure 4-2-b. W10 is first deleted from the collection in the second alpha memory. A new left collection-token ({W1}, {W10}) is formed, with a *delete* flag. This token is then propagated to the following memory nodes. This token causes *breaching* of the tokens in the succeeding beta memories and in the instantiation set. Breaching undoes the effect of merging. The final result of this processing is that the Collection Rete in Figure 4-2-b is reverts to its state in Figure 4-2-a.

This is the basic form of Collection Rete. A battery of optimizations can be applied to this basic structure. Some of these optimizations are simply transplanted from Rete. An example of this is the *delete optimization* introduced by Scales [26]. When a tuple such as W10 is deleted from an alpha memory, new left collection-tokens are not formed; instead, the successor beta memories (and the instantiation set) are scanned, and any copy of the tuple in any token and instantiation is eliminated. This automatically achieves breaching.

In the presence of large amounts of data, individual collections (in alpha memories) can grow quite large. Searching such collections for matching tuples would be expensive. This problem can be alleviated by building *indices* for the alpha memories. Since these alpha memories will be searched for tuples that contain specific values of particular fields (that are tested at the successor beta node), the alpha memories should be indexed on these fields. There are two major indexing schemes that can be used. The first scheme is based on partitioning the alpha memories into *equivalence classes* of tuples. It divides the collection in each alpha memory into a set of *equivalence classes*. The tuples within a single equivalence class have identical values for the fields that are tested for consistency at the successor beta node. In Figure 4-2, {W6,W7,W8,W9} can be divided into two equivalence classes: {W6,W7} and {W8,W9}. Tuples in these two classes have identical values for the field PREVIOUS-PROJECT. Thus, if the first tuple in a class passes the test at the successor beta node, then the entire class are guaranteed to do so. This allows us to quickly locate all of the matching tuples by examining only the first tuple of each equivalence class, rather examining each of the individual tuples. Furthermore, since an equivalence class automatically provides all of the consistent tuples, the computational effort required for the merging process — potentially a very expensive step in Collection Rete — is also greatly reduced.

The second major indexing scheme is based on hashing the tuples in the alpha memories, as in [16]. In the presence of large amounts of data, however, hashing can require sequential search of long lists in individual hash buckets. This can be potentially alleviated by using dynamically resized hash tables but the tradeoffs are as yet unknown.

The beta memories in Collection Rete contain a small

number of left collection-tokens rather than a large number of individual tokens as in Rete. Therefore, building indices on the beta memories, by hashing [16] or otherwise, is not expected to be as beneficial.

5. Experiments

5.1. Benchmarks

We benchmarked implementations of Collection Rete and an optimized version of Rete. The Collection Rete implementation was done as a part of a COPL implementation. For a Rete implementation, we used *CParaOPS5* [1, 19], the public-domain C-based OPS5 implementation available from Carnegie Mellon. It is one of the fastest implementations of Rete and is faster than the previous version whose performance was shown to be comparable to that of *ops5c* developed at University of Texas, Austin [23].

For the implementation of COPL, we modified a derivative of CParaOPS5 to use the Collection Rete algorithm. The COPL implementation uses basic Collection Rete augmented by the delete and equivalence-class optimizations described in the previous section. We have devised several other optimizations, but have deferred their incorporation till we can evaluate their relative efficacy. Following the argument from the previous section, we have not incorporated memory hashing into our implementation. The current implementation of COPL does not support negated conditions; work on implementing them is currently in progress. However, as discussed in Section 3.1, COPL obviates the need for many negated conditions. For some of the productions in the benchmarks below, CParaOPS5 required negated conditions but COPL did not.

The benchmark suite consists of three programs that are able to process varying amounts of data. This allows us to investigate efficiency and scalability of the two algorithms. The programs are:

- *make-teams*: This program operates on a database of employees which contains information about their present department, previous project and an evaluation of how good they are. The task is to build teams of employees given constraints that each member must be from a different department and some of the members must have worked together previously. The programs builds and counts teams that are "good", goodness being defined in terms of the individual evaluations. Data for this benchmark was generated by taking the number of employees as an argument and randomly assigning employees to departments and projects.

- *clusters*: This program operates on a collection of image objects that are characterized by their position and type (e.g. road, hangar, tarmac etc.). It computes the distance between the objects, builds clusters and computes their average size. This task is similar to the those performed by some knowledge-based image understanding systems (like SPAM [21]). Data for this benchmark was generated by taking the number of objects as an argument and placing them randomly in a 100x100 grid.

- *airline-route*: This program operates on a database of airline routes which contains information about source, destination and cost of each flight. The task is to find a minimum cost route for a particular traveler given the desired number of stages. If no route with the desired number of flights can be found, it finds the best alternate route. Data for this benchmark was generated by assuming ten airlines and twenty airports. Each airline was randomly assigned a hub, and all flights were routed for that airline as outbound-inbound pairs to random destinations. The cost for each flight is randomly assigned and is the same in both directions.

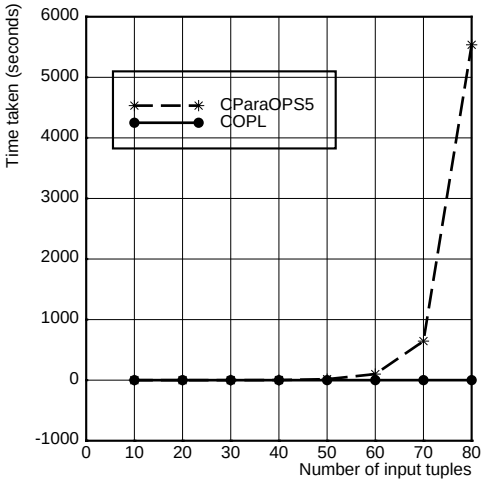
For each benchmark, we ran a sequence of experiments with progressively larger amounts of data. Each sequence of experiments was continued till the CParaOPS5 version ran up against time and/or space limits. In each case, we extended the series of experiments for the COPL version till it too ran up against similar limits. We gathered numbers on the space and the time requirements and the size of working memory for each experiment. For time measurements, we performed each experiment three times and used the average time. All experiments were performed on a pair of Decstation 5000/200 machines, running Mach 2.5, with 96M memory. All the C code was compiled by the MIPS *cc* compiler with the -O option. The CParaOPS5 compiler had all the optimization switches turned on. To determine the total execution time, we used the */bin/time* facility available in Unix.

5.2. Results

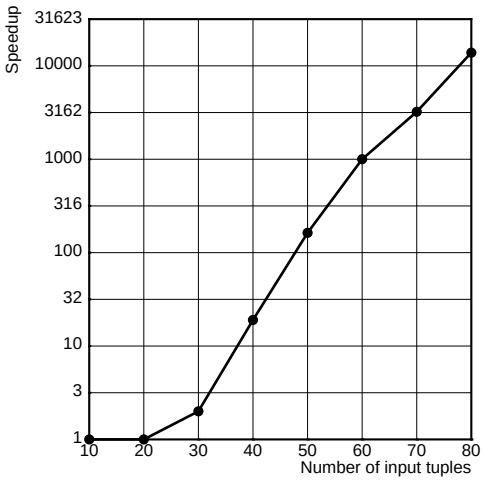
For each benchmark, we plotted both time and space requirements against the input data size. Note that some of execution-time graphs, the line for COPL programs is slightly above the x-axis, and due to the scaling, appears to lie on it. Graphs for execution time and space show results only for the experiments for which both CParaOPS5 and COPL programs could be run.

Figures 5-1, 5-2 and 5-3 contain the graphs plotting the total time and speedups against the input data size for *make-teams*, *clusters* and *airline-route* respectively. The speedup graphs are plotted on a log scale to accommodate the large range. The inflections in the speedup curves are data dependent and not an artifact of the match algorithm. They are caused by the order in which the random data is generated. The maximum speedups were 13482 for *make-teams*, 1429 for *clusters* and 57 for *airline-route*.

Figures 5-4 and 5-5 contain corresponding graphs plotting the maximum size of the match state (including alpha-memory, beta-memory and the instantiation-set) against the input data size. The highest ratios in the size of match state were 13.6 for *make-teams*, 3.4 for *clusters* and

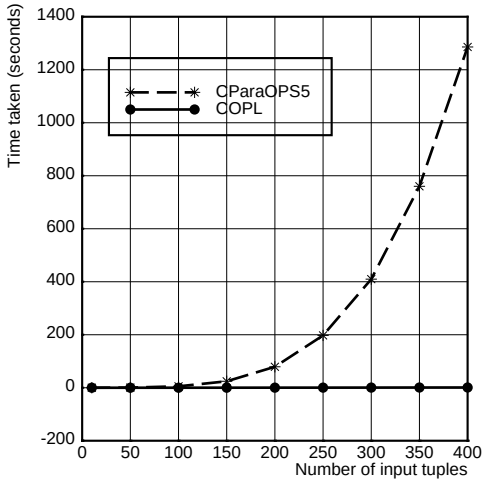


(a) Total time

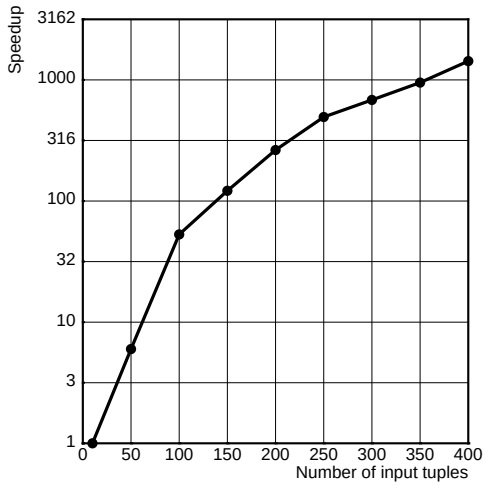


(b) Speedups for COPL

Figure 5-1: Execution time for *make-teams*



(a) Total time



(b) Speedups for COPL

Figure 5-2: Execution time for *clusters*

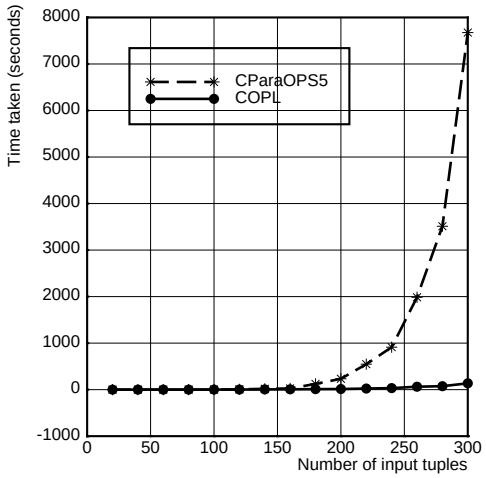
1.6 for *airline-route*.

In all the three cases, for small amounts of data, the execution time and the size of match state for both CParaOPS5 and COPL are comparable. As the size of input data increases, *COPL soon dominates CParaOPS5*.

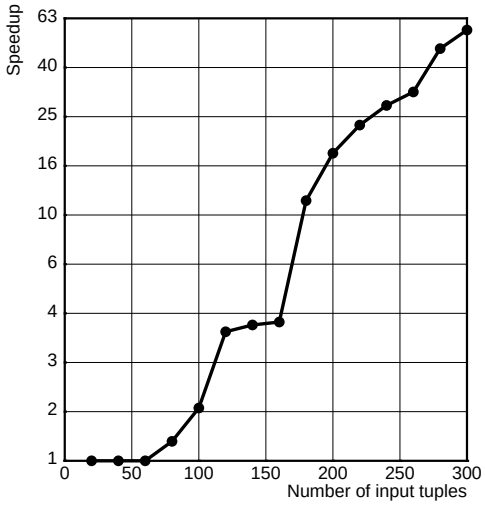
It is important to note here that while the number of input tuples may not be large, the maximum size of the working memory matched is much larger. This maximum size is different from the size of the input tuples, in that it includes all intermediate data generated during the computation. Figures 5-6 and 5-7 show graphs plotting the maximum size of working memory against the execution time. These graphs have been plotted on a log-log scale to accommodate the large range on both axes. They allow us to compare the maximum size of working memory that each pair of programs can process in a given amount of time. These graphs show that in the same amount of time, COPL is able to process up to two orders of magnitude more tuples than CParaOPS5. Furthermore, as more time becomes available, COPL is able to make better use of it than CParaOPS5. These graphs show results for all COPL runs not just those for which the corresponding CParaOPS5 runs could be completed. The largest working memory processed by COPL programs contained about 1.4 million tuples for *make-teams*, a little over 2.5 million tuples for *clusters* and a little over 200,000 tuples for *airline-route*. Corresponding numbers for CParaOPS5 are 3015, 31813 and 36455. While these numbers do seem small compared to the corresponding COPL numbers, it is important to remember that production system programs have usually dealt with no more than a few thousand tuples.

5.3. Analysis

As mentioned in Section 3, three factors govern the speedup and the match state reduction achieved by Collection Rete: (i) the size of the collections that match individual conditions, (ii) the fragmentation caused by the constraints between the conditions, (iii) the number of conditions in the productions. Programs that have large productions with large collections matching each condition and little fragmentation will achieve large time and space

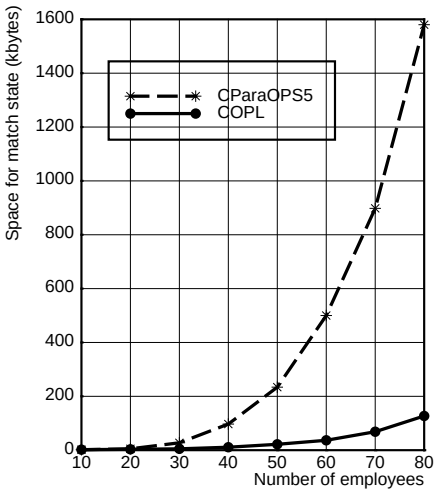


(a) Total time

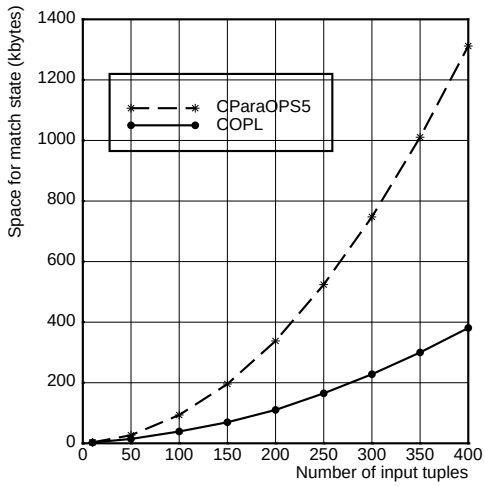


(b) Speedups for COPL

Figure 5-3: Execution time for *airline-route*



(a) *make-teams*



(b) *clusters*

Figure 5-4: Size of the total match state for *make-teams* an

improvements. Programs that don't have some or all of these characteristics will achieve lower or no improvements. The three benchmarks provide an illustration. All three of them have large collections matching individual conditions, and hence show some performance improvement. All three have some fragmentation, but *airline-route* has a significantly higher amount of fragmentation, and shows correspondingly smaller speedups. Of the other two, *make-teams* has more fragmentation than *clusters*, but it also has more conditions (three-to-four) per production than *clusters* (two conditions). This more than offsets the effect of fragmentation in *make-teams*, and leads to large speedups and match state reduction.

While all three programs achieve match state reduction, this reduction is not as high as the speedups achieved. Partly, this is because alpha memories, which typically consume very little match time, consume a relatively large proportion of match-state. Thus, for beta memories alone, the reduction in match state is much higher. However, there are also some small match-state overheads associated with

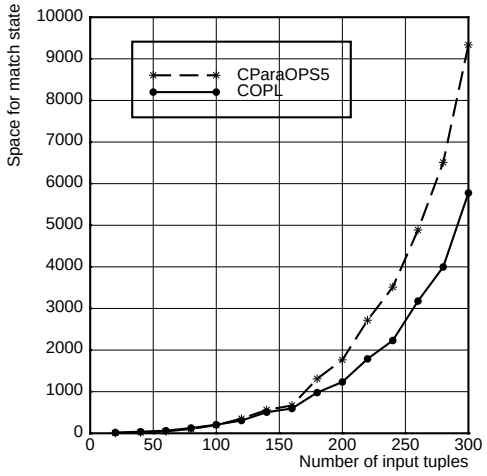
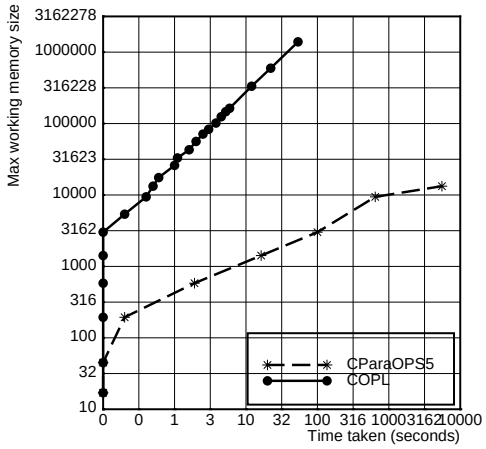
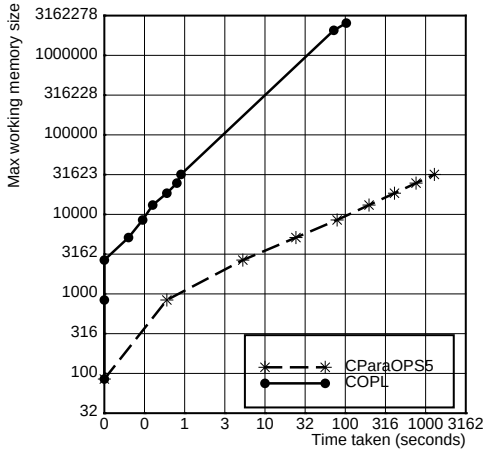


Figure 5-5: Size of the total match state for *airline-route*

COPL. First, the COPL implementation imposes additional organization in form of equivalence classes in alpha



(a) *make-teams*



(b) *clusters*

Figure 5-6: Maximum size of working memory for *make-t*

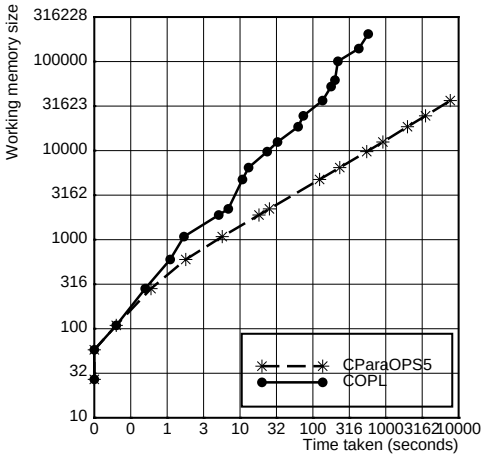


Figure 5-7: Maximum size of working memory for *airline*

memories. Second, it introduces additional organization on the left collection-tokens and the collection-oriented instantiations. In particular, while tuple-oriented tokens consist of an array of pointers to tuples, collection-oriented tokens are represented by an array of pointers to collections of tuples, each collection being implemented as a *cons-list*. The extra space consumed by these overheads depends on the amount of fragmentation in the system. For instance, if there is high fragmentation, as in *airline-route*, then the cons-list becomes a factor in the space consumed.

These results clearly illustrate the efficiency and scaling characteristics of tuple-oriented and collection-oriented match approaches. However, more importantly, we view them as establishing the feasibility of collection-oriented match for matching large amounts of data. For the *clusters* benchmarks, our Collection Rete implementation (with its limitations) is able to match over 2.5 million tuples in about 100 seconds. To the best of our knowledge, this is approximately two orders of magnitude larger than the largest working memory previously dealt with. Obviously, these results are specific to our benchmarks and more research will be required before such large working memories can be routinely dealt with.

6. Discussion

Results of the magnitude described in this paper inevitably raise questions about the fairness of the comparison and the validity of these results for real-world applications. By building a highly optimized system and comparing it with a suboptimal system using benchmarks that contain large and unrealistically complex productions, results can be made to look good. However, in our comparison, the *exact opposite* is true. The target of our comparison, CParaOPS5, is actually a highly optimized system, as testified by the following: (i) CParaOPS5 is based on a well-known compilation technique [12] and includes a variety of optimizations (e.g. hashing of memories, caching of global values, aggressive inlining); (ii) An earlier, slower version of CParaOPS5 was found to be only a factor of 1.5 to 2 slower than the optimized *ops5c* system distributed by the University of Texas, Austin [23]; (iii) For small working memory sizes, CParaOPS5's performance is comparable to that of COPL, indicating that COPL does not have some low level implementation advantage; (iv) CParaOPS5 has provided efficient support for the SPAM knowledge-based image recognition system [21], which regularly takes fifty thousand to a million production firings to run. Compared to CParaOPS5, the COPL system is in a relatively underoptimized state. This is our first implementation of COPL, with no time spent on tuning its performance. Additionally, our benchmark set contains simple productions of three-to-four conditions, with the *clusters* benchmark containing only two conditions per productions. (A larger number of conditions will give COPL much higher speedups.)

Furthermore, traditionally, performance improvements in the production systems world have been confined to single digits [23, 25, 33]. The speedups here go much beyond these single digits. It is against this background that we find the results presented in this paper promising.

However, our benchmarks not completely bias-free. Given that we targeted production systems that will operate with large amounts of data, the benchmarks are dominated

by matching of large collections. Will *real* integrated database-production systems show improvements similar to these benchmarks? If these *real* systems match large collections, then they will obtain similar speedups. The size of collections depends on the *selectivity* of the tests and the number of tuples tested. (Selectivity is defined as the percentage of tests, constant or variable, that fail.) Large collections can occur if the selectivity is low, or if the number of tuples tested is high or both. The expensive learned rules in the Soar production system [20] provide one example of low selectivity. These learned rules so expensive to match that they cause Soar to slowdown with learning rather than speeding up [31]. Image recognition systems like SPAM, and other database systems provide examples of systems where the number of tuples tested are high. We expect that as the amount of data processed by production systems grows, the size of collections will grow. Even for systems with high selectivity, large enough amounts of data will lead to large collections.

One important point here is that collection-oriented match supports a collection-oriented programming model. Production match operations, which were previously considered extremely expensive, are no longer so. This will allow a change in the programming style and is likely to expand the scope of applications to tasks which have hitherto been considered intractable.

7. Related Work

In this section, we discuss the implications of collection-oriented match for match algorithms other than Rete. We have found it relatively easy to transform tuple-oriented algorithms to their collection-oriented analogues. We also discuss other work related to matching large amounts of data.

Treat [22] is the other major algorithm found in the production systems literature. The key difference between Rete and Treat is that Treat does not maintain beta memories as a part of its match state; it only maintains the alpha memories and the instantiation set (see Figure 4-1). However, the operations it performs to determine the instantiations are similar to those of Rete — it too creates left tokens, compares these tokens with tuples in alpha memories to create new left tokens and so on. Treat can be easily transformed to *Collection Treat*, along the same lines as Collection Rete. In Collection Treat, left collection-tokens are compared with collections in alpha-memories, instead of individual tuples, and new left collection-tokens are formed. Collection Treat enjoys three advantages over Treat: (i) a reduction in the size of the instantiation set, which is the main source of space consumption in Treat [24]; (ii) a reduction in the total execution time, given that collection-oriented tokens and instantiations are formed; and (iii) direct support for collection-oriented semantics.

A'-Treat [17] is a refinement of Treat which replaces alpha memories by virtual alpha-memories, which do not maintain state. Since collection-oriented match does not change the organization of the alpha memories, A'-Treat can be transformed in the same way as Treat and the advantages listed above for Treat would carry over to A'-Treat.

Miranker et al.'s LEAPS [24] is another tuple-oriented match algorithm that reduces the amount of state saved. LEAPS has shown large performance improvements over Treat in both execution time and space. LEAPS is a demand-driven match algorithm that produces instantiations based on the demand. For instance, in OPS5, it computes only a single dominant instantiation (instead of all instantiations) as per the demand of OPS5's syntactic selection (conflict-resolution) strategy. However, the implications of LEAPS for langages that do not depend on such syntactic conflict-resolution strategies — such as Soar [20], PPL [2] and others [4]— are unclear. LEAPS also does not directly support collection-oriented languages. LEAPS can be transformed to match the dominant collection instead of the dominant tuple from each alpha memory. Collection LEAPS would generate collection-oriented instantiations, which would allow it to support collection-oriented languages without giving up the advantages of LEAPS.

Another area of related work has been that of collection-oriented (or set-oriented) production languages. Several such languages have been proposed [9, 13, 35]. Collection-oriented match can provide an efficient implementation substrate for these languages. Gordin and Pasik [13] suggest a modification to Rete to support collection-oriented languages. The resulting algorithm merges the tuple-oriented instantiations generated by Rete to generate collection-oriented instantiations. It does not take advantage of the structure of collections to tame the combinatorial explosion,

Collection-oriented match was motivated by our previous work on tokenless match [32, 31]. Hence, there are some similarities between tokenless match and collection-oriented match. However, tokenless match is targeted towards real-time systems and focuses on achieving a polynomially bounded match by limiting expressiveness. In contrast, collection-oriented match imposes no restrictions on expressiveness. As a result, even though it is able to improve production system performance, it does not improve the asymptotic complexity of the match problem. Understanding the relationship between tokenless match and collection-oriented match remains an interesting issue for future work.

8. Conclusions and Future Work

We can now answer the question that was raised at the beginning of this paper. Yes, collection-oriented match algorithms can support a large number of powerful match operations and yet scale well as the amount of data increases. Results related to the maximum working memory size in our benchmarks show that in several cases, which we expect to occur in practice, collection-oriented match is able to match OPS5-style productions against large amounts of data in reasonable amounts of time. While these results are based on an implementation of Collection Rete, a collection-oriented analogue of Rete, this paper also discussed how other tuple-oriented match algorithms can be transformed to their collection-oriented analogues. Based on a preliminary analysis, we expect the transformed

algorithms to scale better than the tuple-oriented originals. Note that although the results in the paper are presented in the context of main-memory resident systems, the collection-oriented approach as such is independent of this feature. In particular, tuple-oriented match algorithms that utilize secondary storage [5, 27, 34] will also benefit from the collection-oriented approach.

While our results have demonstrated large time and space improvements for COPL, much remains to be done. Our immediate plans for further research are to complete the design and implementation of COPL. This will allow us, and others in our environment, to build large applications in COPL. This, in turn, will provide us a better understanding of the nature of computation in collection-oriented match and its utility for real-world tasks. We also hope to use these tasks to evaluate the relative efficacy of hashing and merging optimizations. Another investigation we plan to take up in near future is the development of a uniform framework to understand and evaluate the different ways of dealing with collections in production systems including collection-oriented match and tokenless match.

Acknowledgement

We thank Bob Doorenbos, Dave McKeown, Brian Milnes, Dirk Kalp, and Paul Rosenbloom, for helpful discussions on this topic. We thank Gary Pelton and Rick Lewis for loaning us their workstations for the experiments reported in this paper.

References

1. Acharya A., and Kalp, D. Release Notes for CParaOPS5 5.3 and ParaOPS5 4.4. Distributed with the CParaOPS5 release available from Carnegie Mellon University.

2. Acharya, A. PPL: An explicitly parallel production language for large scale parallelism. Proceedings of the IEEE conference on Tools for AI, 1992, pp. 473-474.

3. Acharya, A., and Tambe, M. Collection-oriented Match: Scaling Up the Data in Production Systems. Tech. Rept. CMU-CS-92-218, School of Computer Science, Carnegie Mellon University, December, 1992.

4. Barachini, F. "The evolution of PAMELA". *Expert Systems 8*, 2 (1991), 87-98.

5. Bein J., King, R., Kamel, N. MOBY: An Architecture for Distributed Expert Database Systems. Proceedings of the Thirteenth International Conference on Very Large Databases, 1987, pp. 13-20.

6. Brant, D. A., Grose, T., Lofaso, B., and Miranker, D. P. Effects of database size on rule system performance: five case studies. Proceedings of the International conference on very large databases, 1991.

7. Brownston, L., Farrell, R., Kant, E. and Martin, N. *Programming expert systems in OPS5: An introduction to rule-based programming.* Addison-Wesley, Reading, Massachusetts, 1985.

8. Buneman, P. and Clemons, E. "Efficiently monitoring relational databases". *ACM Transactions on Database Systems* (September 1979).

9. Delcambre, L. and Etheredge, J. N. The Relational Production Language: A Production Language for Relational Databases. Proceedings of the Second International Conference on Expert Database Systems, 1988, pp. 333-352.

10. Easwaran, K. Specification, implementation and interactions of a rule subsystem in an integrated database system. IBM Research Report, RJ1820, August, 1976.

11. Forgy, C. L. "Rete: A fast algorithm for the many pattern/many object pattern match problem". *Artificial Intelligence 19*, 1 (1982), 17-37.

12. Forgy, C. L. The OPS83 Report. Tech. Rept. CMU-CS-84-133, Computer Science Department, Carnegie Mellon University, 1984.

13. Gordin, D. N. and Pasik, A. J. Set-Oriented constructs: from Rete rule bases to database systems. Proceedings of the ACM SIGMOD conference on management of data, 1991, pp. 60-67.

14. Gupta, A. *Parallelism in production systems*. Ph.D. Th., Computer Science Department, Carnegie Mellon University, 1986. Also a book, Morgan Kaufmann, (1987)..

15. Gupta, A., Forgy, C., Newell, A., and Wedig, R. Parallel algorithms and architectures for production systems. Proceedings of the Thirteenth International Symposium on Computer Architecture, June, 1986, pp. 28-35.

16. Gupta, A., Tambe, M., Kalp, D., Forgy, C. L., and Newell, A. "Parallel implementation of OPS5 on the Encore Multiprocessor: Results and analysis". *International Journal of Parallel Programming 17*, 2 (1988).

17. Hanson, E. N. Rule condition testing and action execution in Ariel. Proceedings of the ACM SIGMOD conference on management of data, 1992, pp. 49-58.

18. *INGRES Version 6.3 Reference Manual.* 1990. INGRES Products Division, Alameda, CA.

19. Kalp, D. Tambe, M., Gupta, A., Forgy, C., Newell, A., Acharya, A., Milnes, B., and Swedlow, K. Parallel OPS5 User's Manual. Tech. Rept. CMU-CS-88-187, Computer Science Department, Carnegie Mellon University, November, 1988.

20. Laird, J. E., Newell, A. and Rosenbloom, P. S. "Soar: An architecture for general intelligence". *Artificial Intelligence 33*, 1 (1987), 1-64.

21. McKeown, D.M., Harvey, W.A., and McDermott, J. "Rule based interpretation of aerial imagery". *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 5 (1985), 570-585.

22. Miranker, D. P. Treat: A better match algorithm for AI production systems. Proceedings of the Sixth National Conference on Artificial Intelligence, 1987, pp. 42-47.

23. Miranker, D., and Lofaso, B. "The organization and performance of a Treat-based production system compiler". *IEEE transactions on knowledge and data engineering 3*, 1 (1991), 3-11.

24. Miranker, D. P., Brant, D. A., Lofaso, B., Gadbois, D., On the performance of lazy matching in production systems. Proceedings of the eigth national conference on artificial intelligence, 1990, pp. 685-692.

25. Nayak, P., Gupta, A. and Rosenbloom, P. Comparison of the Rete and Treat production matchers for Soar (A summary). Proceedings of the Seventh National Conference on Artificial Intelligence, 1988, pp. 693-698.

26. Scales, D.J. Efficient matching algorithms for the SOAR/OPS5 production system. Tech. Rept. KSL-86-47, Knowledge Systems Laboratory, Stanford University, June, l986.

27. Sellis, T., and Lin, C. Performance of DBMS implementations of production systems. Proceedings of the International conference on tools for AI, 1990.

28. Sellis, T., Lin, C., and Raschid, L. "Data intensive production systems: The DIPS approach". *SIGMOD Record 18*, 3 (September 1989), 52-58.

29. Stonebraker, M. "The integration of rule systems with database systems". *IEEE Transactions on Knowledge and Data Engineering 4*, 5 (October 1992), 415-423.

30. *Sybase V4.0 Reference Manual.* 1990. Sybase Corp. Emeryville CA.

31. Tambe, M. *Eliminating combinatorics from production match*. Ph.D. Th., School of Computer Science, Carnegie Mellon University, May 1991.

32. Tambe, M. and Rosenbloom, P. A framework for investigating production system formulations with polynomially bounded match. Proceedings of the Eighth National Conference on Artificial Intelligence, 1990, pp. 693-400.

33. Tambe, M., Kalp. D., and Rosenbloom, P. An Efficient Match Algorithm for Unique-attribute Production Systems. Proceedings of the International Conference on Tools of Artificial Intelligence, 1992.

34. Tan, J. S., Maheshwari, M., and Srivastava, J. GridMatch: A basis for integrating production systems with relational databases. Proceedings of the IEEE conference on Tools for AI, 1990, pp. 400-407.

35. Widom, J. and Finkelstein, S. "A Syntax and Semantics for Set-Oriented Production Rules in Relational Database Systems". *SIGMOD Record 18*, 3 (September 1989), 36-45.

Table of Contents

List of Figures