The Organization and Performance of a TREAT-Based Production System Compiler

Daniel P. Miranker and Bernie J. Lofaso

Abstract—**Performance issues often prevent prototype production system programs from scaling to large deliverable systems. In this paper, we will describe an ensemble of techniques that compile OPS5 production system programs to executable machine code and demonstrate an increase in the execution speed of production system programs by two orders of magnitude over the commonly used LISP-based OPS5 system.**

The compiler is based on the TREAT incremental match algorithm. In this paper, we present a version of the TREAT algorithm, formulated in relational algebra and prove the algorithm correct. The compiler employs optimization techniques derived from relational database systems. Furthermore, the combination of the TREAT algorithm and the compiling techniques has substantially reduced the proportion of time spent in the match phase below the "greater than 90%" figure often cited by developers of other production system environments. To show that these results are not an anomaly of the implementation we compare our performance to the newest optimized RETE-based OPS5 compiler recently released from Carnegie Mellon University.

Index Terms—**Compiler, expert system, incremental match algorithm, OPS5, optimizing compiler, production rule language, production system.**

I. Introduction

PRODUCTION-SYSTEMS, or production-rule languages, have become an established means of encoding expertise into computer programs and rapidly prototyping ill specified systems [6]. Most of the available production-system development environments execute production systems by compiling them into an intermediate structure or match network. The network is then used as an argument to a fixed production-system interpreter. The interpretive overhead of these systems is large and often prevents small-scale prototypes from evolving into full-scale deployable systems. Therefore, it is common for application developers to rewrite compute intensive portions of a prototype production system program before deploying the program. In this paper, we will describe an ensemble of techniques that compile production system programs to executable code and demonstrate an increase in the execution speed of production system programs by two orders of magnitude over commonly used interpretive systems.

Evaluating the satisfaction of a set of production rules using simple matching techniques is computationally expensive and wasteful.[1] An important development that led to the effective execution of production system programs began with the observation that the actions of a fired rule affect only a small proportion of the working memory and that most of it remains the same from cycle to cycle [5]. Rather than reevaluating the rule system each cycle, incremental match algorithms have been developed that compute incremental changes to the set of satisfied rules from incremental changes to the working memory. The RETE match is the most commonly used incremental match algorithm [5]. Several other incremental match algorithms have appeared in the literature [20], [18], [15] as well as refinements to the RETE match [12], [4].

The incremental match problem appears in many other areas. It underlies the constraint-based inheritance methods used in object systems [25]. Updating database views in the presence of dynamic changes to the base relations is a specialization of the incremental match [2]. The semi-naive evaluation of Datalog programs is a specialization of the TREAT incremental match algorithm [24].

The compiler described herein, OPS5c, is based on a sequential version of the TREAT incremental match algorithm. The TREAT incremental match algorithm was first developed as a parallel algorithm [14]. TREAT is much less space intensive than the RETE match. In a parallel computer an algorithm's space requirements often translate into communication and contention overhead [22]. Serendipitously, it was determined that the TREAT algorithm outperforms RETE, even in a sequential environment.

The compiler is written in C using the standard Unix compiler tools and produces C as its target code. The resulting C code must then be compiled for the target machine, thus forming a portable system. We show that using the TREAT algorithm we can compile LHS patterns into very short, compact code segments such that most data references can be kept local to the fast registers of the processor.

We have used this compiler to extend our experience with optimization and indexing techniques for rule-based systems. We have determined that very detailed optimizations may only be worthwhile in the presence of detailed cost information and that ultimately the performance gained from the cost data is probably modest. The optimization techniques employed in OPS5c create a distinct optimized code sequence for each condition element of each rule. This method can lead to very large program images. However, the code generated by the compiler can be organized to have good locality properties with respect to virtual paging systems. We have found that the single most important factor in the performance of a production system may be the care with which the production system environment is integrated with the memory hierarchy of the host computer. For example, we have found that, without care, dynamic memory management can consume as much as 60–70% of the execution time of a production system program [10].

The OPS5c compiler produces the fastest sequential OPS5 executables. Of greater consequence, where it has been previously reported that the match phase of the production system cycle requires greater than 90% of the total execution time, the compilation and optimization techniques in the OPS5c compiler have substantially reduced this proportion, often to below 50%.

Manuscript received March 2, 1990; revised October 31, 1990. This work was supported in part by the Office of Naval Research under Contract N00014-86-K-0763 and in part by a grant from Texas Instruments.

D. P. Miranker is with the Department of Computer Science, University of Texas at Austin, Austin, TX 78712.

B. J. Lofaso is with the Applied Research Laboratory, University of Texas at Austin, Austin, TX 78712.

IEEE Log Number 9042063.

[1] Readers unfamiliar with production system languages should first read Section I-A.

1041–4347/91/0300–0003$01.00

```
(p ops5-example-rule
   (c1 ^a1 <x> ^a3 red)
   (c2 ^a1 <x> ^a2 <y>)
  -(c3 ^a2 <y>)
  -->)
```

Fig. 1. An OPS5 production rule.

The reduced proportion of time spent in match could be obtained by implementing the remaining portions of the system poorly. To show that these results are not an anomaly of the implementation we compare our performance to the newest optimized RETE-based OPS5 compiler, ParaOPS5, recently released from Carnegie Mellon University [8].

In the remainder of the Introduction we define production systems, give an example of a rule in OPS5 syntax, and briefly describe the benchmarks used for this paper. Section II presents a formulation of the TREAT match algorithm using relational algebra and gives a proof of correctness. The organization of the compiler is described, Section III, followed by performance results, Section IV, and the effects of join optimization, Section V.

A. Production Systems and Terminology

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* that are matched against the working memory. A pattern element is also called a *condition element* (CE). The RHS contains directives that update the working memory by adding or removing facts, and directives that affect external side effects, such as reading or writing an I/O channel.

In operation, a production system interpreter repeatedly executes the following cycle of operations:

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) Conflict Set Resolution: From the conflict set, choose a subset of instantiations according to some predefined criteria. In practice, only a single instantiation is chosen.
3) Act: Execute the actions in the RHS of the rules indicated by the selected instantiations.

Fig. 1 contains an example of a production rule written in OPS5 syntax that will be used in examples below [3]. Working memory elements (WME's) in OPS5 are represented by an initial constant denoting a record type or *class name.* The constant is followed by a list of attribute value pairs. Attribute names are prefixed with a caret. Condition elements may be negated. Negation is represented in OPS5 by a "−" character and is defined as set difference [24].

It is convenient to make an analogy between the LHS of OPS5 rules and relational database queries. We can view the WM as a set of relational tuples where the class names represent relation names and the OPS5 attribute names represent the attribute names of the tuples. So defined, matching the constants appearing in a rule's condition element is analogous to the relational select operator. The binding of variables between CE's is analogous to performing a database join. Thus, we refer to two distinct stages of the matching process, the *select phase* and the *join phase.*

TABLE I
BENCHMARK STATISTICS

System	Number of Rules	Average Size of WM	Number of Rule Firings
Waltz	33	42	70
Jig25	6	50	58
Mapper	236	1143	84
Mesgen	143	34	138
Robot	75	15	410
Tourney	17	123	528
MAB	13	11	14
Weaver	637	152	751
Rubik	70	287	326

B. The Benchmark Suite

A number of OPS5 programs were used as benchmarks for the systems described in this paper. Due to the rapid evolution of the compiler and the gathering of data over time it is not possible for us to report meaningful data for all these benchmarks for each and every table. A brief description of these programs is provided here. Table I summarizes statistics about the programs.

- Waltz: A program that executes the Waltz constraint algorithm for labeling a blocks world drawing [26].
- Robot: A program that plans the movements for a robot arm.
- Tourney: A program that schedules players for a bridge tournament.
- Jig25: A simple jigsaw puzzle solver.
- Mesgen: A program that generates the daily news report on the behavior of the Dow Jones stock market index.
- Mapper: A program that plans a route between two points in New York City.
- MAB: Monkey and Bananas.
- Weaver: A VLSI box router [21].
- Rubik: A program that solves Rubik's cube.

II. THE TREAT ALGORITHM

It is useful when evaluating a rule system to maintain an index structure, called an *alpha-memory,* for each CE in the rule program [11], [5]. An alpha-memory provides fast access to those WME's that satisfy each CE independent of the satisfaction of the other CE's.

To describe the TREAT match we will assume that, without loss of generality, a rule is composed of a number of positive CE's followed by zero or more negated CE's. We will further assume that there exists a set of alpha-memories, R_i, one for each positive CE and set of alpha-memories, N_i, one for each negated CE. Using relational algebraic notation, the contribution to the conflict-set of a single rule is defined as

$$\mathrm{CS} = (R_1 \bowtie \cdots \bowtie R_n) - (\cdots((R_1 \bowtie \cdots \bowtie R_n \ltimes N_1) \ltimes N_2) \cdots \ltimes N_m)$$

where n, m are, respectively, the number of positive and negated CE's. To simplify the notation, define $\boldsymbol{R}$, $\boldsymbol{N}$, $\boldsymbol{S}$, and $\boldsymbol{V}$ such that

$$\mathrm{CS} = \boldsymbol{R} - (\boldsymbol{R} \ltimes \boldsymbol{N})$$
$$\mathrm{CS} = R_1 \bowtie \boldsymbol{S} - ((\boldsymbol{R} \ltimes N_1) \ltimes \boldsymbol{V}).$$

A rule is called an *active rule* if $\forall_i$, $R_i \neq \phi$. A rule can be satisfied only if it is active.

In the parallel version of TREAT, each working memory update is applied to the alpha-memories concurrently. These

updates are accumulated and the join phase for each rule may then be processed in parallel. In the sequential version of TREAT, the steps are executed in depth-first order. Each update is tested for membership in each alpha-memory. If the update (newly created WME) is a member of the alpha-memory the join code is called immediately and the conflict set is updated.

The ability to share code for common rule subexpressions, first proposed in the context of the RETE match [5], appears to derive from the sequential depth-first evaluation of the matching and is not specific to either match algorithm. Although sharing was not implemented in OPS5c is should be apparent from the section on code generation, Section III-B, that the implementation of sharing would be a simple additional optimization of the current system.

In OPS5c alpha-memories are represented as linked lists. A WME is processed by hashing its class type to determine the subset of alpha-memories that are effected and then sequentially testing the WME for membership in each of those alpha-memories. If a test confirms membership in an alpha-memory the alpha-memory is updated. Updating an alpha memory results in either adding or removing a pointer to the WME. The updated alpha memory may correspond to either a positive or a negated CE. If the rule is active, one of these four possible actions is used to update the conflict set.

Case 1: Adding an element t to the alpha-memory of a positive CE.

Without loss of generality assume that the element t is added to R_1. Let $R_1' = R_1 \cup t$ and CS' represent the conflict set after t has been added. We call t the *seed working memory element,* since it will root the search for new instantiations. If t is an element of R_1 we call the ith CE the *seed CE.* The action of TREAT in this case is to add new instantiations to the conflict set that contain a reference to the seed element. We may express this incremental update operation to the conflict set as

$$\mathrm{CS}' = \mathrm{CS} \cup (t \bowtie \boldsymbol{S} - t \bowtie \boldsymbol{S} \ltimes \boldsymbol{N}).$$

The proofs of correctness for cases 1, 3, and 4 are omitted. The reader may reconstruct them by following the outline of the proof of case 2.

Case 2: Deleting an element t from the alpha-memory of a positive CE.

Without loss of generality assume that the element t has been removed from R_1. Let $R_1' = R_1 - t$ represent incrementally removing a WME, t, from R_1. The action of TREAT in this case is to examine the conflict set for any instantiations that contain a reference to t and remove them from the conflict set. We may express this incremental update operation to the conflict set as

$$\mathrm{CS}' = \mathrm{CS} - \mathrm{CS} \ltimes t.$$

Proof of correctness:

$$\begin{aligned}
\mathrm{CS}' &= R_1' \bowtie \boldsymbol{S} - R_1' \bowtie \boldsymbol{S} \ltimes \boldsymbol{N} \\
&= (R_1 - t) \bowtie \boldsymbol{S} - (R_1 - t) \bowtie \boldsymbol{S} \ltimes \boldsymbol{N} \\
&= ((R_1 \bowtie \boldsymbol{S}) - (t \bowtie \boldsymbol{S})) - ((R_1 \bowtie \boldsymbol{S} \ltimes \boldsymbol{N}) \\
&\qquad -(t \bowtie \boldsymbol{S} \ltimes \boldsymbol{N})) \\
&= ((R_1 \bowtie \boldsymbol{S}) - ((R_1 \bowtie \boldsymbol{S} \ltimes \boldsymbol{N}) - (t \bowtie \boldsymbol{S} \ltimes \boldsymbol{N})) \\
&\qquad -(t \bowtie \boldsymbol{S}) \qquad (2) \\
&= (\mathrm{CS} \cup t \bowtie \boldsymbol{S} \ltimes \boldsymbol{N}) - (t \bowtie \boldsymbol{S}) \qquad (3) \\
&= \mathrm{CS} - (t \bowtie \boldsymbol{S}) \\
&= \mathrm{CS} - (CS \ltimes t). \qquad \square
\end{aligned}$$

Let A, B, and C represent sets. The proof moves from line 1 to line 2 by using the set identity $(A - B) - C = (A - C) - B$. The next step of the proof uses the set identity $A - (B - C) = (A - B) \cup (A \cap B \cap C)$.

Case 3: Adding an element t to the alpha-memory of a negated CE.

Without loss of generality assume that the element t is added to N_1. Let $N_1' = N_1 \cup t$. The actions of TREAT in this case are to remove instantiations that may have been invalidated by t. We may express this incremental update operation to the conflict set as

$$\mathrm{CS}' = \mathrm{CS} - ((\boldsymbol{R} \bowtie t) \ltimes \boldsymbol{V}).$$

Proof omitted.

Case 4: Deleting an element t from the alpha-memory of a negated CE.

Without loss of generality assume that the element t has been removed from N_1. Let $N_1' = N_1 - t$. The actions of TREAT in this case are to add instantiations to the conflict set that may have become eligible by removing t. However, there may be other elements in N_1 that prevent any instantiations from entering the conflict set. We may express this incremental update operation to the conflict set as

$$\mathrm{CS}' = \mathrm{CS} \cup (\boldsymbol{R} \bowtie t - (\boldsymbol{R} \ltimes N_1') \ltimes \boldsymbol{V}).$$

The proof of case 4 is omitted. A proof can be developed by defining the two subcases determined by the following lemmas. The proofs of the subcases follow the same outline as the proof of case 2.

Lemma 1: Either $\boldsymbol{R} \ltimes t \cap \boldsymbol{R} \ltimes N_1' = \phi$ *or* $\boldsymbol{R} \ltimes t \subset \boldsymbol{R} \ltimes N_1'$.

Since t is a single tuple the semijoin of $\boldsymbol{R} \ltimes t$ is computed using a single value. If there exists a tuple in $\boldsymbol{R} \ltimes t$ that is also in $\boldsymbol{R} \ltimes N_1'$ then every tuple in $\boldsymbol{R} \ltimes t$ will also be in $\boldsymbol{R} \ltimes N_1'$.

Corollary: If $\boldsymbol{R} \ltimes t \cap \boldsymbol{R} \ltimes N_1' = \phi$ then $\boldsymbol{R} \ltimes N_1' = \boldsymbol{R} \ltimes N_1 - \boldsymbol{R} \ltimes t$.

Corollary: If $\boldsymbol{R} \ltimes t \subset \boldsymbol{R} \ltimes N_1'$ then $\boldsymbol{R} \ltimes N_1' = \boldsymbol{R} \ltimes N_1$. $\square$

III. The Compiler

A. Organization

The OPS5c compiler is organized conventionally [1]. The front-end, written using LEX and YACC, parses the OPS5 source into an intermediate form. The code generator is driven by the intermediate form and the symbol table and produces C code as its target. Our optimizations are performed on the intermediate form. We assumed that the C compiler that produces the final executables would perform lower level optimizations.

A nontrivial aspect of the compiler was to capture the typing flexibility and data representations that are normally part of symbolic computing environment. OPS5c represents the values of working memory elements as vectors whose elements are of type lisp-atom, where lisp-atom is defined as a union of the C types needed to capture the LISP types allowed in OPS5, including symbol. To assure consistency of token ids the compiler tokenizes all the strings appearing in the OPS5 source and produces a static component of a run-time symbol table. Although it is commonly assumed that C programs are much faster than LISP programs, the representation of LISP data structures in C is larger and slower than the representation of those structures by a good LISP system.

Part of OPS5c is a 50K byte run-time library. Besides supporting a LISP-like symbol table, the library includes an OPS5

TABLE II
PROPORTION OF EXECUTION TIME SPENT IN DYNAMIC MEMORY MANAGEMENT

System	With C malloc() and free()	Fixed Block Allocator
Jig25	42%	10%
Mapper	42%	5.4%
Mesgen	37%	1%
Robot	72%	12%
Tourney	13%	2.9%
Waltz	64%	10%

command interpreter with debugging features, functions that manipulate lists, and a fixed block memory allocator. Initially OPS5c used the standard C dynamic memory allocation functions malloc() and free(). However, timing profiles of early versions of the compiler revealed that some programs spent over half of their execution time in dynamic memory management. The introduction of fixed-block memory allocation for each of the two most commonly used data structures reduced the proportion of time spent in dynamic memory management to 12% or less. See Table II.

An earlier TREAT-based system written for the DADO parallel computer performed very well on conventional machines for small programs but thrashed for larger programs [13]. It has been observed, as a consequence of the research into the parallel execution of production system programs, that production system programs display memory locality properties similar to those of conventional computer programs [18], [14], [7]. The average size of a production system working set varies from 2 to 30 rules depending on the particular production system program and underlying match algorithm.

The OPS5c code generator produces code in three sections in order to exploit the inherent locality of production system programs. The output of each section from the code generator is loaded contiguously in memory. The three sections produced correspond to procedures for executing the select and join operations of match and the RHS actions. For a given change to working memory many more select tests are performed than join sequences and many more join sequences are executed than RHS actions. Each of the three operations requires progressively larger code segments. By not intermingling select operators with join code the selected operators are localized to one part of memory. When control passes to the join phase instructions are localized to another part of memory. Although the executables produced by the OPS5c compiler can be very large, we have yet to find a system for which paging time for instruction (text) pages is a problem.

B. Code Generation

The version of the OPS5c system with indexing has been described elsewhere [16]. In this version, alpha-memories are represented as linked lists and accessed sequentially. Additions to the working memory made from the top level are processed by hashing their class attributes to a bucket containing the select tests for each CE containing that class. When a new WME satisfies a set of select tests a pointer to it is added to the corresponding alpha-memory. A set of backpointers is maintained for each WME such that no matching is required to remove WME's from alpha-memories. For additions to the WM that result from an RHS action the class is usually known *a priori* and the hashing step is skipped.

```
join_1(){ gi0 = new_wme;
          VAR_BIND(0,gi0,1)
          WME_BIND(2,gi1)
          TEST(test_eq,0,gi1,1)
          VAR_BIND(1,gi1,2)
          NFILTER_1(3)
          TEST(test_eq,1,gi2,2)
          NFILTER_2
          gi3 = NULL;
          POS_CREATE_INST(0,3) }}}}}
```

Fig. 2. Sample join sequence for the first CE of the rule in Fig. 1.

The code generator produces a join function for each possible seed CE. This organization allows the join optimizer to develop an optimum permutation of the joins starting from each CE. The variables of the seed element are bound and the remainder of the joins are computed depth-first by nested loops. Thus, a do-while is generated for each CE other than the seed CE. Each do-while scans its corresponding alpha-memory list testing WME's for consistent variable bindings and creating new variable bindings as appropriate. If a consistent binding is found the remaining variables for the CE are bound and the next do-while is entered. Although negation in OPS5 is defined as set difference, negated CE's can be implemented as filters. These filters are implemented as loops which scan negated alpha-memories and reject the most recent CE bindings if the negated alpha-memory contains an element matching all previous variable bindings. When the innermost test is satisfied, a new instantiation has been computed and is added to the conflict set.

The join phase code generator assembles variants of four basic macros WME_BIND, TEST, VAR_BIND, and NFILTER. The macro sequence for the join function for the first CE of the example rule is illustrated in Fig. 2. A pointer is assigned to the seed element and variables in it are bound using the VAR_BIND macro. The arguments to the VAR_BIND macro are a variable identifier, a working memory element pointer, and an attribute offset into the working memory element. Where possible, the compiler exploited the C "register" declaration to force variable bindings and index pointers into the registers. We expect RISC optimizing compilers to override our register declarations and exploit the large register sets available in modern processor chips.

After the seed pointer and variable bindings are assigned, loops must be set up to scan each alpha-memory. The WME_BIND macro initializes the pointer for a loop and sets up a do-while. The next innermost do-while is entered when the pattern predicate applied by TEST passes. Note that one TEST macro is inserted for each variable that must be tested. The loops for the filters for negated CE's do not nest. Thus, the NFILTER is composed of two parts. The C definitions of each of these macros is illustrated in Fig. 3. Fig. 4 contains an Assembly language sequence that could be generated by the compiler. Notice that the code is extremely compact and that all but two data references per loop are local to a processor's fast registers.

IV. PERFORMANCE RESULTS

Table III shows the execution time and speedup of the OPS5c compiler with seed optimization compared to the LISP-based OPS5 system that is distributed from Carnegie Mellon University. Both systems were run on an HP9000/370, a Motorola 68030-based workstation. The compiled executables are up to two orders of magnitude faster than the LISP-based OPS5 system.

```
#define WME_BIND(amem_num,wme_ptr)
   ResetAMemScan(&amem[amem_num]);
   while (wme_ptr = ScanAMem(&amem[amem_num])) {

#define VAR_BIND(var_num,wme_ptr,attr_offset)
   var[var_num] = GetWmeAttr(wme_ptr,attr_offset);

#define TEST(test_name,var_num,wme_ptr,attr_offset)
   if (!test_name(GetWmeAttr(wme_ptr,attr_offset),var[var_num]))

#define FILTER_1(amem_no) \
   ResetAMemScan(&amem[amem_no]); \
   while (wme_id = ScanAMem(&amem[amem_no])) { \
     if (

#define FILTER_2 \
    ) break;     \
  }       \
    if (wme_id != NULL) \
continue;
```

Fig. 3. Primitive macros for TREAT join code.

```
/* structure declaration for alpha and beta memories */
struct alpha_mem {alpha-memory *next; wme *wmep;};

register wme *Rnewwme;     /* pointer to the new working memory element */
register struct alpha-mem *r2, *r4; /* pointers for scanning alpha mems */
register pattern-vars r<x>, r<y>;  /* registers to hold bound variables */

Assembly code for TREAT match of the first CE of the example rule.

entry-C1:       mov a1(Rnewwme), r<x>   ;Bind <x> from the seed
C2-init:        mov @alpha-ce2, r2      ;Set up alpha-memory pointer chain
C2-loop:        mov 4(r2), r3           ;Get pointer to the wme
                cmp a1(r3), r<x>        ;test <x>
                jeq C3-init             ;propagate token?
C3-fail:
C2-next:        cmp r2, nil             ;test for end
                jeq done
                mov @r2,r2              ;bump pointer
                jmp C2-loop
C3-init:        mov a2(r2), r<y>        ;bind <y>
                mov @alpha-ce3, r4      ;Set up alpha-memory pointer chain
C3-loop:        mov 4(r4), r5
                cmp a2(r5), r<y>        ;If variable binds, instantiation
                jeq C3-fail             ;fails
C3-next:        cmp r4 nil              ;Test for last alpha-memory element
                jeq C3-pass
                mov @r4, r4
                jmp C3-loop
C3-pass:        Call make-instantiation
                jmp C2-next
```

Fig. 4. Compact TREAT assembly code.

Table IV shows the proportion of time spent executing each part of the production system cycle for three OPS5 benchmark programs. The match time is broken into its two subphases. Only a modest amount of time is spent in conflict set resolution. The miscellaneous category is the time spent in miscellaneous utility functions and cannot be easily proportioned among the other phases. In none of these programs, nor in any of the others tested by Lofaso, did the total match time exceed 90%. The highest proportion of time spent in match by any system was by the Tourney program which spends only 80% of its time in match.

To demonstrate that the reduction in match time is not an anomaly of our system we compare the execution time of the output of the OPS5c compiler to that of the ParaOPS5 compiler. The ParaOPS5 compiler is a parallelizing OPS5 compiler developed at Carnegie Mellon University based on the RETE

TABLE III
EXECUTION TIME IN SECONDS COMPARING OPS5C AND THE LISP OPS5 INTERPRETER FROM CMU

System	OPS5c	LISP OPS5	Relative Speed
Tourney	6.3	1142.6	181.4
Jig25	0.4	28.4	71.0
MAB	<0.1	1	>60
Rubik	36.7	2760	75.2
Weaver	33.6	1253	37.2

TABLE IV
PERCENT TIME SPENT IN EACH PART OF THE CYCLE

	Match					
System	Select	Join	Total Match	Conflict Set	Act	Misc.
Mapper	23.8	1.4	25.2	0.5	28.3	45.9
Waltz	26.6	17.9	44.5	1.8	22.7	31.0
Tourney	3.6	76.7	80.3	2.0	7.5	10.1
Weaver	32.4	42.0	74.4	3.64	18.5	7.11

TABLE V
EXECUTION TIME IN SECONDS COMPARING ParaOPS5 AND OPS5c ON AN ENCORE MULTIMAX

System	ParaOPS5	OPS5c	Relative Speed
Tourney	46.26	2.2	2.15
Jig25	1.5	1.1	1.4
MAB	0.43	0.1	4.3
Rubik	127	119	1.07

match. In its optional sequential mode, ParaOPS5 is similar in structure to OPS5c. ParaOPS5 has a large run-time library written in C that provides many ancillary routines but produces in-line matching code. The differences between the two compilers are, ParaOPS5 is based on the RETE match, it supports only two LISP atom types, it produces native assembler code for the match portion of its target code and thus can allocate values to registers and avoid the inefficiencies introduced by the portable system.[2] Last, ParaOPS5 incorporates hash indexing into its alpha-memories.

Table V shows the execution time in seconds of the three OPS5 programs we could run on an Encore Multimax computer under both the ParaOPS5 and OPS5c. Since the systems are similar except for the match code, we conclude that the additional speed of the OPS5c system is due to match and thus the proportioning of time within the production system cycle is not an anomalous result.

Note that much of the work on parallelizing production system execution has focused on parallelizing the match phase [22], [7], [18]. The reduction in proportion of match time has serious implications for those parallel systems that parallize only the match. We conclude from Amdahl's law that introducing parallelism into a well-constructed matcher will cause the remainder of the production system cycle to become a serious sequential bottleneck [19].

[2]From private discussions with Miland Tambe the parallel overhead for ParaOPS5 is about 20%. Examination of the assembler output for OPS5c executables indicates that the code is 2–4 times larger and slower than the assembler sequences illustrated above.

V. OPTIMIZATION

A primary optimization in the execution of rule-based systems is to minimize the size of intermediate join results when matching a rule [23]. Optimization has been well studied in the context of the RETE match. These approaches include static heuristics applied at compile time, such as ART's *join from the right,* explicit user control over the network structure, as provided in ECLPS, and automated systems that develop statistics from one run and recompile in order to optimize later runs, as studied by Ishida. [12], [4], [9].

Earlier TREAT work has argued that it is not worthwhile to optimize join orders dynamically at run-time [17], [14]. In our opinion a good compiler should optimize the first execution of a program and do so without manual intervention. Manual optimizations require the users to understand the underlying implementation of the system and can make code harder to maintain. Thus, OPS5c determines the join order statically at compile time, without programmer intervention. A distinct join order is generated for each possible seed element. Furthermore, a distinct code sequence is produced for each seed. Producing distinct code for each seed may appear to be an expensive time/space tradeoff. Indeed, for large production systems, OPS5c can produce large executables. We have found that the expense is strictly at compile time and that at run time the large program images do not burden the Unix paging system.

Using OPS5c, we have experimented with three different methods for join ordering, lexical ordering, an improved version of seed-ordering, and a method based on a heuristic cost function. In lexical order the joins are computed in lexical order including the seed element. In lexical ordering, only a single join sequence and code segment are produced per rule.

In seed-ordering the seed CE is considered first and the remaining condition elements evaluated in their lexical order. This works well since a single change to an alpha-memory immediately binds its variables and restricts the search for instantiations. Retaining lexical order for the remaining CE's retains any manual optimizations done by the programmer. The improved version of seed-ordering pushes the filters for the negated CE's as early in the join sequence as possible. Forcing negated CE's early in the join order is a manual optimization often suggested in the OPS5 literature [3].

The heuristic optimization method builds a query graph representing the rule [23]. Each vertex in a rule query graph represents a single CE. Edges are drawn between vertices representing CE's with common variables. Vertices are labeled with the size of the CE's alpha-memory. Edges are labeled with the selectivity of the binding tests. The optimizer computes the cost of the computation for each possible spanning tree of the query graph and returns the path with the minimum cost. The problem of determining the minimum cost spanning tree of a query graph has been shown to be NP-complete [23]. We have found that the exhaustive search is not computationally demanding for rules containing fewer than 12 CE's.

It is not possible, *a priori,* to determine the size of the alpha-memories and the selectivity of the binding tests. Rather than derive cost information from earlier runs or resorting to user supplied pragmas, the OPS5c heuristics assume that alpha-memories are of constant size. A constant value for selectivity was derived for each type of pattern predicate. For example, the "not-equal" predicate is much more likely to be true than an "equal" predicate. If there were multiple bindings to be tested the product of their respective selectivities was used.

Table VI shows the number of comparisons required to bind

TABLE VI
NUMBER OF COMPARISON FOR THREE DIFFERENT JOIN OPTIMIZATIONS

Program	Lexical	Seed	Improvement over Lexical	Heuristic	Improvement over Seed
Jig25	59,373	38,189	0.64	38,144	0.99
Mapper	14,657	11,449	0.78	8695	0.75
Mesgen	168	167	0.99	178	1.07
Robot	8903	8672	0.97	8672	1.00
Tourney	99,649,965	1,781,954	0.02	1,463,701	0.82
Waltz	87,922	32,221	0.37	38,286	1.19

the variables of 6 sample OPS5 programs. These experiments confirm earlier results that TREAT implementations should include at least seed optimization. The improvement due to the heuristic method is inconsistent. The heuristic optimization improved the performance of three of the programs and did not improve or reduce the performance of three others. Detailed examination of Waltz and Mesgen revealed the failure of the heuristic cost function was due to the assumption that all the alpha-memories were identical in size which incorrectly led to the domination of the heuristic measures of selectivity.

Note that the heuristic optimization demonstrates only a modest span of improvement over seed ordering, 25% to −19%. These results are on a similar scale as those reported in related research [9], [17]. Recall that these results optimize only the time spent in the join phase of the match. We conclude that the gains provided by detailed join optimization are not likely to substantially improve the performance of the system as a whole and that it is likely that efforts to improve performance would be better directed at other aspects of the system.

VI. CONCLUSIONS

The OPS5c compiler was crafted to integrate well with each aspect of a computer system's memory organization. By using the TREAT algorithm we are able to generate compact loop structures to perform matching such that most of the data accesses are to the fast registers of the processor. Dynamic memory management was handled by a fixed block allocator. The executable object code was organized to minimize paging overhead. The result is that OPS5c produces the fastest executable OPS5 code of any sequential system. The combination of the optimization and compilation techniques has reduced the proportion of time spent in the match phase to below 90% of the execution time and in many examples to below 50%. This proportion raises a serious issue for those projects that have explored accelerating production system execution by applying parallelism exclusively to the match phase.

Our results on using a heuristic cost function for optimizing join orders confirms other researchers results that a rule optimizer must consider the cardinality of all of the alpha memories to make any ordering decisions. Given the reduced proportion of the time spent in match, particularly in the join phase of the match, efforts to optimize the execution speed of the system are likely to be better spent in other areas. These areas may include organizing the alpha memories for fast access rather than as linked lists and optimizing the compiling the RHS's of the rules to reduce the time in the act phase.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques and Tools.* Reading, MA: Addison-Wesley, 1986.

[2] J. A. Blakeley and N. L. Martin, "Join index, materialized view, and hybrid-hash join: A performance analysis," in *Proc. Sixth Int. Conf. Data Eng.*, IEEE Computer Society Press, Feb. 1990, pp. 256–263.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5.* Reading, MA: Addison-Wesley, 1985.

[4] B. Clayton, *ART Reference Manual,* Inference Corp., 1985.

[5] C. L. Forgy, "A fast algorithm for the many pattern/many object pattern matching problem," *Artif. Intell.,* vol. 19, pp. 17–37, 1982.

[6] M. S. Fox, "AI and expert systems, Myths, legends and facts," *IEEE Expert,* pp. 8–20, Feb. 1990.

[7] A. Gupta, *Parallelism in Production Systems.* Los Altos, CA: Pittman/Morgan-Kaufman, 1988.

[8] A. Gupta, C. L. Forgy, D. Kalp, A. Newell, and M. Tambe, "Results of parallel implementation of OPS5 on the Encore multiprocessor," in *Proc. Int. Conf. Parallel Processing,* IEEE, Computer Society Press, 1988, pp. I:271–280.

[9] T. Ishida, "Optimizing rules in production system programs," in *Proc. Nat. Conf. Artif. Intell.,* AAAI, Aug. 1988, pp. 699–704.

[10] B. J. Lofaso Jr, "On join optimization for an OPS5 compiler," Master's thesis, Dep. Comput. Sci., Univ. of Texas at Austin, 1988.

[11] J. McDermott, A. Newell, and J. Moore, "The efficiency of certain production system implementations," in *Pattern-directed Inference Systems,* Waterman and Hayes-Roth, Eds. New York: Academic, 1978.

[12] M. I. Schor, T. P. Daly, H. S. Lee, and B. R. Tibbitts, "Advances in RETE pattern matching," in *Proc. Nat. Conf. Artif. Intell.,* AAAI, Aug. 1986, pp. 226–232.

[13] D. P. Miranker, "TREAT: A better match algorithm for AI production systems," in *Proc. Nat. Conf. Artif. Intell.,* AAAI, Aug. 1987, pp. 42–47.

[14] D. P. Miranker, *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* Los Altos, CA: Pittman/Morgan-Kaufman, 1989.

[15] D. P. Miranker, D. Brant, B. J. Lofaso, and D. Gadbois, "On the performance of lazy matching in production systems," in *Proc. 1990 Nat. Conf. Artif. Intell.,* AAAI, July 1990, pp. 685–692.

[16] D. P. Miranker, B. J. Lofaso, G. Farmer, A. Chandra, and D. Brant, "On a TREAT based production system compiler," in *Proc. 10th Int. Conf. Expert Syst.,* Avignon, France, June 1990, pp. 617–630.

[17] P. Nayak, A. Gupta, and P. Rosenbloom, "Comparison of the Rete and Treat production matchers for soar (a summary)," in *Proc. Nat. Conf. Artif. Intell.,* AAAI, Aug. 1988, pp. 693–698.

[18] K. Oflazer, "Partitioning and parallel processing of production systems," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., 1986.

[19] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers.* New York: McGraw-Hill, 1987.

[20] L. Raschid, T. Sellis, and C. C. Lin, "Exploiting concurrency in a DBMS implementation for production systems," in *Proc. Int. Symp. Databases Parallel Distributed Syst.,* 1988.

[21] R. Joobbani and D. P. Siewiorek, "WEAVER: A knowledge-based routing expert," *IEEE Design Test Comput.,* pp. 12–23, Feb. 1986.

[22] S. J. Stolfo and D. P. Miranker, "The DADO production system machine," *J. Parallel Distributed Comput.,* vol. 3, pp. 269–296, June 1986.

[23] J. D. Ullman, *Principles of Database Systems.* Rockville, MD: Computer Science Press, 1982.

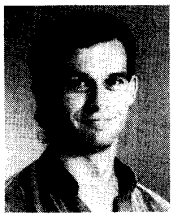
[24] ——, *Principles of Database and Knowledge-Base Systems. Vol. 1.* Rockville, MD: Computer Science Press, 1988.

[25] M. van Biema, "The constrain-based paradigm: The integration of the object-oriented and the rule-based programming paradigms," Ph.D. dissertation, Columbia Univ., 1990.

[26] P. H. Winston, *Artificial Intelligence.* Reading, MA: Addison-Wesley, 1977.

Daniel P. Miranker received the S.B. degree in mathematics from MIT, Cambridge, MA, in 1979, and the M.S. and Ph.D. degrees in computer science from Columbia University in 1983 and 1987, respectively.

He is an Assistant Professor of Computer Science at the University of Texas at Austin. His research interests include parallel symbolic computer architecture, expert database systems, and the real-time, fault-tolerance execution of decision systems.



Bernie J. Lofaso received the B.S. degrees in chemical engineering and computer science from Louisiana State University in 1980 and 1984, respectively, and a Master of Arts degree in computer science from the University of Texas at Austin in 1989.

Formerly employed at the Exxon Research and Development Labs, he now works at the Applied Research Labs, University of Texas at Austin. His interests include expert systems, computer architecture, graphics, and avoiding Unix system administration.