To be presented at the 1994 ACM SIGSOFT Conference, New Orleans

Reengineering a Complex Application Using a Scalable Data Structure Compiler[1]

Don Batory, Jeff Thomas, and Marty Sirkin
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
{batory, jthomas, marty}@cs.utexas.edu

Abstract

P2 is a scalable compiler for collection data structures. High-level abstractions insulate P2 users from data structure implementation details. By specifying a target data structure as a composition of components from a reuse library, the P2 compiler replaces abstract operations with their concrete implementations.

LEAPS is a production system compiler that produces the fastest sequential executables of OPS5 rule sets. LEAPS is a hand-written, highly-tuned, performance-driven application that relies on complex data structures. Reengineering LEAPS using P2 was an acid test to evaluate P2's scalability, productivity benefits, and generated code performance.

In this paper, we present some of our experimental results and experiences in this reengineering exercise. We show that P2 scaled to this complex application, substantially increased productivity, and provided unexpected performance gains.

1 Introduction

Programming and debugging data structures consumes a disproportional amount of resources in the construction of software. Collection data structures (e.g., arrays, lists, trees) are well-understood, but general-purpose tools to reduce the burden of implementing them have yielded mixed results. High-level languages, such as SETL, have demonstrated the usefulness of simple and powerful data structure abstractions; unfortunately, questions about the performance of these languages and their incompatibility with mainstream languages (e.g., C and C++) have constrained their popularity.

1. This research was supported in part by Applied Research Laboratories at The University of Texas and Schlumberger.

The emergence of software libraries (e.g., the Booch C++ Components, libg++, NIHCL) has lessened the burden of data structure programming for applications with simple needs. Unfortunately, applications that require specialized data structures benefit marginally from libraries; consequently, specialized data structures are still coded by hand. Adding more components to contemporary libraries is not the answer. It has been observed that no finite library of conventionally-designed components could ever encompass the enormous spectrum of data structures that arise in practice. In short, conventional libraries are inherently unscalable [Bat93, Big94].

Scalable libraries will offer primitive building blocks and will be accompied by a compiler that generates target data structures from specified compositions of blocks. Although building block libraries grow at the rate at which new blocks are added, there is a combinatorial number of ways in which blocks can be composed. Thus the domain of data structures that can be generated grows geometrically as each new block is added. It is this scalable approach to data structure construction that we have taken in developing the P2 data structure compiler [Sir93].

More specifically, the P2 compiler is based on GenVoca, a model of scalable software construction [Bat92, Bat94b]. The GenVoca approach standardizes fundamental abstractions of a domain; GenVoca components implement standardized interfaces and thus are plug-compatible, interchangeable, and interoperable. GenVoca has been applied to independently built generators for the domains of databases (Genesis [Bat88]), communication networks (Avoca/*x*-kernel [Hut91]), distributed file systems (Ficus [Hei93]), and avionics software (ADAGE [Cog93]). Our work on P2 extends the diverse list of disparate domains for which GenVoca generators have been constructed.

The key to the success of P2 will be determined primarily on its ability to scale to a variety of applications, deliver increased programmer productivity, and make minimal concessions to run-time efficiency. In [Bat93] we described the GenVoca model, how P2 implements the model, and presented some experimental results for a simple application. As our next attempt to evaluate P2, we felt that a complicated, hand-coded data structure application had to be reengineered using P2. The application that we chose was LEAPS.

LEAPS is a production system compiler that produces the fastest sequential executables of OPS5 rule sets [Mir90, Mir91]. LEAPS is a hand-written, highly-tuned, performance-driven application that heavily relies on complex and unusual collection data structures, none of which are found in conventional libraries. Because LEAPS was developed locally at the University of Texas (and that expertise on its construction was readily available), reengineering LEAPS using P2 was a logical choice for an acid test to evaluate the potential of P2's scalability, productivity gains, and the performance of its generated code.

In this paper, we present some of our experimental results and experiences in this reengineering exercise. We will show that P2 scaled to this complex application, substantially increased productivity, and provided significant performance improvements. We begin by presenting features of P2 that are relevant to LEAPS.

2 The Domain Model of the P2 Compiler

The P2 model of the collection data structures domain identifies cursors, containers, and composite cursors as fundamental data structure programming abstractions. P2 components offer different implementations of these abstractions. In the following sections, we explain the features of P2 that were used in the reengineering of LEAPS. Readers may recognize the influence of database abstractions on the P2 domain model [ACM91].

2.1 Cursors and Containers

Collection data structures—arrays, binary trees, ordered lists—implement the container abstraction. A *container* is a sequence of elements, where all the elements are instances of a single data type. Elements can be referenced and updated only by a run-time object called a *cursor* (see Figure 1).

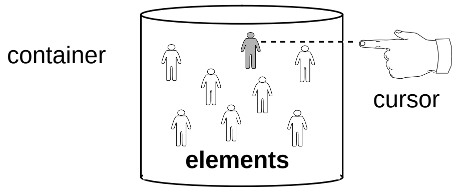

Figure 1. Basic P2 Abstractions

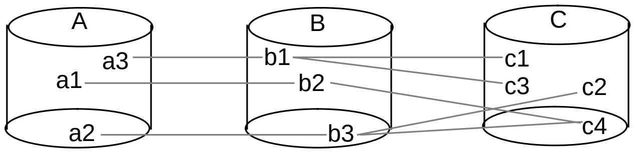

Figure 2. A Multicontainer Relationship

The P2 programming language is a superset of C. P2 introduces statements for cursor and container declarations, along with special operations on cursors and containers. An abbreviated declaration of a container of **EMPLOYEE_TYPE** instances is shown below, along with declarations of a cursor (**all_employees**) that references all elements of this container and another cursor (**selected_employees**) that references only those elements whose **deptno** field has the value **10**:

```
// Declaration of the employee container.
container <EMPLOYEE_TYPE> employee;

// Cursor that references all elements in
// the employee container.
cursor <employee> all_employees;

// Cursor that references selected
// elements of employee container.
cursor <employee> where “$.deptno == 10”
   selected_employees;
```

P2 offers an (extensible) set of container and cursor operations. For example, the **foreach** construct is used to iterate over qualified elements of a container. The **foreach** loop below prints the names of selected employees:

```
// For each element whose deptno field
// has the value 10.
foreach( selected_employees )
{
  // Print the employee name.
  printf( “%s\n”, selected_employees.name);
}
```

2.2 Composite Cursors

Complex data structures consist of multiple containers whose elements are interconnected by pointers. A *relationship* among containers C_1, C_2, ..., C_n is a set of n-tuples $<e_1, e_2, \ldots, e_n>$ where element e_i is a member of container C_i. Figure 2 depicts a relationship for containers **A**, **B**, and **C** whose 3-tuples are:

```
{ (a3,b1,c1), (a3,b1,c3), (a1,b2,c4),
  (a2,b3,c2), (a2,b3,c4) }
```

A *composite cursor* enumerates the n-tuples of a relationship. More specifically, a composite cursor *k* is an n-tuple of cursors, one cursor per container of a relationship. A particular n-tuple $<e_1, e_2, ..., e_n>$ of a relationship is encoded by having the *i*th cursor of *k* positioned on element e_i. By advancing *k*, successive n-tuples of a relationship are retrieved.

As an example, a composite cursor **c** that joins elements of the **department** and **employee** containers that share the same value of the **deptno** field is specified in P2 as:[2]

```
compcurs < d department, e employee >
   where “$d.deptno == $e.deptno” c;
```

d and **e** are aliases for container names. (As we will see in Section 3.2, aliases are useful for expressing the joins of containers with

2. Note that predicates in P2 are expressed by strings. Field **F** of the element referenced by a cursor is denoted “**$.F**”. A cursor over container with alias **X** is denoted “**$X**”.

themselves in an unambiguous way). The **foreach** loop below prints pairs of names of related **department** and **employee** elements. Readers may recognize this loop as a natural join between **department** and **employee** containers:

```
foreach( c )
{
  printf ("(%s,%s)\n",c.d.name,c.e.name);
}
```

Occasionally, it is useful to retrieve only n-tuples of a relationship that involve specific objects. Suppose we are interested only in the 3-tuples of Figure 2 that involve element **b3** of container **B** (i.e., tuples **(a2,b3,c2)** and **(a2,b3,c4)**). Such a retrieval is called *seeding* a relationship with **b3**. Seedings are expressed in P2 by augmenting the **compcurs** declaration with a **given** clause (which lists aliases of all containers that are to be seeded). Prior to a **foreach**, the **given** cursors must be positioned on the seeding elements. The above example with **b3** would be expressed by the following **compcurs** declaration and **foreach** code fragment:

```
// Declare seeded_composite_cursor seeded
// by b.
compcurs < a A, b B, c C > given < b >
   seeded_composite_cursor;

// Position seeded_composite_cursor.b
position( seeded_composite_cursor.b, &b3 );

// Iterate over seeded tuples.
foreach( seeded_composite_cursor )
{ ... }
```

Updating elements within a **foreach** loop is possible. Such updates may affect the n-tuples that are subsequently retrieved by a composite cursor. For example, after we delete an element of an n-tuple, we do not want to retrieve another n-tuple that contains this deleted element. Recall composite cursor **c** which returns pairs of related **department** and **employee** elements. The **foreach** loop of Figure 3a prints each retrieved ordered pair and then deletes the **department** element of that pair.

(a)
```
foreach(c)
{
  printf("(%s, %s)\n",
         c.d.name, c.e.name);
  delete(c.d);
}
```

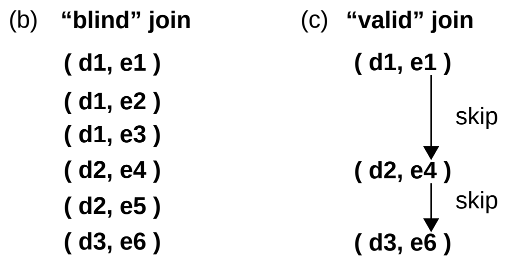

Figure 3. Updating elements within a foreach loop.

The "blind" join of Figure 3b and the "valid" join of Figure 3c illustrate two possible semantics for the foreach loop of Figure 3a. A blind retrieval ignores updates (e.g., deletions) to a container which are made while the container is being traversed. A valid retrieval, on the other hand, reflects these updates by performing a test (e.g., checking to see if the **department** element of a pair is deleted) prior to each cursor advancement, in order to make sure that the next n-tuple is meaningful (e.g., it is meaningless to perform **printf** or **delete** on a deleted element).

In this example, after deleting the element **d1**, the blind join still retrieves the pairs **(d1, e2)** and **(d1, e3)**, even though they involve the deleted element **d1**. Similarly, after deleting the element **d2**, the blind join retrieves the pair **(d2, e5)**. The valid join, on the other hand, skips over pairs that contain a deleted element.

P2 supports validation of n-tuples using the **valid** clause of a composite cursor declaration. The following declaration and code fragment eliminates the problems of a blind retrieval of (**department**, **employee**) pairs by returning only pairs of undeleted elements:

```
compcurs < d department, e employee >
   where "$d.deptno == $e.deptno"
   valid "!deleted($d)"
   valid_composite_cursor;

// Skips pairs with deleted elements.
foreach( valid_composite_cursor )
{
   delete( valid_composite_cursor.d );
}
```

Note that tuple validation is more general than merely testing for tuple deletion. P2 permits any predicate to be used for element validation. For example, the **deptno** field of a **department** element might be updated within a **foreach** loop. In this case, the **department** element has not been deleted, but its modification may affect the sequence of (valid) tuples that can be produced. Tuple validation is a general-purpose feature that is useful in graph traversal and garbage collection algorithms, where cursors may be positioned over elements that are suddenly deleted and cursor validation is needed to ensure correct executions.

2.3 Customizing Data Structure Specifications

P2 programs are written in terms of cursor, composite cursor, and container abstractions without regard to how these abstractions are implemented. The P2 compiler automatically translates P2 declarations and operations into C code. In order for P2 to accomplish this, P2 users must specify an implementation of these abstractions by composing building-blocks from the P2 library. Such a composition is declared in a **typex** (*type expression*) declaration:

```
typex { simple_typex =
          top2ds[qualify[dlist[
                   malloc[transient]]]];
      }
```

`simple_typex` is a composition of five P2 components, where each component encapsulates a consistent data and operation refinement of the cursor-container abstraction and is responsible for generating the code for this refinement [Sir93]. The **`top2ds`** layer, for example, translates **`foreach`** statements into primitive cursor operations (**`reset`**, **`advance`**, **`end_of_container`**); **`qualify`** translates qualified **`advance`** operations into **`if`** tests and unqualified **`advance`** operations; **`dlist`** connects all elements of a container onto a doubly-linked list; **`malloc`** allocates space for elements from a heap; and **`transient`** allocates heap space from transient memory. P2 code generation relies on sophisticated macro expansion and partial evaluation techniques [Bat93].

A type expression is a high level declarative specification of the implementation of cursors and containers. Minor changes to a type expression can generate substantially different code. Usually, a P2 program includes only a few type expressions and each type expression itself is at most a few lines long. Thus, tuning and maintaining a P2 program is often a matter of changing a few lines of type expressions.

3 Reengineering the LEAPS Algorithms

OPS5 is a forward-chaining expert system [McD78, For81]. LEAPS (*L*azy *E*valuation *A*lgorithm for *P*roduction *S*ystems) is a compiler that translates OPS5 rule sets into C programs. LEAPS produces the fastest sequential executables of OPS5 rule sets, sometimes outperforming OPS5 interpreters by several *orders* of magnitude [Mir90, Mir91]. Besides the expected performance gains made by compilation, LEAPS relies on special search algorithms and sophisticated data structures to make rule processing efficient.

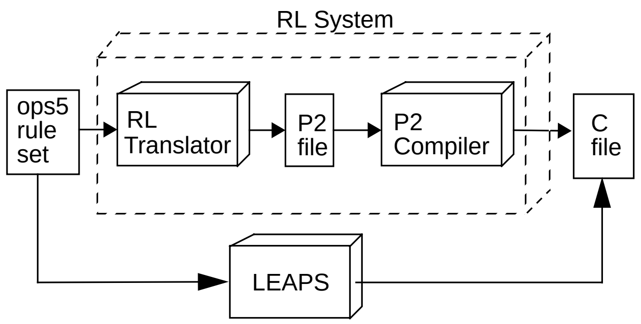

Figure 4. Relationship between LEAPS and RL

Figure 4 shows a relationship between LEAPS and P2. To reengineer LEAPS required us to translate OPS5 rule sets into a P2 program; this translator was called RL (*R*eengineered *L*eaps). The RL-generated P2 program would then be translated into a C program by the P2 compiler, thus effectively accomplishing in two translation steps what the LEAPS compiler does in one. All of the LEAPS algorithms would be embedded in the generated P2 program.

We faced two difficult challenges in this reengineering effort. First, it was well-known that the LEAPS algorithms were difficult to understand. This was due, in large part, to a lack of high level abstractions appropriate to express the details of the algorithms. Since there was no appropriate language in which to express the algorithms, it was difficult for the LEAPS developers to explain (or code) them. Thus, elegance of specification of LEAPS algorithms (in P2) was an important goal for us. Second, even if LEAPS algorithms had an elegant expression, the code that P2 produced would have to be very efficient to compete with LEAPS.

This section shows that LEAPS does have an elegant specification in P2. In Section 4, we consider the performance and productivity gains using RL/P2 generated code.

3.1 OPS5 Background

OPS5 rule sets begin with container declarations called **`literalize`** statements:

```
(literalize rel1 val)
(literalize rel2 val)
```

The above statements declare containers **`rel1`** and **`rel2`** whose elements have a single field called **`val`**. LEAPS dynamically infers the data types of element fields. In RL, we chose to augment **`literalize`** statements by statically supplying the data type for each field.

The remainder of an OPS5 rule set is a sequence of rules of the form:

```
(p print-sequence
   (rel1 ^val <r>)
   (rel2 ^val <s> ^val > <r>)
   (rel1 ^val <t> ^val > <r> ^val > <s>)
 -->
   (write strictly increasing sequence
          <r> <s> <t>))
```

The name of the above production is **`print-sequence`**. The clauses prior to the arrow **`-->`** define the rule's *qualification*; the clauses after the arrow define the rule's *action*. A rule qualification is a conjunction of clauses called (*positive*) *condition elements* (CEs). A CE serves two purposes: (1) to bind variables to particular fields of an element and (2) to qualify elements on the basis of their field values. The first CE in **`print-sequence`** binds the variable **`r`** to the **`val`** field of a **`rel1`** element. The second CE binds variable **`s`** to the **`val`** field of a **`rel2`** element *and* requires this field to be greater than **`r`**. The third CE binds variable **`t`** to the **`val`** field of a **`rel1`** element *and* requires this field to be greater than both **`s`** and **`r`**. In effect, the qualification of this rule is to identify 3-tuples **(`rel1`** element**, `rel2`** element**, `rel1`** element**)** whose **`val`** values are in strictly increasing order. The action of this rule is to print the **`val`** values of a 3-tuple.

We chose the **`print-sequence`** rule as an example because it is simple. OPS5 rules can be more complicated when negated CEs are included. A *negated condition element* is a predicate that disqualifies n-tuples that satisfy the positive CEs of a rule. OPS5 rule actions can also be more complex. Actions may specify element creation, deletion, modification, and calls to external routines. A more complete discussion of the capabilities of OPS5 can be found in [McD78, For81]. We will use the **`print-sequence`** rule to illustrate the translations performed by RL. A full explanation of RL translation is given in [Bat94a].

Forward-chaining inference engines, including LEAPS, use a match-select-action cycle. Rules that can be matched (i.e., tuples found to satisfy their qualification) are determined, one n-tuple is selected, and its corresponding action is fired. This cycle continues until a fix-point has been reached (i.e., no more rules can be fired). Conventional eager match algorithms are inherently slow, as they materialize *all* tuples that satisfy the predicate of a rule. These materialized tuples are stored in data structures and have a negative impact on performance because they must be updated as a result of executing rule actions. A fundamental contribution of LEAPS is the lazy evaluation of tuples; i.e., tuples are materialized only when needed. This approach drastically reduces both the space *and* time complexity of forward-chaining inference engines, and provides LEAPS with its phenomenal increase in rule execution efficiency.

3.2 Rule Translation

The difficult part of converting OPS5 rules into P2 code is the translation of rule qualification; translating rule actions is straightforward. There are four basic steps in rule qualification translation. The first is to convert positive CEs to P2 predicates. Figure 5a shows the `print-sequence` rule and Figure 5b shows its corresponding composite cursor declaration. Note that each CE of the rule corresponds to a container that is to be joined.

(a)
```
(p print-sequence
     (rel1 ^val <r>)
     (rel2 ^val <s> ^val > <r>)
     (rel1 ^val <t> ^val > <r> ^val > <s>)
   -->
```

(b)
```
# define ps_query “$b.val > $a.val
                  && $c.val > $b.val
                  && $c.val > $a.val”

typedef compcurs < a rel1, b rel2, c rel1 >
        where ps_query curs_ps;
```

Figure 5a-b. Rule Translation Step 1: Conversion of Selection Predicates

The lazy evaluation of composite cursors in LEAPS centers around the concept of dominant object. The *dominant object* is the most recently updated element that has not yet been processed. “Processed” means that the element has not seeded and fired all of the rules that it could. In order to support the seeding of dominant objects, multiple copies of an OPS5 rule are spawned, one copy for each different condition element that is being seeded. The second step in rule translation is to replicate a composite cursor definition, one copy for each possible seed position. Figure 6a shows the format of a cursor declaration produced in Step 1; Figure 6b shows the replication of this rule with different seeds. Note that the effect of this rewrite is to translate an n-way join into n (n-1)-way joins, each simpler than the original n-way join.

(a)
```
typedef compcurs < a …, b …, c … >
        where ps_query  curs_ps;
```

(b)
```
typedef compcurs < a …, b …, c … >
        given < a >
        where ps_query curs_ps_a;

typedef compcurs < a …, b …, c … >
        given < b >
        where ps_query curs_ps_b;

typedef compcurs < a …, b …, c … >
        given < c >
        where ps_query curs_ps_c;
```

Figure 6a-b. Rule Translation Step 2: Replication of Composite Cursors by Seeding

OPS5 semantics impose a fairness criterion such that no n-tuple can fire a rule more than once. Fairness is achieved in LEAPS through the use of timestamps and temporal qualifications. Every element has an RL-augmented timestamp field `_ts` that indicates when the element was last updated. OPS5 semantics are realized by requiring that all elements of an n-tuple have timestamps no older than the timestamp of the dominant object that seeded the n-tuple. Figure 7a shows a cursor definition produced in Step 2; Figure 7b shows the addition of temporal predicates to the `where` clause of the cursor.

(a)
```
typedef compcurs < a …, b …, c … >
        given < a >
        where ps_query  curs_ps_a;
```

(b)
```
#define ps_temporal_qual
        “$b._ts <= dominant_timestamp
         && $c._ts <= dominant_timestamp”

typedef compcurs < a …, b …, c … >
        given < a >
        where ps_query && ps_temporal_qual
        curs_ps_a;
```

Figure 7a-b. Rule Translation Step 3: Addition of Temporal Predicates

Once a rule is fired, the composite cursor is placed on a stack, thereby suspending its execution. At some later time when the element that seeded the composite cursor again becomes dominant, the composite cursor is popped and advanced to the next n-tuple. During the time the cursor was on the stack, any or all of the elements of the last n-tuple it produced could have been modified or deleted. Consequently, advancements of composite cursors must be validated. This is accomplished by adding a `valid` predicate to each cursor declaration. Figure 8a shows a cursor definition produced in Step 3; Figure 8b shows the addition of the `valid` predicates.

(a)
```
typedef compcurs < a ..., b ..., c ... >
        given < a >
        where ps_query && ps_temporal_qual
        curs_ps_a;
```

(b)
```
#define ps_valid_pred “!deleted($a)
                       && !deleted($b)
                       && !deleted($c)”

typedef compcurs < a ..., b ..., c ... >
        given < a >
        where ps_query && ps_temporal_qual
        valid ps_valid_pred
        curs_ps_a;
```

Figure 8a-b. Rule Translation Step 4: Addition of Validation Predicates

3.3 Other Issues

There are additional issues regarding the translation of OPS5 rule sets into P2 programs that are worth mentioning. First, there are two more translation steps when rule qualifications involve negated CEs. These steps are no more complicated than the four defined in the previous section [Bat94a].

Second, when an element is inserted in LEAPS, it is pushed onto a wait-list stack for subsequent seeding. Composite cursors, whose execution was suspended, are placed on a join-stack. The stack whose top element has the most recent timestamp is chosen to be the dominant object on the next execution cycle. In RL, the wait-list stack and join stack are unified. This gives a very compact and elegant representation of the primary cycle loop (see Figure 9a).

Third, the P2 procedures for rule firings are also compact (see Figure 9b). If a cursor has not yet been created, one is allocated on the heap, initialized, and positioned on the seeding element. Control then falls to the `foreach` statement. If a cursor (whose execution has been suspended) has been created, control continues at the end of the `foreach` statement (where validation tests are performed by P2). Once an n-tuple is generated, the rule is fired and the procedure is exited. After all n-tuples have been generated, control passes to the next rule for possible firing.

Also included in RL are additional LEAPS optimizations: the use of predicate indices (i.e., linking together all elements of a container that satisfy a given predicate), active rule optimizations (i.e., skipping rules that are known a priori not to be able to produce n-tuples), using symbol tables to minimize string comparison times, inlining of container insertion and deletion operations, plus others. Overall, RL provided a faithful re-implementation of the LEAPS algorithms.

4 Results

Three results surprised and impressed our LEAPS colleagues. First, P2 permitted a clean, high-level specification of the LEAPS algorithms (i.e., the RL translator and the P2 files it generated). LEAPS algorithms were previously difficult to explain and comprehend because of a lack of high level abstractions appropriate to express their details. The P2 container and cursor abstractions served this purpose particularly well. The other two results were improved performance and productivity. We describe the benchmarks we used to test our implementation and elaborate on these results in the following sections.

(a)
```
execute_production_system()
{
  while(1) {
    // Get the top of the stack.
    reset_start(top);
    if (end_of_container(top)) {
      // The stack is empty.
      // We’re at a fix-point.
      break;
    }
    else {
      // The stack is not empty.
      fresh = !top.curs;
      dom_timestamp = top.time_stamp;
      (*top.current_rule)();
    }
  }
}
```

(b)
```
void seed_rule_ps_a ( void )
{
  curs_ps_a *c;
  if (fresh) {
    c = (curs_ps_a*)
         malloc(sizeof(curs_ps_a));
    top.curs = (void*) c;
    init(*c);
    position(c->a, top.cursor_position);
  }
  else {
    c = (curs_ps_a *) top.curs;
    goto cnt;
  }
  foreach(*c) {
    fire_rule_ps( c );
    return;
    cnt:; // perform valid tests here
  }
  free(c);
  fresh = TRUE;
  top.current_rule = nextrule;
  nextrule();
}
```

Figure 9a-b. Execution Cycle & Rule Seeding Procedures

4.1 The Benchmarks

We tested LEAPS and RL on five rule set benchmarks provided by our LEAPS colleagues, which they considered a sufficient test for RL. These benchmarks have been used for years to measure the performance of rule execution engines, and are typical of LEAPS applications [Bra91]. Further, these benchmarks were particularly useful because (unlike some OPS5 rule sets) they required an input file. We were thus able to run each benchmark with input files of various sizes, to better characterize the performance difference between LEAPS and RL/P2.

The **`basic_cycle`** benchmark consists of a single rule. It counts from 0 to the number specified in its input file. This benchmark is useful for evaluating the execution-cycle overhead of forward-chaining inference engines.

The **`tripl`** benchmark consists of 2 rules. Given k, the number specified in the input file, it finds all triples (n1, n2, n3) where n1 < n2 < n3 ≤ k.

The **`manners`** benchmark consists of 8 rules. It finds table seating arrangements for guests subject to constraints (e.g., the sex and hobbies of the guests). The input file is a set of guests.

The **`waltz`** and **`waltzdb`** benchmarks consist of 33 and 38 rules respectively. Both label two dimensional graphs. **`waltz`** labels junctions of 2 or 3 edges; **`waltzdb`** labels junctions of 4 or 5 edges. Although both rule sets are functionally similar, they are implemented differently and have different execution behaviors. The input files for both are the graphs to be labeled.

Although the number of rules per rule set seems small, the programs generated by RL, P2, and LEAPS are substantial (see Table 1). The LEAPS and RL/P2 programs for **`waltzdb`**, for example, exceed 15,000 lines of code.[3] To our knowledge, we believe the C file for **`waltzdb`** generated by RL/P2 to be among the largest ever generated by a data structure compiler.

4.2 Performance Results

The performance of RL/P2 generated programs surprised us and our LEAPS colleagues. From the beginning, the primary design goal of LEAPS was performance. Because LEAPS uses data structures that were carefully optimized and hand-coded by experts, whereas RL uses generic data structures whose code is generated automatically by a general-purpose tool (i.e., P2), we expected the performance of RL/P2 programs to only approach that of LEAPS-produced programs. Nevertheless, our results show that in all cases RL/P2 programs are substantially faster than their LEAPS counterparts. Speedups ranged from 1.5 (for **`waltzdb`**) to 2.5 (for **`tripl`**).

3. LEAPS generated programs are linked with a 10,000 line run-time library. Thus, 10,000 lines should be added to the size of LEAPS-generated files.

Figures 10a through 14a show the running time of the LEAPS and RL/P2 versions of the benchmarks, and Figures 10b through 14b show the speedup of the RL generated version relative to the LEAPS generated version.[4]

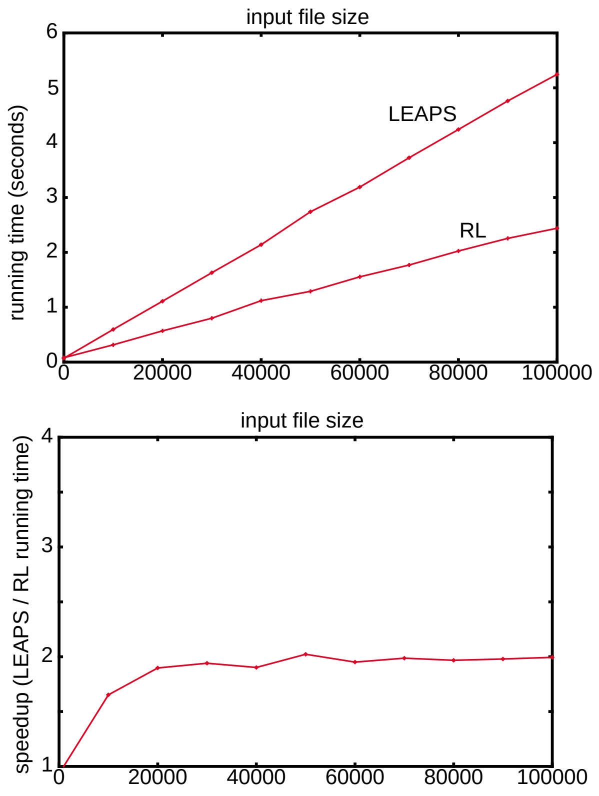

Figure 10a-b. Performance Graphs for the `basic_cycle` Benchmark

We discovered that the performance improvements of RL/P2 over LEAPS are largely due to two factors. First, P2 is able to perform complex code optimizations automatically, while such optimizations are difficult or impractical to perform by hand. Thus, the first factor is the high quality of code produced by the P2 compiler.

Second, RL implements the LEAPS functionality more cleanly than does LEAPS:

4. These results were obtained on a DECstation 5000/240 using the **`gcc 2.5.8`** compiler with the **`-O2`** option.

Rule Set	**Rule Set Size**	**RL-generated P2 Program Size**	**P2-generated C Program Size**	**LEAPS-generated C Program Size**
`manners`	8 rules	770 lines	3,300 lines	2,300 + 10,000 lines run-time
`waltz`	33 rules	2,400 lines	13,600 lines	10,000 + 10,000 lines run-time
`waltzdb`	38 rules	3,100 lines	15,800 lines	15,000 + 10,000 lines run-time

Table 1: Size of Generated Programs

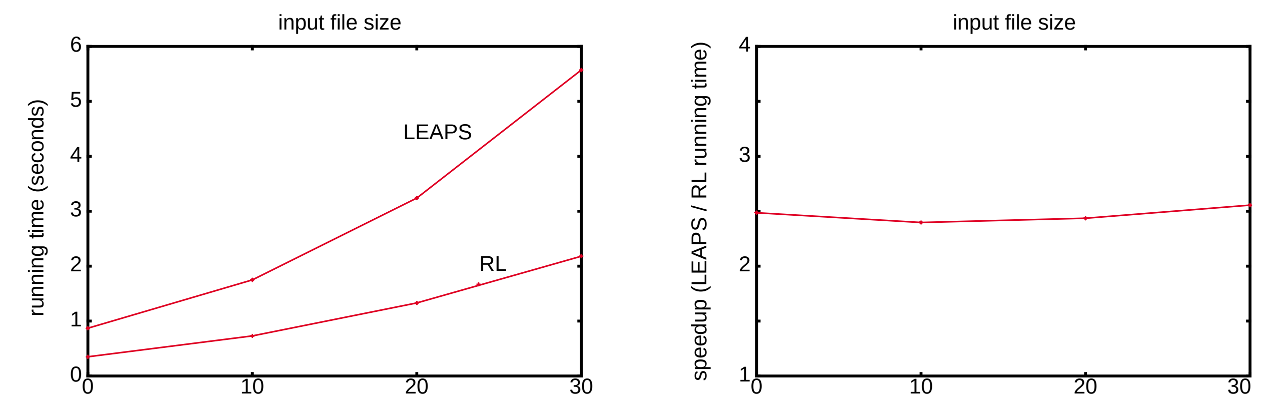

Figure 11a-b. Performance Graphs for the `tripl` Benchmark

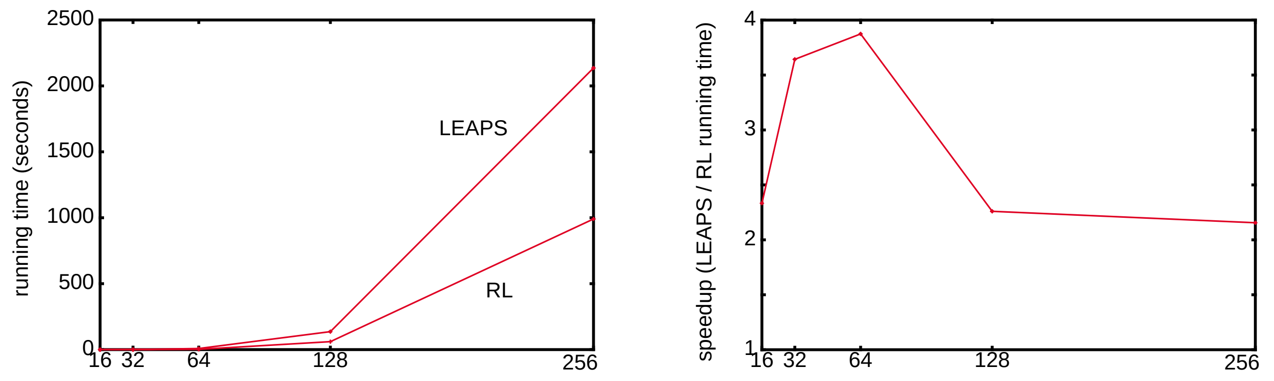

Figure 12a-b. Performance Graphs for the `manners` Benchmark

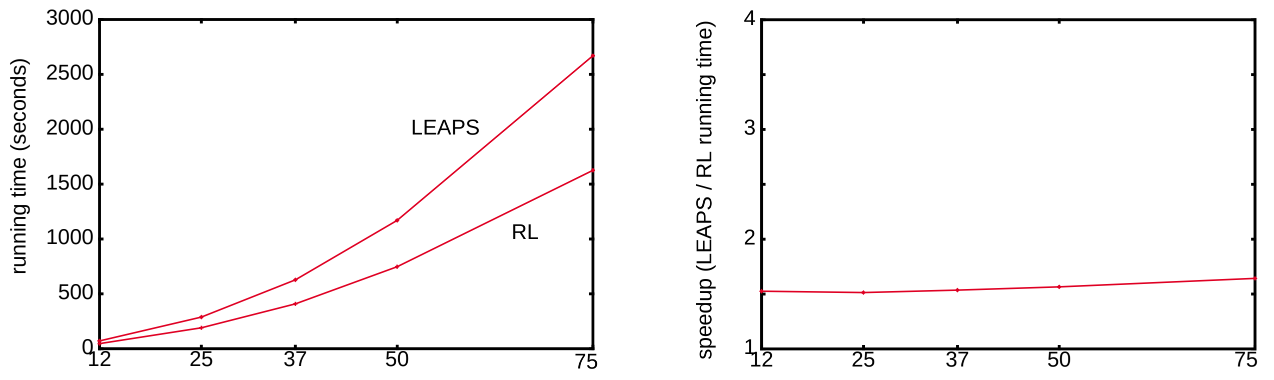

Figure 13a-b. Performance Graphs for the `waltz` Benchmark

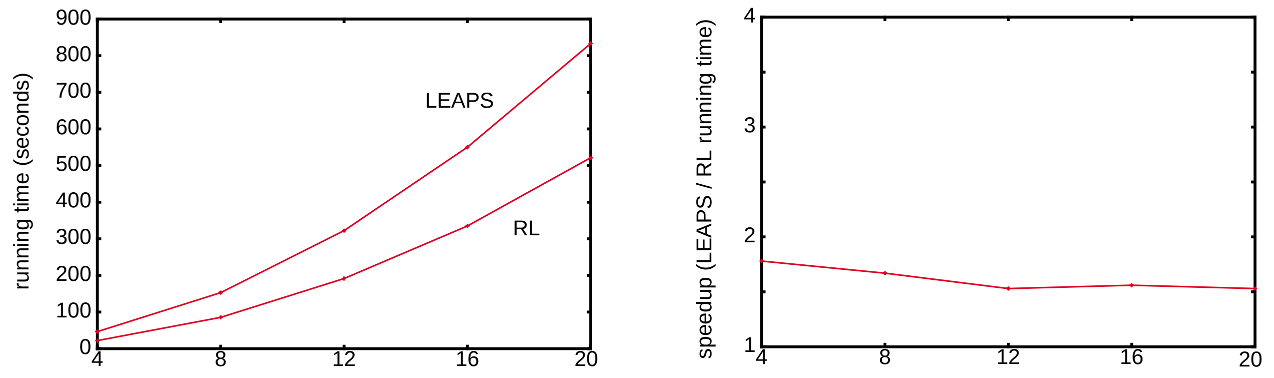

Figure 14a-b. Performance Graphs for the `waltzdb` Benchmark

- RL uses a unified wait-list/join stack. This reduces execution cycle overhead (see the results for the `basic_cycle` benchmark, in particular).
- RL does not replicate predicate indices (also called alpha memories). This significantly reduces the overhead of element insertions and deletions.
- RL uses a static type system. This reduces the run-time overhead for element qualification.
- RL does not perform garbage collection. LEAPS performs rudimentary garbage collection, which does not seem to be effective and has run-time overhead.

It is worth noting that the above four bullet-items were conjectured to be performance problems by our LEAPS colleagues, but the complexity of the LEAPS implementation discouraged, if not precluded, experiments to test these conjectures. Our research confirmed their intuition by providing performance numbers. It is only *after* we presented our results that the conjectures of our LEAPS colleagues came to light. We did *not* take these conjectures into account in designing RL. Rather, the high-level abstractions provided by P2 encouraged these design decisions and made it easy for us to experiment with and finally implement these optimizations in RL. Thus, the second factor in the performance improvement of RL/P2 over LEAPS is the high level abstractions provided by P2.

4.3 Productivity Results

Interviews of the LEAPS development team indicated that they believed that LEAPS experts could build a version of LEAPS (functionally equivalent to RL) in no less than twelve weeks. They also believed domain novices such as ourselves (experienced C programmers, with no expertise in expert system or forward-chaining inference engines) would take at least twice that long. Given P2, however, coding RL was straightforward. We wrote an initial prototype of RL within one week, and over the next seven weeks enhanced it with additional LEAPS optimizations. Thus, P2 reduced the programming time by a factor of three.[5] In short, P2 enabled novices (ourselves) to program like domain experts.

The productivity benefits of P2 can also be seen in decreased source code size. The implementation of LEAPS consists of approximately 20,000 lines of code. RL consists of approximately 4,000 lines of code. Although it was unnecessary for us to implement every feature provided by LEAPS (since not all LEAPS features were used in by our benchmark set), we estimate that to do so would require us to add not more than 1,000 lines to RL. Thus, not only did P2 reduce programming time by a factor of three, it reduced the total number of lines of code necessary by a factor of four. Thus, by either measure, P2 affords a large productivity advantage.

5. Actually, part of the eight weeks was spent waiting for P2 to be debugged, so our estimates of how long RL took to develop is conservative.

Besides improved productivity, P2 provided other important software engineering benefits:

- **Initial implementation**. Our initial prototype of RL used general-purpose components from the P2 library. Thus, P2 permitted us to get an RL prototype up and running very quickly (i.e., one week).
- **Library Scalability**. Two RL-specific components were added to the P2 library in order to improve the performance of RL/P2 generated code. These components implemented timestamp ordered lists and predicate indices, and were used in the version of RL/P2 that we benchmarked. Overall, the number of lines of code added to the P2 library was minimal (i.e., 335 lines). We remind readers that these components are *GenVoca building blocks*; i.e., they must be composed with other building blocks for memory allocation, element qualification, etc. (e.g., the `top2ds`, `qualify`, etc. components of Section 2.3) in order to produce the desired data structure. The size of comparable components that would have to be added to conventional (template) libraries would be considerably larger because they implement specific compositions of many GenVoca building blocks.
- **Tuning**. The P2 separation of data structure abstractions from data structure implementations enabled us to tune RL-generated programs easily by altering `typex` (i.e., component composition) statements. The complexity of attempting similar optimizations in LEAPS is daunting and would require significant rewriting and debugging. (We note that changing data structures in LEAPS was never attempted because of the effort required; entire rewrites were performed instead).
- **Maintenance.** Altering `typex` statements allowed us to implement enhanced versions of LEAPS (e.g., with persistent containers) in days [Sir94]. Comparable changes to LEAPS [Bra93, Bro94] required months of effort.
- **Design Scalability.** Interviews of the LEAPS development team indicated that debugging LEAPS was very difficult and that all of the major problems were caused by errors in data structures. Developing LEAPS without P2 is a monumental undertaking. Its complexity discourages attempts at trying different designs and fine-level tuning. We have learned that abstracting away the voluminous and complex data structure implementation details of an application promotes clean and efficient designs. In short, this is an issue of the scalability of software design; organizing fewer details often leads to better and more maintainable products.

5 Conclusions

In [Bat93], we presented results that showed that for a simple application, the performance of P2 generated code was at least as good as components in popular template libraries. Moreover, we observed that P2 offered much greater possibilities for software productivity, because of the uniform and high-level abstractions in which P2 users would program. These simple test cases, however, failed to convincingly demonstrate that P2 could scale to address much larger problems.

In this paper, we report our first major experiment to evaluate P2 on a large and complex application. We reengineered the LEAPS production system compiler, a performance-driven forward-chaining inference engine that was hand-written and highly-tuned. LEAPS relies on complex data structures and search algorithms that, because of their unusual and application-specific nature, were not offered by available template libraries. Moreover, the retrieval requirements of LEAPS—specifically those embodied in P2's composite cursors—far surpass the capabilities of current and prototype template libraries. We believed that LEAPS would provide an acid test to evaluate the scalability, productivity, and performance advantages of P2.

Our results using P2 surpassed our expectations:

- Only two simple components had to be written to augment the set of components in the P2 library to implement LEAPS.
- We were able to achieve a clean, compact, and high-level specification of the LEAPS algorithms.
- P2 reduced by an estimated factor of three the programming time and by an estimated factor of four the volume of code that had to be written.
- The performance of RL/P2-generated files for the benchmarks that we considered was at least 50% faster than those produced by LEAPS. These results either confirmed suspected performance problems in LEAPS or they identified cleaner, more maintainable ways of implementing future versions of LEAPS.

All of the above was accomplished *without* expertise in expert systems or forward-chaining inference engines. In short, P2 enabled novices (ourselves) to program like domain experts.

Our experiences using P2 have been very encouraging to date. Other reengineering experiments will be needed to better understand the strengths and limitations of P2. We firmly believe, however, that scalable data structure compilers like P2 will be valuable tools of future software engineering environments.

Acknowledgments. We thank David Brant at The Applied Research Laboratories at the University of Texas for supporting our research. We thank Dan Miranker and Bernie Lafaso for their patience in explaining the LEAPS algorithms to us, and we thank Vivek Singhal and Bart Geraci for their insightful comments on earlier drafts of this paper.

6 References

[ACM91] Association for Computing Machinery, "Next Generation Database Systems", *Communications of the ACM*, October 1991.

[Bat88] D. Batory, et al. "Genesis: An Extensible Database Management System", *IEEE Transactions on Software Engineering*, November 1988.

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *Proc. ACM SIGSOFT*, December 1993.

[Bat94a] D. Batory, "The LEAPS Algorithms", unpublished report, May 1994.

[Bat94b] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. "The GenVoca Model of Software System Generators". *IEEE Software*, September 1994.

[Big94] T. Biggerstaff. "The Library Scaling Problem and the Limits of Concrete Component Reuse", *IEEE International Conference on Software Reuse*, November 1994.

[Bra91] D. Brant, T. Grose, B Lofaso, and D. Miranker, "Effects of Database Size on Rule System Performance: Five Case Studies", *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB)*, 1991.

[Bra93] D. Brant and D. Miranker, "Index Support for Rule Activiation", *Proc. ACM SIGMOD*, May 1993.

[Bro94] J. Browne, et al. "A New Approach to Modularity in Rule-Based Programming", Department of Computer Sciences, University of Texas at Austin, April 1994.

[Cog93] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proc. AGARD*, 1993. Also, Technical Report ADAGE-IBM-93-04, IBM Owego, New York, May 1993.

[For81] C. Forgy, *OPS5 User's Manual*, Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[Hei93] J. Heidemann and G. Popek, "File System Development with Stackable Layers", Technical Report CSD-930019, Department of Computer Science, UCLA, July 1993. To appear in *ACM Transactions on Computer Systems*.

[Hut91] N. Hutchinson and L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, January 1991.

[McD78] J. McDermott, A. Newall, and J. Moore, "The Efficiency of Certain Production Systems", *Pattern Directed Inference Systems*, Waterman, Hayes, Roth (ed), Academic Press, New York, 1978.

[Mir90] D. Miranker, D. Brant, B. Lofaso, and D. Gadbois, "On the Performance of Lazy Matching in Production Systems", *Proc. National Conference on Artificial Intelligence*, 1990.

[Mir91] D. Miranker and B. Lofaso, "The Organization and Performance of a TREAT Based Production System Compiler", *IEEE Transactions on Knowledge and Data Engineering*, March 1991.

[Sir93] M. Sirkin, D. Batory, and V. Singhal, "Software Components in a Data Structure Precompiler", *Proc. 15th International Conference on Software Engineering*, May 1993.

[Sir94] M. Sirkin, "A Software System Generator for Data Structures", Ph.D. dissertation, Dept. of Computer Science, University of Washington, 1994.