1997 Symposium on Software Reuse (SSR), 136-145.

Memory Simulators and Software Generators*

Guillermo Jiménez-Pérez and Don Batory

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(gjimenez,batory)@cs.utexas.edu

Abstract

We present results on re-engineering a highly-tuned, hand-coded memory simulator using the P2 container data structure generator. This application was chosen because synthesizing the simulator's data structures would not exploit P2's primary advantage of automatically applying sophisticated code optimization techniques. Thus, we initially believed that using P2 would be an overkill and that P2-generated code would provide no performance advantages over hand-coding. On the contrary, we found that P2 produced more efficient code and that it offered significant advantages to software development that we had had not previously realized.

1 Introduction

Software generators will become invaluable tools for producing high-performance domain-specific software. Conventional monolithic generators, such as lex and yacc, are very good at producing sophisticated code, but the domains for which they work are very restricted. More powerful generators are extensible and can produce high-performance code for very complex domains. The central feature of such generators are open-ended libraries of reusable components. Application software that is to be generated is defined as a composition of components. Because a small number of components can be composed in many different ways, generator libraries are actually compact representations of vast domain-specific application libraries. Thus, selecting (i.e., generating) a high-performance implementation for an application is often straightforward. Another major advantage of generators is application *evolvability*: it is easy to evolve an application merely by redefining its composition of components and regenerating.

Although software generators are very promising, understanding their capabilities and limitations is still in its infancy. We strongly believe that practical knowledge about generators must be collected through experimentation — specifically by reengineering complex, hand-coded applications using generators — to fully appreciate the tradeoffs that are involved.

*This research was supported in part by the Applied Research Laboratories at the University of Texas and Microsoft Research. Guillermo Jiménez was supported by the Instituto Tecnológico y de Estudios Superiores de Monterrey, México.

Biggerstaff and Richter [Big87] observed that there are two rather different kinds of generator technologies: compositional and transformational. Both compose components in similar ways, but the nature of their components are quite different. *Compositional* components encapsulate the code that applications execute at run-time. *Transformational* components encapsulate algorithms that *generate* the code that applications execute at run-time. The advantage of transformational technologies is that domain-specific optimizations can be an integral part of software generation; such optimizations are performed at application generation-time, yielding efficient source code. Compositional technologies generally don't perform domain-specific optimizations (or if they do, the optimizations are performed at application run-time, with a concomitant and sizable run-time overhead). Compositional technologies may be preferred over transformational technologies in domains where domain-specific optimizations play a minimal role in application performance or in domains where components must be composable at application run-time and cannot be limited to static compile-time compositions.

Domain modeling for transformational generators (e.g. DRACO [Nei91] and KIDS [Smi90]) can be quite different than that used for compositional generators (e.g. frameworks [Joh88]). However, the GenVoca design paradigm [Bat92] unifies important aspects of both compositional and transformational approaches by treating them as alternative ways of implementing GenVoca domain models. This is possible because GenVoca components represent stereotypical refinements that occur in a domain. Such refinements can be implemented as "stupid" components (i.e., those that do not encapsulate domain-specific optimizations) of compositional generators, or as "intelligent" components (that do encapsulate such optimizations) of transformational generators.[1] Having a single modeling methodology that decomposes domains into reusable components without a priori commitments to a compositional or transformational implementation, makes GenVoca a very powerful domain modeling methodology.

This paper presents results of reengineering a hand-coded application using the GenVoca data structures generator P2 [Bat93a, Bat94, Tho95], and explores the advantages and disadvantages of using generators over hand-coding. But more importantly, our experiments shed light on the tradeoffs of choosing a compositional implementation over a transformational implementation of a domain model. Our target application is a C++ memory management simulator. The unusual feature of this application is that synthesizing its data structures does not require the sophisticated domain-specific optimizations (e.g., query optimizations) that have previously distinguished the use of P2. Thus, the use of a transformational generator for such an application would seem to be an overkill, or at least put it on a disadvantaged basis for comparison with compositional generators.

For this reason, we expected that P2 would produce code that was about 10-15% slower than code that which is written and optimized by a domain expert. In this paper we present results that show the opposite: that P2 produced code that is almost 10% faster than hand-optimized and we explain the reasons why this is so. Further, we believe that the same performance would have been achieved through the use of compositional generators, so an important result of our work suggests that performance may not be a critical factor in choosing between compositional and transformational implementation approaches; other factors (e.g., dynamic compositions) may be much more important.

[1] We have implemented the GenVoca domain model of container data structures both transformationally (e.g., the P2 generator) and compositionally (the P++ library [Sin96]).

2 The P2 Container Data Structures Generator

P2 is a GenVoca generator of container data structures [Bat93a, Bat94, Tho95]. A *container* is a collection of elements that are instances of a single data type. *Cursors* reference and update elements of a container. Common data structures — arrays, binary trees, ordered lists, etc. — are implementations of containers. Every P2 component implements a specific container data structure or a data structure feature (e.g., memory management, persistent storage, encryption). Thus, compositions of P2 components can define complex data structures with rich sets of features. Examples will be given shortly.

The goal of P2 is to simplify the programming of container data structures. P2 separates the specification of container implementations (e.g., as compositions of P2 components) from programs that access containers. To accomplish this, P2 provides a superset of the C language that adds cursor and container data types, along with operations on these types, as primitives to C. P2 users code their programs in terms of these implementation-independent types. Compositions of P2 components are used to define implementations of container data types and their corresponding cursor types. The advantage of this organization is data structure evolvability: by altering *only* the composition of components and regenerating (i.e., the P2 program itself is otherwise unchanged), it is easy to explore different implementations of P2 containers. [2]

To illustrate the simplicity of coding data structure applications in P2, consider an application that deals with the simulation of page references in memory. `PAGE_TYPE` is a C struct that defines an ordered pair of unsigned long integers (`pageno`, `touches`) that indicate the frequency with which a particular page is referenced during a program execution. A large number of instances of this type will need to be maintained by a simulator. An abbreviated declaration of a P2 container `page_cont` to hold instances of `PAGE_TYPE` is shown below.

```
typedef struct {                    // C struct declaration
 unsigned long pageno;
 unsigned long touches;
} PAGE_TYPE;

container <PAGE_TYPE> page_cont;    // abbreviated declaration of a
                                    // container of PAGE_TYPE instances
```

To retrieve elements (i.e., `PAGE_TYPE` instances) of container `page_cont`, two cursors are used (see below). One (`all_pages`) references all elements of the container, while a second cursor (`selected_page`) references only elements that satisfy the predicate (`touches > 20`). [3]

```
cursor <page_cont> all_pages;       // P2 declaration of a cursor that
                                    // ranges over all elements of page_cont
```

[2] A similar approach using C++ templates is presented in [Sit96]. There component compositions are made through parameterization; plugging and unplugging is done by parameter substitution.

[3] Note that predicates in P2 are expressed by strings: attribute `A` of the element referenced by a cursor is denoted `$.A`. The `$` denotes to P2 the name of the cursor.

```
cursor <page_cont>                  // declaration of a cursor that ranges over
   where "$.touches > 20"           // only those elements of page_cont that
   selected_page;                   // have been touched more than 20 times
```

In general, P2 cursors and containers are parameterized data types. Containers are parameterized by the type of element that is to be stored; cursors are parameterized by the container to be traversed and optionally by a selection predicate and sort criterion. Cursor and container types are first-class; they can be used like any C data type.

P2 offers an (extensible) set of operations on cursors and containers. The code fragment below illustrates the P2 `foreach` construct, which is used to iterate over elements of a container. Once a cursor is positioned, the referenced element can be examined, updated, or deleted.

```
foreach( selected_page )                              // for each selected page
{
  printf("touches: %d \n", selected_page.touches); // print number of touches
  if (selected_page.touches < 100)                ; // examine touches
     delete( selected_page );                         // delete page
}
```

As mentioned earlier, a P2 program is abstract (and hence not executable) because the cursor and container data types have no implementation. P2 components are primitive "building blocks" of their implementation. Consider the following sets `DS` and `MEM` of parameterized P2 components:

```
DS  = { avl[ DS ],          // avl tree
        splay[ DS ],        // splay tree
        dlist[ DS ],        // doubly-linked list
        odlist[ DS ],       // ordered doubly-linked list
        sequential[ MEM ],  // sequential storage
        heap[ MEM ],        // heap storage
        avail[ DS ],        // avail-list memory management
        ... }

MEM = { transient,          // transient memory
        persistent,         // persistent memory
        ... }
```

Each P2 component implements some data structure (e.g., `avl`, `splay`) or data structure feature (e.g., `sequential`, `heap`, `avail`). A composition of P2 components, called a *type equation*, defines a particular implementation of cursor and container data types from a large family of possible implementations. Consider the following type equations and the implementations they define:

```
typex { simple  = avl[ heap[ transient ]];
        complex = splay[ odlist[ array[ persistent ]]];
      }
```

`simple` is a composition of three components. The `avl` component generates code that stores elements of a container in an avl-tree, `heap` allocates space for elements from a heap, and `transient` stores elements in transient memory. Thus, `simple` defines a container implementation that stores elements in an avl tree, whose nodes are allocated from a heap in transient memory.[4]

Similarly, `complex` defines a container implementation that stores elements in a splay tree where splay-tree nodes are linked together in an ordered linked list and the resulting nodes are stored sequentially in persistent memory. As these examples suggest, with a very small number of components, very large and complex families of data structure implementations can be specified.

A type equation is a 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 equations and each equation is at most a few lines long. Thus, tuning and maintaining a P2 program is often a matter of changing a few lines and regenerating [Bat93a, Bat94].

3 Locality of Reference Studies

The Object-Oriented Programming Systems (OOPS) Research Group at The University of Texas at Austin is studying different aspects of memory management in operating systems [Wil95]. One project is to analyze the locality of references in memory hierarchies. The goal is to show that locality of references is a consequence of regularities in both the structure of programs and in how memory allocators map program objects onto virtual address spaces [Wil96].

OOPS research is experimentally driven: references of actual program executions are studied, rather than using randomly generated memory references. (The belief is that randomly generated memory references do not reflect the true usage of memory in real programs). A simulator was written to support the actions of an OS memory manager using an LRU — Least Recently Used — page eviction policy. This simulator, called `LRUsim`, takes as input a *trace file*, which contains a sequence of trace records that indicate the order in which memory references occur. A *trace record* is the pair <operation, address>, where an operation can be a data load, data store, or instruction fetch, and an address refers to a memory location. (This distinction among operations allows different analyses for program instruction references from data references).

Translating word references to page references using a page size parameter, `LRUsim` maintains an LRU ordering of pages. Every time a page is touched, its position (from the head of the queue) is recorded and the page is moved to the front of the queue. The output of the simulator is a sequence of pairs $< p, c >$, where p is the position of a page (from the front of the queue) and c is a count of the number of times that a page in that position has been referenced. This count reflects patterns to repeated references to LRU queue positions, independent of which pages are actually referenced. Such output is important as it reflects the locality characteristics of a program in a way that is independent of any particular memory size, but which can be interpreted with respect to any memory size of interest. For example, for a memory size of m pages, references to queue positions 1 through m represent hits, and references to m and above represent misses. This information is also useful in analyzing memory hierarchies. Assuming that a level in a hierarchy

[4] An avl tree, like ordered lists and splay trees, has an additional parameter — the key field. Key fields are expressed as *annotations* to a container declaration. These annotations are not shown in our declaration of `page_cont` above, but are discussed in [Bat93a].

has n pages, the level below it has k pages, etc., we can see how hits and misses behave in the memory hierarchy by seeing how page hits or misses concentrate between pages 1 and n, pages $n+1$ to $n+k$, etc. Histograms of `LRUsim`'s output reveals if page hits (or misses) concentrate in particular levels of the memory hierarchy.

Several analyses can be performed on the output generated by `LRUsim`, the most common is graphing the behavior of memory references using different page sizes, different queue sizes, different memory sizes, etc. This allows the OOPS group to obtain a better understanding of locality for different OS memory management design parameters and page eviction policies.

Trace files of programs are generated using special tools, such as *Shade* [Cme93], an instruction-set simulator and custom trace generator. Because trace files can be quite large (e.g., several gigabytes in size) and simulation run times can be long (e.g., days), it is vital that the simulator itself be modularized so that different implementations of the queue data structure and its supporting data structures be tried to improve performance. In the next section, we examine the container data structures the OOPS group has chosen to use in `LRUsim` to maximize its efficiency.

4 Data Structures of the Simulator

`LRUsim` has had a long design history. It's most recent incarnation is a set of C++ modules, where each module deals with a particular task in the simulation. Efficiency has been a major concern since the simulator has been known to run for several days for large trace files. Thus, optimizations have been carefully introduced to enhance the simulator's performance. For example, generic algorithms are avoided if performance is compromised. Rather than defining a general class and then subclassing to implement similar, yet different, functionalities, code is replicated (with slight changes) to deal with particular needs. This has the advantage that customized algorithms are always used and performance degradation resulting from inheritance (i.e., virtual dispatching) is not introduced.

C++ templates were used as the means by which `LRUsim` data structures were modularized. The goal was to define a common interface for a set of different container data structures. This would enable experimentation with different structures — a limited version of that offered by P2 — simply by swapping one data structure template for another. Templates have the advantage that they don't compromise run-time efficiency, while permitting genericity [Str91]. In the following paragraphs, we explain the `LRUsim` data structures and their interrelationships.

The core of `LRUsim` is the module that implements the LRU mechanism, called `LRU_mech`. `LRU_mech` maintains the LRU ordering of memory page references (touches). Figure 1 shows the three basic container data structures that are used. One structure, called the *Queue*, stores page numbers in LRU order. A second structure, called the *Index*, provides a fast way of locating a page in the Queue given its page number. The Index is a composite data structure consisting of a hash table and a set of avl trees, one tree per hash bucket. A third structure, called the *Count*, maintains <position, count> statistics, and is implemented much like the Index. [5]

[5] The OOPs group calls the Queue structure the "Position Manipulation Mechanism", the Index is the "Page Manipulation Mechanism", and the Count is the "Counting Mechanism". We've chosen different names to make their design easier to understand.

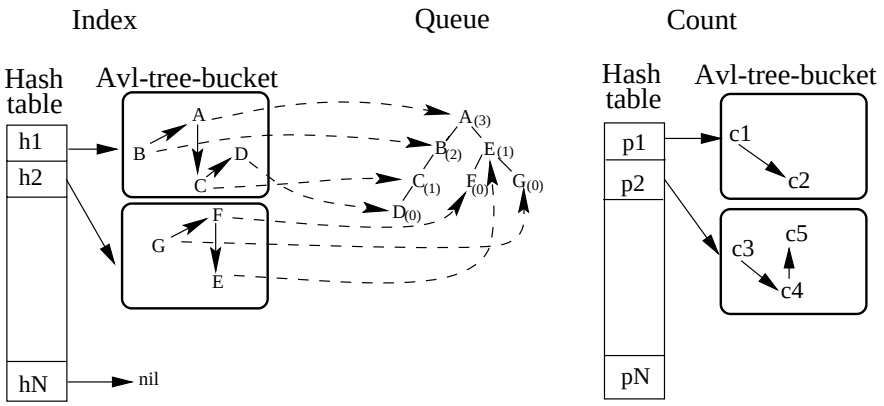

Figure 1: Data structures in the original implementation.

The *Queue* maintains pages in LRU order using a modified avl-tree. This tree does not use a key for insertions. Rather, every new node becomes the left-most leaf of the tree (i.e., it is being inserted at the front of the queue). When a page is referenced, it is unlinked from its current position and relinked at the head of the queue. Thus, unlike traditional queues where node deletions occur at the ends of a queue, node deletions can occur anywhere.

Using an avl-tree to represent a queue has advantages over single or double-linked lists. The main advantage is that the relative position of a node in the queue can be obtained in $O(\log n)$ time, instead of $O(n)$ time. To accomplish this, every node maintains the size of its left sub-tree. The numbers between parentheses in Figure 1 are the left sub-tree sizes for each node. Given these sizes, it is easy to obtain the position of a node n from the front of the queue by traversing the tree from n to the root. For example, the position of node `G` in Figure 1 is $(0+1)+(1+1)+(3+1) = 7$.

The modified avl tree, henceforth called a *position_avl* tree, associates positions with each node. Since it does not use keys for insertion, there needs to be another mechanism for finding a particular page (avl-node) quickly given its page number. The Index (see Figure 1) is used for this purpose.

The *Index* is a two-level data structure. The first level is a hash table, and the second level is a set of avl trees, one avl tree for each hash bucket (Figure 1). Unlike position_avl trees, these avl-trees are "normal". Each avl-tree node has a page number and a pointer to the Queue node that defines that page. The page number is the key of the node. (Note that decreasing the number of buckets would increase the size of the avl trees, since more collisions would occur).

To determine if a page is in memory, the page number is hashed to a bucket. The avl-tree associated with that bucket is searched for the node of that given page number. If the node is found, the pointer to the page in the Queue is followed and the Queue's node is moved to the front of the Queue. If the node is not found, a new node is added to the front of the Queue, and another node is inserted into the appropriate bucket/avl-tree (with a pointer to its corresponding Queue node).

The third data structure, the *Count*, (also shown in Figure 1), maintains the reference of touches to specific positions in the LRU queue. It is similar to the Index, but does not require pointers to Queue nodes. (Position references are hashed to a bucket, and the bucket of <position,count> pairs is searched, and the count is incremented). Note that a fixed-size array is another possible implementation for Count. The reason why an avl-tree implementation is used is because the maximum size of the array (i.e., the maximum value of position) is generally not known. In

general, the output of LRUsim is a dump of the Count data structure.

5 Simulator Implementation in P2

Approximately 60 percent of the code in LRUsim deals with data structures; the remaining deals with I/O and data type conversion. Due to the modularization of LRUsim, and the fact that it was written in C++, it was unnecessary for us to re-write the whole application in P2: few (if any) advantages would have been gained. Instead, we used P2 to generate the LRUsim data structures in C (in the module LRU_mech.h) and then we linked these structures directly to LRUsim. In this section, we show that implementing these data structures was straightforward and present our benchmarking results.

P2 provides a standard, high-level interface to all container data structures. Initially we thought that LRU_mech would require customized operations to the Queue, Index, and Count data structures because of their specialized semantics. However, we discovered that any operation that we needed could be expressed easily as calls to P2 container and cursor operations.

Some changes to P2 were needed, however. There was no P2 component that implemented the position_avl tree algorithm, and hence we had to write this component (denoted position_avl). We adapted the avl-tree algorithm from HPUX[6] to conform to the interface and protocol standards of P2 components; this took approximately three weeks to write and debug.[7] We feel that if we were not constrained by the requirements imposed by P2 in writing components, it would have simplified our task slightly (e.g., by a few days). So the effort needed to extend P2 was not great.

From P2's point of view, there are three types of container data structures in LRU_mech: Queue, Hash, and Bucket. (The Index and Count data structures each are composites of Hash and Bucket). Specifying these containers in P2 was straightforward. Consider the declaration of the Queue container:

```
typedef struct {
   unsigned long   page;                    // page number
}  Queue_Element;

typex Queue_eq  =  position_avl[avail[heap[transient]]];

container <Queue_Element, Queue_eq>  Queue;  // container declaration
```

Queue_Element is a C struct that declares the type of element to be stored in a container. (It consists of a single page number). The type equation Queue_eq defines a container that stores elements in a position_avl tree, where nodes of this tree are memory managed (i.e., they are not physically deleted, but are recycled upon deletion), and when physically allocated are drawn from a heap in transient memory. The Queue container, Queue, is defined in a line that unites

[6] AVL Tree V2.0, 22-January-1993 is available in http://hpux.ask.uni-karlsruhe.de/hpux/Languages/avl-2.0.html

[7] The three weeks included testing, recognizing that an available avl tree component in P2 had bugs and had to be scrapped, etc. So the development time for position_avl was intertwined with many activities.

Queue_Element and Queue_eq together. Similarly simple declarations were needed for the Hash and Bucket containers. [8] [9]

Once the P2 containers were declared, the next step was to implement simulator actions in terms of operations on P2 containers, and then to link the P2-generated code (the P2LRU_mech.h module) to the C++ LRUsim source. The original hand-crafted containers of LRU_mech were implemented as classes, whereas P2 generated code was expressed in C functions. Thus, some simple modifications to LRUsim were required: C++ member function calls were replaced by their corresponding C function calls.

The diagram in Figure 2 shows the steps necessary to generate a P2LRU_mech.h module (the P2-generated module implementing the LRU policy). P2 generates the desired C source from a P2 program (LRU_mech.p2). The resulting P2LRU_mech.c code is then modified slightly via a sed script so that it can be compiled with C++ and linked with LRUsim and other C++ modules.

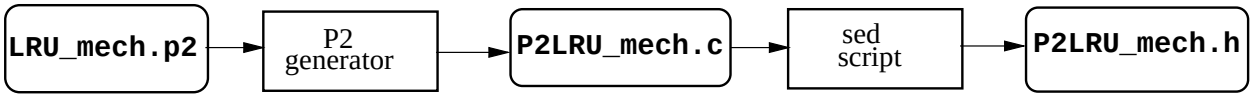

Figure 2: Generating an LRU mechanism module.

5.1 Run-Time Comparisons

One of the reasons why we chose LRUsim over other possible applications is that the LRUsim data structures were rather simple. One of the great advantages that we have observed for transformational generators is their ability to perform sophisticated domain-specific optimizations and customizations automatically. The data structures and search algorithms of LRUsim were not complicated enough to invoke such optimizations (e.g., query optimizations) and thus, we expected the code generated by P2 to be slower than that of the hand-crafted and hand-tuned code for the data structures in LRUsim. To our surprise, P2 code executed faster.

To evaluate the performance of P2 generated code, we were given trace files by the OOPS group that they used previously in their performance studies. Our first benchmarks were the trace files, tex, cc1, and spice, that are commonly used to analyze different aspects related to memory usage [Hen90]. These files are relatively small compared to more "industrial strength" trace files p2c, expresso, and ghostscript, which are two orders of magnitude larger [Wil95, Wil96]. Running times for both the original C++ and P2 versions of LRUsim are shown in Table 1.

Although the original LRUsim C++ code had undergone several revisions and significant manual tunings, our first version using P2 ran a surprising 30 percent faster on all benchmarks. Upon profiling both implementations, we observed that the C++ Count data structure was cleaned each

[8] The type equations for the Hash and Bucket containers are: Hash_eq = hash[heap[transient]] and Bucket_eq = avl[heap[transient]], respectively.

[9] The need for memory management (i.e., garbage collection) in Queue_eq stems from the fact that P2 containers do not support element unlink and relink operations. (Recall that when a page is touched in the Queue, it is removed (unlinked) from its current position and relinked at the head of the Queue). Relinking and unlinking were emulated using existing P2 operations by deleting the element and then reinserting. The avail layer improved the performance of this operation sequence by never physically discarding the element slot thereby avoiding the overhead of heap allocation and heap deallocation. This particular problem did not arise in the Index and Count data structures.

FILE	Number of Instructions		Running Time		%Speedup
	Load	Store	P2	C++	
tex	130K	104K	100	130	30
spice	150K	66K	115	153	33
cc1	159K	83K	124	163	31
p2c	59.8M	16.4M	51.7K	68.4K	32
ghostscript	75.7M	37M	64K	84.2K	31
espresso	414M	132M	378K	482K	27

NOTE: K = thousand, M = million.

Table 1: Benchmark Execution Times (in seconds).

time its contents were dumped to an output file. Cleaning is invoked when one kind of instruction is read from the input (users can indicate if the instruction is a load, store, or fetch — our benchmarks triggered cleanings on loads). All of our test files were dominated by load instructions (see Table 1); this meant that most of the time, the Count structure contained only a few records. But Count's implementation in C++ consists of 1024 buckets, where each bucket was an avl tree. Whenever Count was cleaned, every bucket was assumed to contain records, so expensive functions to clean empty buckets were called unnecessarily millions of times.

We learned an important lesson from this experiment: there are different ways in which data structure code (or domain-specific code, in general) can be optimized. Our previous experience with transformational generators encouraged us to focus our attention on domain-specific optimizations that could be performed automatically. For example, P2 performs sophisticated query optimizations that manipulate predicates and chooses the most efficient data structure to traverse for a given predicate. Since these optimizations were absent from `LRUsim`, (i.e., there was only one container data structure in any container implementation and the selection predicates were always null), there must have been other optimizations at work. We discovered that these optimizations were actually coding "tricks" — also known as *best-practice approaches* — in the programming of data structure algorithms. Thus the way algorithms are programmed has a significant impact on performance, and evidently the way algorithms were coded in P2 components was quite efficient. Stated differently, generators glue prewritten algorithms together. The "domain-specific optimizations" of generators are simply decisions generators make to select the fastest algorithm from a competing set of algorithms. However, if the algorithms themselves are inefficiently programmed, generators cannot possibly synthesize efficient code. Thus, to generate efficient code in general requires both efficient algorithm implementations *and* domain-specific optimizations.

Generator-synthesized algorithms have a higher probability of performing better than hand-coded algorithms because the author of generator components is focussing his/her attention on the most efficient encoding/implementation of a highly-constrained problem (in the case of P2, a particular data structure algorithm). Consequently, *much* more effort is focussed on optimizing what might otherwise appear to be "minute" coding details. In contrast, when a complex data structure — that corresponds to compositions of multiple P2 components — is coded from scratch, there are *so* many implementation details to keep straight that it is utterly impractical and far too time consuming to optimize code at the same level of detail. Programmers will attempt only a fraction of these optimizations. For this reason, generator components (and their compositions) often out-perform hand-written code.

FILE	P2	C++	%Speedup
tex	100	108	7.7
cc1	122	136	11.4
spice	116	124	7.5
p2c	51.7K	56.8K	9.8
ghostscript	64K	66.3K	4.0
espresso	378K	402.2K	6.4

NOTE: K = thousand.

Table 2: Revised Benchmark Execution Times (in seconds).

To confirm this insight, we repaired the C++ implementation, and reran the C++ benchmarks (see Table 2). Although the reparation improved the performance of the C++ version by over 20 percent, the P2 version still ran faster. (Although 5 to 10 percent improvement may not seem significant, it is still noticeable. For example, the P2 execution of the `p2c` benchmark completed 1.3 hours *faster* than the C++ version). Again, the differences can be attributed to more efficient encodings of algorithms. More specifically we found:

- Avl tree traversals were implemented through function recursion in C++. P2 algorithms were iterative, and were faster by a constant factor. In hindsight, the OOPS group realized that recursion was chosen in the C++ case for reasons of clarity and maintenance, rather than speed. Iterative implementations were used in P2 because (a) iterative implementations were believed to be faster and (b) iteration loops are small (because their focus is restricted to data structure algorithms), hence code clarity and maintenance was not sacrificed.
- many of the low-level "tricks" that are used in P2 to encode data structure algorithms efficiently (e.g., no recursion, aggressive inlining) could indeed be introduced into the C++ version of `LRUsim`. But the OOPS group recognized that they wouldn't attempt such a level of optimization because the resulting code would be "horrendous to maintain and understand". Moreover, they felt that it would be much too difficult to propagate these changes in their hand-written code.

5.2 Different Data Structures

As an interesting secondary experiment, we explored the evolvability of P2 container implementations. Altering data structures required that we change only a couple lines of our P2 program (i.e., the container type equations), and then regenerate the C code. Table 3 shows running times with different implemenations of the Hash container. Note that the avl tree had the fastest execution times for these benchmarks.

Although these experiments didn't reveal better ways to store `LRUsim` records, it did demonstrate an important capability of P2. Software designs that use particular data structures have built-in assumptions about how data is to be used and accessed. With few exceptions, once the target software is implemented and tested on an actual work load, some of the original design decisions are recognized to be sub-optimal. At this point, software designers face two unpleasant options: either leave the data structures as they are, knowing that improved performance may be sacrificed, or

DATA STRUCTURE	FILE					
	tex	spice	cc1	p2c	ghostscript	espresso
avl	100	115	124	62K	81K	378K
binary	107	121	132	67.4K	84.7K	417K
splay	106	123	128	64.4K	83.2K	467K

NOTE: K = thousand.

Table 3: Benchmark Execution Times for Different Data Structures (in seconds).

redesign, recode, and reoptimize the data structures for yet another round of testing. P2 *substantially* reduces the cycle time for testing new data structures. To alter a type equation and recompile a P2 application takes minutes; to redesign, recode, and reoptimize a data structure by hand can take weeks. Thus, to confirm or improve design choices for using particular data structures is quite easy with P2. [10]

5.3 Introspection

Code Quality. We learned several important lessons from these experiments which we believe apply to all GenVoca generators. GenVoca generators compose locally optimized components (code fragments) to produce efficient implementations for stereotypical programming problems in a particular software domain. (P2 produces container and cursor implementations for data structures). Although components may generate locally optimal code, code that is produced by composing locally optimal fragments is not guaranteed to be globally optimal. Thus, it is always possible that hand-coded applications can perform better than generated code. However, the issue is "how much effort is needed to do better than generated code?". The experiments in this paper (which in hindsight is also supported by our previous experiences [Bat93a, Bat94]) suggest that level of effort to optimize code manually to achieve the performance of generated code is far greater than any programmer would be willing to invest. Composing locally optimal components produces *very* good software that for practical purposes is noticeably better on average than what can be produced by hand.

Productivity. To place our observations on code quality in perspective, one should keep in mind that the greatest benefit of using generators are the gains in productivity and the reduction of software maintenance. While it is difficult for us to determine the exact gains (e.g., `LRUsim` has a long history of development, we were unfamiliar with memory management simulators before we began our work, etc.), it is possible to get a feeling for the productivity benefits of using P2 by examining the sizes of the respective source files. The functionality of the `LRU_mech.p2` file is spread across multiple files of the C++ `LRUsim` implementation. From our best estimates, the C++ implementation is approximately 6650 lines of source code, whereas the P2 version is only 2550 lines, a 60 percent reduction in code volume. A factor of 2-3 improvement in productivity appears to be consistent with other uses of P2 [Bat94]. That generators also produce efficient code (for the reasons we cited earlier) makes their use that much more attractive.

Software Maintainability. Generators can substantially improve software understandability, quality, and maintainability. An algorithm is easier to understand and maintain if it has a compact

[10] A similar approach for component substitution was advocated in [Sit96], with emphasis on performance evolution.

specification as a short function instead of a collection of collaborating functions. The P2 functions that define the LRU mechanism are very small. The largest has only 52 lines of code. However, this same function is expanded to about *thirteen hundred* lines in the generated C code, and is equivalent to many small functions in the C++ version of `LRUsim`. By raising the level of abstraction, P2 substantially reduces the level of complexity in which programmers must code. This in turn enhances program understandability and maintainability. In our post-benchmark interviews with the OOPS group, they cited the ability to compactly write very complicated code as the major benefit of P2. They believed that it would allow them to revise and restructure code in ways that would otherwise be difficult, if not impossible.

Inlining. On a related note, we noticed that there was a performance advantage in generating large functions. For data structure applications, where functions often involve small computations, the overhead of function calls becomes significant. Because P2 aggressively inlines, most function boundaries are eliminated. While tagging all (or most) C++ functions with the `inline` directive might accomplish a similar goal, often programmers fail to make these directives explicit in their source code. Not doing so leads to degraded performance. P2, on the other hand, automatically inlines.

Implementation Approaches. Another lesson is that if there is a choice between a compositional and transformational implementation of a GenVoca domain model (because domain-specific optimizations are not important), we know of no performance reason why a compositional implementation should be chosen over a transformational implementation. All of the benefits of efficient encodings of algorithms can be achieved in both compositional and transformational implementations. As we mentioned earlier, the `LRUsim` application was chosen specifically because it put transformational generators in a negatively-biased position. Despite this, P2 still did quite well. Our experiments suggest that reasons other than performance, such as the ability to compose components at application run-time rather than statically at application compile-time, should be the determining factor.

Drawbacks. While there are many important advantages to generators, there are drawbacks. At present, transformational generators are closely integrated with specific programming languages. The P2 programming language, for example, implements a superset of C. If another transformational generator, say for the domain of network protocols, were to be written and were to also extend C, it is unclear whether a single program could reference both the cursor and container data types of P2 and the data types of the protocol generator. Thus, applications that use both data structures and network protocols would have to choose to code their application in P2 (thus forcing them to hand-code protocols) or to code their application using the protocol generator (thus forcing them to hand-code data structures). This is clearly unacceptable; we now believe that this inability for generators like P2 to interoperate with other generators is a serious shortcoming.

A more general approach to the construction of transformational generators is as extensions to extensible languages/compilers. A domain-specific generator should be encapsulated as one or more packages that *extends* a specific language and its compiler. By linking generator packages to the compiler, the programming language will be enhanced (i.e., new domain-specific data types and programming constructs will become available). Thus, to create a programming language that generates implementations for both data structures and network protocol abstractions, one would need to link both a data structure generator and protocol generator to the target compiler. Thus, as long as there are no ambiguities in the linguistic extensions that are being made, generators for different domains can be plugged and unplugged from the compiler. This advanced vision of

software componentry is being realized by the IP project at Microsoft Research [Sim95, Sma96] and the Jakarta project at the University of Texas at Austin.

6 Conclusions

The power, capability, and potential of software generators continues to surprise us. We selected a data structure application — a memory simulator — for re-engineering that we expected a hand-coded approach would produce more efficient software. The reason for us believing so was that our previous experience with generators suggested that the primary advantage of generators was their ability to perform difficult, domain-specific optimizations automatically. The data structures of the memory simulator were so simple that none of these optimizations were needed. Thus, we had no a priori reason to believe that generators could produce more efficient code.

Our experiments demonstrated the contrary: generated code was more efficient. The reason is that components of generator libraries encapsulate locally-optimized algorithms, where the focus of component writers is to optimize what would otherwise be considered "minute" coding details. Although composing locally-optimized components does not guarantee globally optimized code, the code that is produced is indeed very good. We observed that good programmers can only attempt a fraction of all the optimizations that are present within components, and primarily for this reason, generated programs perform better than hand-written programs. Thus, it would seem that generators have a built-in advantage over programmers for software development: components can encapsulate best-practice approaches for producing efficient domain-specific software that no single programmer (or team of programmers) will be able to duplicate with any economy. Although the greatest benefit of using generators is gains in productivity and a reduction of software maintenance, that generators produce efficient code makes their use that much more attractive.

Another important result from our work is insight in choosing between a compositional implementation of a domain model or a transformational implementation. Our results suggest that when both compositional and transformational implementations can be considered, performance of the generated code may not be the deciding factor. We believe that the performance we observed using P2 (where domain-specific optimizations were never invoked) could be achieved by compositional implementations. Thus, other factors — whether components need to be composed dynamically or not — are more important. Defining these factors more precisely is a subject of future work.

Finally, it has become evident that one of the key obstacles to the proliferation of generators like P2 is the way that they are built. Presently, they are implemented as extensions to existing programming languages. This approach essentially makes them non-interoperable with other generators (that extend a possibly different language). A key problem for GenVoca generators is building extensible programming languages that can modularly support the addition of multiple generator technologies in a seamless fashion. This too is a subject of our future work [Sma96].

Acknowledgements. We are grateful for the help, advice, and time that Paul Wilson, Doug van Wieren, Mark Johnstone, and other members of the OOPS group have contributed to our work.

References

[Bat92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Big87] T. Biggerstaff and C. Richter. Reusability framework, assessment and directions. *IEEE Software*, pages 41–49, March 1987.

[Bat93a] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. *Proc.ACM SIGSOFT'93*, December 1993.

[Bat93b] D. Batory and D. Vasavada. Software components for object-oriented database systems. *International Journal of Software Engineering and Knowledge Engineering*, 3(2):165–192, 1993.

[Bat94] Don Batory, Jeff Thomas, and Marty Sirkin. Reengineering a complex application using a scalable data structure compiler. *Proc. ACM SIGSOFT*, 1994.

[Cme93] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, University of Washington, 1993.

[Hen90] John L. Hennessy and David A Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[Joh88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[Nei91] J. Neighbros. Draco: A method for engineering reusable software systems. In R. Prieto-Diaz and G . Arango, editors, *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.

[Sim95] C. Simonyi. The death of computer languages, the birth of intentional programming. *Microsoft Research. Also, NATO Science Committee Conference*, September 1995.

[Sin96] Vivek Singhal. *P++: A Language for Large-Scale Reusable Software*. PhD thesis, Dept. of Computer Sciences, The University of Texas at Austin, 1996.

[Sit96] S. Sreerama, D. Fleming, and M. Sitaraman. Graceful object-based performace evolution. Technical report, West Virginia University, 1996.

[Sma96] Y. Smaragdakis and D. Batory. Distil: A transformation library for data structures. Technical Report in progress, Department of Computer Sciences, The University of Texas at Austin, 1996.

[Smi90] D. R. Smith. Kids: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, September 1990.

[Str91] Bjarne Stroustrup. *The C++ Programming Language, second edition*. Addison-Wesley, 1991.

[Tho95] Jeff Thomas and Don Batory. P2: An extensible lightweight dbms. Technical Report TR-95-04, Department of Computer Sciences, The University of Texas at Au stin, February 1995.

[Wil95] Paul Wilson, Mark Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. *International Workshop on Memory Management*, September 1995.

[Wil96] Paul Wilson. Locality of reference, patterns in program behavior, memory management, and memory hierarchies. Technical report, The University of Texas at Austin, 1996.