

Reconsidering Custom Memory Allocation

Emery D. Berger
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01002
emery@cs.umass.edu

Benjamin G. Zorn
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Kathryn S. McKinley
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
mckinley@cs.utexas.edu

ABSTRACT

Programmers hoping to achieve performance improvements often use custom memory allocators. This in-depth study examines eight applications that use custom allocators. Surprisingly, for six of these applications, a state-of-the-art general-purpose allocator (the Lea allocator) performs as well as or better than the custom allocators. The two exceptions use regions, which deliver higher performance (improvements of up to 44%). Regions also reduce programmer burden and eliminate a source of memory leaks. However, we show that the inability of programmers to free individual objects within regions can lead to a substantial increase in memory consumption. Worse, this limitation precludes the use of regions for common programming idioms, reducing their usefulness.

We present a generalization of general-purpose and region-based allocators that we call *reaps*. Reaps are a combination of regions and heaps, providing a full range of region semantics with the addition of individual object deletion. We show that our implementation of reaps provides high performance, outperforming other allocators with region-like semantics. We then use a case study to demonstrate the space advantages and software engineering benefits of reaps in practice. Our results indicate that programmers needing fast regions should use reaps, and that most programmers considering custom allocators should instead use the Lea allocator.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Dynamic storage management

General Terms

Algorithms, Experimentation, Performance, Reliability

1. Introduction

Programmers seeking to improve performance often incorporate custom memory allocators into their applications. Custom allo-

cators aim to take advantage of application-specific allocation patterns to manage memory more efficiently than a general-purpose memory allocator. For instance, 197.parser (from the SPECint2000 benchmark suite) runs over 60% faster with its custom allocator than with the Windows XP allocator [4]. Numerous books and articles recommend custom allocators as an optimization technique [7, 25, 26]. The use of custom memory allocators is widespread, including the Apache web server [1], the GCC compiler [13], three of the SPECint2000 benchmarks [34], and the C++ Standard Template Library [11, 32], all of which we examine here. The C++ language itself provides language constructs that directly support custom memory allocation (by overloading operator `new` and `delete`) [10].

The key contributions of this work are the following. We perform a comprehensive evaluation of custom allocation. We survey a variety of applications that use a wide range of custom allocators. We compare their performance and memory consumption to general-purpose allocators. We were surprised to find that, contrary to conventional wisdom, custom allocation generally does not improve performance, and in one case, actually leads to a performance degradation. A state-of-the-art general-purpose allocator (the Lea allocator [23]) yields performance equivalent to custom memory allocators for six of our eight benchmarks. These results suggest that most programmers seeking faster memory allocation should use the Lea allocator rather than writing their own custom allocator.

The custom allocators that do provide higher performance both use *regions*. Regions provide high-performance but force the programmer to retain all memory associated with a region until the last object in the region dies [14, 15, 17, 19, 30, 39]. We show that the performance gains of regions (up to 44%) can come at the expense of excessive memory retention (up to 230%). More importantly, the inability to free individual objects within regions greatly complicates the programming of server applications like Apache which rely on regions to avoid resource leaks. Many programs cannot use regions because of their memory allocation patterns. If programs with intensive memory reuse, producer-consumer allocation patterns, or dynamic arrays were to use regions, they could consume very large or even unbounded amounts of memory.

We present a generalization of regions and heaps we call *reaps*. Our implementation of reaps provides the performance and semantics of regions while allowing programmers to delete individual objects. We show that reaps nearly match the speed of regions when used in the same way, and provide additional semantics and generality. Reaps provide a reusable library solution for region allocation with competitive performance, the potential for reduced memory consumption, and greater memory management flexibility than regions. We demonstrate individual object deletion using reaps with a case study in which we add a new module to Apache. The orig-

This work is supported by NSF ITR grant CCR-0085792 and Darpa grant F33615-01-C-1892. This work was done while Emery Berger was a research intern at Microsoft Research and a doctoral student at the University of Texas. Emery Berger was also supported by a Microsoft Research Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

Copyright 2002 ACM 1-58113-417-1/02/0011 ..\$5.00

inal version of this program uses `malloc/free`. We show that by modifying only a few lines to use the `reap` interface, we can get region semantics with individual object deletion and thus reduce memory consumption significantly.

The remainder of this paper is organized as follows. We discuss related work in Section 2. We describe our benchmarks in Section 3. In Section 4, we analyze the structure of custom memory allocators used by our benchmark applications and explain why regions do not provide sufficient support for many applications, in particular server applications like Apache. In Section 5, we describe reaps and present our implementation in detail. We describe our experimental infrastructure and methodology in Section 6. In Section 7, we present experimental results, including a comparison to previous allocators with region-like semantics, and present our case study. We discuss our results in Section 8, explaining why we believe programmers used custom memory allocators despite the fact that these do not provide the performance they promise, and we conclude in Section 9.

2. Related Work

Numerous articles and books have appeared in the trade press presenting custom memory allocators as an optimization technique. Bulka and Mayhew devote two entire chapters to the development of a number of custom memory allocators [7]. Meyers describes in detail the use of a freelist-based per-class custom allocator in “Effective C++” [24] and returns to the topic of custom allocators in the sequel [25]. Milewski also discusses per-class allocators as an optimization technique [26]. Hanson devotes a chapter to an implementation of regions (“arenas”), citing both the speed and software engineering benefits of regions as motivation [20]. Ellis and Stroustrup describe the syntactic facilities that allow overloading operator `new`, simplifying the use of custom allocators in C++ [10], and Stroustrup describes per-class allocators that use these facilities [37]. In all but Hanson’s work, the authors present custom memory allocation as a widely effective optimization, while our results suggest that only regions yield performance improvements. We present a generalization of custom allocators (reaps) and show that reaps capture the high performance of region allocators.

Region allocation, variously known as arenas, groups, and zones [19, 30] has recently attracted attention as an alternative to garbage collection. Following the definitions in the literature, programmers allocate objects within a region and can delete all objects in a region at once but cannot delete individual objects [14, 15, 17, 19, 30, 39]. Tofte and Talpin present a system that provides automatic region-based memory management for ML [39]. Gay and Aiken describe *safe* regions which raise an error when a programmer deletes a region containing live objects and introduce the RC language, an extension to C that further reduces the overhead of safe region management [14, 15]. While these authors present only the benefits of regions, we investigate the hidden memory consumption cost and limitations of regions and present an alternative that avoids these drawbacks and combines individual object deletion with the benefits of regions.

To compute the memory cost of region allocation, we measure the memory consumed using regions and when objects are freed immediately after their last reference. We use binary instrumentation to determine when objects are last referenced and post-process a combined allocation-reference trace to obtain peak memory consumption and object drag, the elapsed time between last use and reclamation of an object. Our definition of drag differs slightly from the original use of the term by Runciman and Rojemo [31]. In their work, drag is the time between last use and unreachability of an object, which in a garbage-collected environment defines availability for reclamation. Shaham, Kolodner and Sagiv measure drag

by performing periodic object reachability scanning in the context of Java, a garbage-collected language [33].

The literature on general-purpose memory allocators is extensive (see Wilson’s survey for a comprehensive description [43]). Here we describe the Windows XP and Lea allocators [23, 28], which we use in this study because of their widespread use (the Lea allocator forms the basis of the Linux memory allocator [16]). The Windows allocator is a best-fit allocator with 127 exact-size quicklists (one linked list of freed objects for each multiple of 8 bytes), which optimize for the case when many requests are for small same-sized objects. Objects larger than 1024 bytes are obtained from a sorted linked list, sacrificing speed for a good fit. The Lea allocator is an approximate best-fit allocator with different behavior based on object size. Small objects (less than 64 bytes) are allocated using exact-size quicklists. Requests for a medium-sized object (less than 128K) and certain other events trigger the Lea allocator to *coalesce* the objects in these quicklists (combining adjacent free objects) in the hope that this reclaimed space can be reused for the medium-sized object. For medium-sized objects, the Lea allocator performs immediate coalescing and *splitting* (breaking objects into smaller ones) and approximates best-fit. Large objects are allocated and freed using `mmap`. The Lea allocator is the best overall allocator (in terms of the combination of speed and memory usage) of which we are aware [22].

In addition to the standard `malloc/free` interface, Windows also provides a Windows-specific memory allocation interface that we refer to as Windows Heaps (all function calls begin with `Heap`). The Windows Heaps interface is exceptionally rich, including multiple heaps and some region semantics (but not nested regions) along with individual object deletion [28]. `Vmalloc`, a memory allocation infrastructure, also provides (non-nested) regions that permit individual object deletion [41]. We show in Section 7.3 that neither of these implementations match the performance of regions or reaps, and reaps capture the same semantics.

The only previous work evaluating the impact of custom memory allocators is by Zorn (one of the authors). Zorn compared custom (“domain-specific”) allocators to general-purpose memory allocators [44]. He analyzed the performance of four benchmarks (cfrc, gawk, Ghostscript, and Perl) and found that the applications’ custom allocators only slightly improved performance (from 2% to 7%) except for Ghostscript, whose custom allocator was outperformed by most of the general-purpose allocators he tested. Zorn also found that custom allocators generally had little impact on memory consumption. His study differs from ours in a number of ways. Ours is a more comprehensive study of custom allocation, including a benchmark suite covering a wide range of custom memory allocators, while Zorn’s benchmarks include essentially only one variety.¹ We also address custom allocators whose semantics differ from those of general-purpose allocators (e.g., regions), while Zorn’s benchmarks use only semantically equivalent custom allocators. Our findings therefore differ from Zorn’s, in that we find that certain custom allocators (especially regions) consistently yield performance improvements over existing general-purpose memory allocators, despite the fact that the general-purpose allocators are much faster now.

While previous work has either held that custom memory allocators are a good idea (articles in the trade press), or a waste of time (Zorn), we find that both are true. Most custom allocators have no impact on performance, but regions in particular have both high performance and some software engineering benefits. We show that the inability of programmers to delete objects within regions may

¹These allocators are all variants of what we call per-class allocators in Section 4.2.

Benchmarks		
custom allocation		
<i>197.parser</i>	English parser [34]	<i>test.in</i>
<i>boxed-sim</i>	Balls-in-box simulator [8]	<i>-n 3 -s 1</i>
<i>C-Breeze</i>	C-to-C optimizing compiler [18]	<i>espresso.c</i>
<i>175.vpr</i>	FPGA placement & routing [34]	test placement
<i>176.gcc</i>	Optimizing C compiler [34]	<i>scilab.i</i>
<i>apache</i>	Web server [1]	see Section 3
<i>lcc</i>	Retargetable C compiler [12]	<i>scilab.i</i>
<i>mudlle</i>	MUD compiler/interpreter [14]	<i>time.mud</i>
general-purpose allocation		
<i>164.gzip</i>	GNU zip data compressor [34]	<i>test/input.compressed 2</i>
<i>181.mcf</i>	Vehicle scheduler [34]	<i>test-input.in</i>
<i>186.crafty</i>	Chess program [34]	<i>test-input.in</i>
<i>252.eon</i>	Ray tracer [34]	<i>test/chair.control.cook</i>
<i>253.perlbnk</i>	Perl interpreter [34]	<i>perfect.pl b 3</i>
<i>254.gap</i>	Groups language interpreter [34]	<i>test.in</i>
<i>255.vortex</i>	Object-oriented DBM [34]	<i>test/lendian.raw</i>
<i>300.twolf</i>	CAD placement & routing [34]	<i>test.net</i>
<i>espresso</i>	Optimizer for PLAs [35]	<i>test2</i>
<i>lindsay</i>	Hypercube simulator [43]	<i>script.mine</i>

Table 1: Benchmarks and inputs. We include the general-purpose benchmarks for comparison with custom allocation in Section 7.3. All programs are written in C except C-Breeze and 252.eon, which are written in C++.

lead to a substantial increase in memory consumption and limits their applicability. We develop a generalized memory allocator that preserves the high performance of regions while providing greater flexibility and a potential reduction in memory consumption.

Regions have also been incorporated into Real-Time Java to allow real-time guarantees that cannot be provided by any existing garbage collector algorithm or implementation [5]. These regions, while somewhat different from traditional region-based allocators in that they are associated with one or more computations [2], suffer from the same problems as traditional regions. In particular, threads in a producer-consumer relationship cannot use region allocation without causing unbounded memory consumption. We believe that adapting reaps to the setting of Real-Time Java is a fruitful topic for future research.

3. Benchmarks

We list the benchmarks we use in this paper in Table 1, including general-purpose allocation benchmarks that we use for comparison with custom allocation in Section 7.3. Most of our benchmarks come from the SPECint2000 benchmark suite [34]. For the custom allocation benchmarks, we include a number of programs used in prior work on memory allocation. These programs include those used by Gay and Aiken (Apache, lcc, and mudlle) [14, 15], and boxed-sim, used by Chilimbi [8]. We also use the C-Breeze compiler infrastructure [18]. C-Breeze makes intensive use of the C++ Standard Template Library (STL), and most implementations of the STL use custom allocators, including the one we use in this study (STLport, officially recommended by IBM) [11, 32].

We use the largest inputs available to us for most of the custom allocation benchmarks, except for 175.vpr and 197.parser. For these and the general-purpose benchmarks from SPEC2000, we used the test inputs. The overhead imposed by our binary instrumentation made runtimes for the reference inputs and the resultant trace files intractable. We excluded just one SPEC benchmark, 256.bzip2, because we could not process even its test inputs.

We describe all of the inputs we used to drive our benchmarks in

Table 1 except for Apache. To drive Apache, we follow Gay and Aiken and run on the same computer a program that fetches a large number of static web pages. While this test is unrealistic, it serves two purposes. First, it isolates performance from the usual network and disk I/O bottlenecks, magnifying the performance impact of custom allocation. Second, using the same benchmark as Gay and Aiken facilitates comparison with their work.

3.1 Emulating Custom Semantics

Custom memory allocators often support semantics that differ from the C memory allocation interface. In order to replace these custom allocators with `malloc` and `free`, we must emulate their semantics on top of the standard allocation calls. We wrote and tuned a region emulator to provide the full range of region semantics used by our benchmark applications, including nesting and obstacles (see Section 4.2). The region emulator uses the general-purpose allocator for each allocated object, but records a pointer for each object so that when the application deletes a region, the region emulator can call `free` on each allocated object. We record the pointer information for allocated objects in an out-of-band dynamic array associated with each region, rather than within the allocated objects. This method ensures that the last access to any allocated object is by the client program and not by our region emulator. Using this technique means that our region emulator has no impact on object drag, the elapsed time between last use and reclamation of an object, which we measure in Section 7.3. However, region emulation has an impact on space. Every allocated object requires 4 bytes of memory (for its record in the dynamic array) in addition to per-object overhead (4–8 bytes). Eliminating this overhead is an advantage of regions, but the inability to free individual objects may have a much greater impact on space, which we explore in Section 6.1.

4. Custom Allocators

In this section, we explain exactly what we mean by custom memory allocators. We discuss the reasons why programmers use them and survey a wide range of custom memory allocators, describing briefly what they do and how they work.

We use the term custom memory allocation in a proscribed way to denote any memory allocation mechanism that differs from general-purpose allocation in at least one of two ways. First, a custom allocator may provide more than one object for every allocated chunk of memory. Second, it may not immediately return objects to the system or to the general-purpose allocator. For instance, a custom allocator may obtain large chunks of memory from the general-purpose allocator which it carves up into a number of objects. A custom allocator might also defer object deallocation, returning objects to the system long after the object is last used or becomes unreachable.

Our definition of custom memory allocators excludes wrappers that perform certain tests (e.g., for null return values) before returning objects obtained from the general-purpose memory manager. We also exclude from consideration memory allocators that serve primarily as infrastructures for implementing object layout optimizations [9, 40].

4.1 Why Programmers Use Custom Allocators

There are a variety of reasons why programmers use custom memory allocators. The principal reason cited by programmers and authors of books on programming is runtime performance [7, 20, 24, 25, 26, 37]. Because the per-operation cost of most general-purpose memory allocators is an order of magnitude higher than that of custom allocators, programs that make intensive use of the

(a) Time spent in memory management operations for eight custom allocation benchmarks, with their allocators replaced by the Windows allocator (see Section 3.1). Memory management operations account for up to 40% of program runtime (on average, 16%), indicating a substantial opportunity for optimization.

(b) Memory consumption for eight custom allocation benchmarks, including *only memory allocated by the custom allocators*. Most of these consume relatively small amounts of memory on modern hardware, suggesting little opportunity for reducing memory consumption.

Figure 1: Runtime and space consumption for eight custom allocation benchmarks.

allocator may see performance improvements by using custom allocators.

Improving performance

Figure 1(a) shows the amount of time spent in memory management operations on eight applications using a wide range of custom memory allocators, with the custom memory allocator replaced by the Windows allocator. We use the region emulator from Section 3.1 for 176.gcc, Apache, lcc, and mudlle. Many of these applications spend a large percentage of their runtime in the memory allocator (16% on average), demonstrating an opportunity to improve performance by optimizing memory allocation.

Nearly all of our benchmarks use custom allocators to improve performance. This goal is often explicitly stated in the documentation or source code. For instance, the Apache API (application-programmer interface) documentation claims that its custom allocator `ap_palloc` “is generally faster than malloc.” The STLport implementation of STL (used in our runs of C-Breeze) refers to its custom allocator as an “optimized node allocator engine,” while 197.parser’s allocator is described as working “best for ‘stack-like’ operations.” Allocation with obstacks (used by 176.gcc) “is usually very fast as long as the objects are usually small”² and mudlle’s region-based allocator is “fast and easy.” Because Hanson cites performance benefits for regions in his book [20], we assume that they intended the same benefit. Lcc also includes a per-class custom allocator, intended to improve performance, which had no observable performance impact.³ The per-class freelist-based custom allocator for boxed-sim also appears intended to improve performance.

Reducing memory consumption

While programmers primarily use custom allocators to improve performance, they also occasionally use them to reduce memory

consumption. One of our benchmarks, 175.vpr, uses custom allocation exclusively to reduce memory consumption, stating that its custom allocator “should be used for allocating fairly small data structures where memory-efficiency is crucial.”⁴ The use of obstacks in 176.gcc might also be partially motivated by space considerations. While the source documentation is silent on the subject, the documentation for obstacks in the GNU C library suggests it as a benefit.⁵ Figure 1(b) shows the amount of memory consumed by custom allocators in our benchmark applications. Only 197.parser and 176.gcc consume significant amounts of memory on modern hardware (30MB and 91MB, respectively). However, recall that we use small input sizes in order to be able to process the trace files.

Improving software engineering

Writing custom code to replace the general-purpose allocator is generally not a good software engineering practice. Memory allocated via a custom allocator cannot be managed by another allocator, including the general-purpose memory manager. Inadvertently calling `free` on a custom-allocated object can corrupt the heap and lead to a segmentation violation. The result is a significant bookkeeping burden on the programmer to ensure that objects are freed by the correct allocator. Custom allocators also can make it difficult to understand the sources of memory consumption in a program. Using custom allocators often precludes the use of memory leak detection tools like Purify [21]. Use of custom allocators also precludes the option of later substituting a parallel allocator to provide SMP scalability [3], a garbage collector to protect against memory leaks [29], or a shared multi-language heap [42].

However, custom memory allocators can provide some important software engineering benefits. The use of region-based custom allocators in parsers and compilers (e.g., 176.gcc, lcc, and mudlle) simplifies memory management [20]. Regions provide separate memory areas that a single call deletes in their entirety.

²From the documentation on obstacks in the GNU C library.

³Hanson, in a private communication, indicated that the only intent of the per-class allocator was performance. In the results presented here, we disabled this custom allocator to isolate the impact of its region-based allocators.

⁴See the comment for `my_chunk_malloc` in `util.c`.

⁵“And the only space overhead per object is the padding needed to start each object on a suitable boundary.”

Benchmark	Allocator	Motivation			Policy				Mechanism		
		performance	space	software engineering	same API	region-Delete	nested lifetimes	multiple areas	chunks	stack optimized	same-type optimized
197.parser	custom pattern	■			■				■		
boxed-sim	per-class	■			■			■			■
C-Breeze	per-class (STL)	■			■			■			■
175.vpr	region		■			■		■	■		
176.gcc	obstack region	■	■	■		■		■	■	■	
apache	nested region	■		■		■	■	■	■		
lcc	region	■		■		■		■	■		
mudlle	region	■		■		■		■	■		
reaps		■		■	■	■	■	■	■		

Table 2: Characteristics of the custom allocators in our benchmarks, including reaps. Performance motivates all but one of the custom allocators, while only two were (possibly) motivated by space concerns (see Section 4.1). “Same API” means that the allocator allows individual object allocation and deallocation, and “chunks” means the custom allocator obtains large blocks of memory from the general-purpose allocator for its own use (see Section 4.2).

Multithreaded server applications use regions to isolate the memory spaces of separate threads (*sandboxing*), reducing the likelihood that one thread will accidentally or maliciously overwrite another thread’s data. Server applications like the Apache web server also use regions to prevent memory leaks. When a connection is terminated or fails, the server tears down all memory associated with the connection simply by freeing the associated region. However, regions do not allow individual object deletion, so an entire region must be retained as long as just one object within it remains live. This policy can lead to excessive memory consumption and prevents the use of regions for certain usage patterns, as we explore in Section 4.3.

4.2 A Taxonomy of Custom Allocators

In order to outperform the general-purpose memory allocator, programmers apply knowledge they have about some set of objects. For instance, programmers use regions to manage objects that are known to be dead at the same time. Programmers also write custom allocators to take advantage of object sizes or other allocation patterns.

We break down the allocators from our custom allocation benchmarks in terms of several characteristics in Table 2. We divide these into three categories: the *motivation* behind the programmer’s use of a custom allocator, the *policies* implemented by the allocators, and the *mechanisms* used to implement these policies. Notice that in all but one case (175.vpr), performance was a motivating factor. We explain the meaning of each characteristic in the descriptions of the custom allocators below.

Per-class allocators. Per-class allocators optimize for allocation of the same type (or size) of object by eliding size checks and keeping a freelist with objects only of the specific type. They implement the same API as `malloc` and `free`, i.e., they provide individual object allocation and deletion, but optimize only for one size or type.

Regions. Regions allocate objects by incrementing a pointer into large chunks of memory. Programmers can only delete regions in their entirety. Allocation and freeing are thus as fast as possible. A region allocator includes a `freeAll` function that deletes all memory in one operation and includes support for multiple allocation areas that may be managed independently. Regions reduce bookkeeping burden on the programmer and reduce memory leaks, but do not allow individual objects to be deleted.

Figure 2: An example of region-based memory allocation. Regions allocate memory by incrementing a pointer into successive chunks of memory. Region deletion reclaims all allocated objects *en masse* by freeing these chunks.

Two of the custom allocators in this survey are variants of regions: nested regions and obstacks. *Nested regions* support nested object lifetimes. Apache uses these to provide regions on a per-connection basis, with sub-regions for execution of user-provided code. Tearing down all memory associated with a connection requires just one `regionDelete` call on the memory region.

An *obstack* is an extended version of a region allocator that adds deletion of every object allocated after a certain object [43]. This extension supports object allocation that follows a stack discipline (hence the name, which comes from “object stack”).

Custom patterns. This catch-all category refers to what is essentially a general-purpose memory allocator optimized for a particular pattern of object behavior. For instance, 197.parser uses a fixed-size region of memory (in this case, 30MB) and allocates after the last block that is still in use by bumping a pointer. Freeing a block marks it as free, and if it is the last block, the allocator resets the pointer back to the new last block in use. This allocator is fast for 197.parser’s stack-like use of memory, but if object lifetimes do not follow a stack-like discipline, it exhibits unbounded memory consumption.

4.3 Problems with Regions

As the above description shows, many custom allocators are based

(a) A lattice of APIs, showing how reaps combine the semantics of regions and heaps.

(b) A diagram of the heap layers that comprise our implementation of reaps. Reaps adapt to their use, acting either like regions or heaps (see Section 5). The CoalesceableHeap layer adds per-object metadata that enable a heap to subsequently manage memory obtained from a region.

Figure 3: A description of the API and implementation of reaps.

on regions. Regions can have both performance and software engineering advantages over general-purpose memory allocation, but can considerably increase memory consumption. More importantly, regions cannot be used for many allocation patterns. In particular, regions cannot be used when implementing unbounded buffers, dynamic arrays, or producer-consumer patterns. Because programmers cannot reclaim individual objects within regions, programs using any of these allocation patterns would consume unbounded amounts of memory. These limitations are a practical problem. For instance, the Apache API manages memory with regions (“pools”) to prevent resource leaks. Programmers add functionality to Apache by writing *modules* compiled into the Apache server. Regions constrain the way programmers write modules and prevent them from using natural allocation patterns like producer-consumer. In general, programmers must rewrite applications that were written using general-purpose allocation. This restriction is an unintended consequence of the use of regions to satisfy Apache’s need for heap teardown and high performance.

Ideally, we would like to combine general-purpose allocation with region semantics, allowing for multiple allocation areas that can be cheaply deleted en masse. This extension of region semantics with individual object deletion would satisfy the needs of applications like Apache while increasing their allocation pattern coverage. This interface comprises all of the semantics provided by the custom allocators we surveyed, excluding obstack deletion. A high-performance implementation would reduce the need for conventional regions and many other custom allocators. These are the goals of the allocator that we describe in the next section.

5. Reaps: Generalizing Regions and Heaps

We have designed and implemented a generalization of regions and general-purpose memory allocators (heaps) that we call *reaps*. Reaps provide a full range of region semantics, including nested regions, but also include individual object deletion. Figure 3(a) depicts a lattice of API’s, showing how reaps combine the semantics of regions and heaps. We provide a C-based interface to reap

Figure 4: An example of reap allocation and deallocation. Reaps add metadata to objects so that they can be freed onto a heap.

allocation, including operations for reap creation and destruction, clearing (freeing of every object in a reap without destroying the reap data structure), and individual object allocation and deallocation:

```
void reapCreate (void ** reap, void ** parent);
void reapDestroy (void ** reap);
void reapFreeAll (void ** reap); // clear
void * reapMalloc (void ** reap, size_t size);
void reapFree (void ** reap, void * object);
```

We built our implementation of reaps using Heap Layers [4]. The Heap Layers infrastructure allows programmers to compose allo-

cator “layers” to build high-performance memory allocators much more easily than by modifying the code of an existing memory allocator. These layers are C++ implementations of *mixins* [6], using classes whose superclass is a template argument. With mixins, the programmer creates allocators from composable layers that a compiler implements efficiently.

5.1 Design and Implementation

Our implementation of reaps includes both a region-like allocator and support for nested reaps. Reaps adapt to their use, behaving either like regions or like heaps. Initially, reaps behave like regions. They allocate memory by bumping a pointer through geometrically-increasing large chunks of memory (initially 8K), which they thread onto a doubly-linked list. Unlike regions, however, reaps add object headers to every allocated object. These headers (“boundary tags”) contain metadata that allow the object to be subsequently managed by a heap. Reaps act in this region mode until a call to `reapFree` deletes an individual object. Reaps place freed objects onto an associated heap. Subsequent allocations from that reap use memory from the heap until it is exhausted, at which point it reverts to region mode. An example of reap allocation appears in Figure 4.

Figure 3(b) depicts the design of reaps in graphical form, using Heap Layers. Memory requests (`malloc` and `free`) come in from below and proceed upwards through the class hierarchy. We adapt `LeaHeap`, a heap layer that approximates the behavior of the `Lea` allocator, in order to take advantage of its high speed and low fragmentation. In addition, we wrote three new layers: `NestedHeap`, `ClearOptimizedHeap`, and `RegionHeap`.

The first layer, `NestedHeap`, provides support for nesting of heaps. The second layer, `ClearOptimizedHeap`, optimizes for the case when no memory has yet been freed by allocating memory very quickly by bumping a pointer and adding necessary metadata. `ClearOptimizedHeap` takes two heaps as arguments and maintains a boolean flag, `nothingOnHeap`, which is initially true. While this flag is true, `ClearOptimizedHeap` allocates memory from its first heap, bumping a pointer and adding per-object metadata as a side effect of allocating through `CoalesceableHeap`. We require this header information so that we can subsequently free this memory onto a heap. Bypassing the `LeaHeap` for this case has little impact on general-purpose memory allocation, speeding up only the initial allocation of heap items, but it dramatically improves the performance of region allocation. When an object is freed, it is placed on the heap and the `nothingOnHeap` flag is set to false. `ClearOptimizedHeap` then allocates memory from its second heap. When the heap is empty, or when the region is deleted, the `nothingOnHeap` flag is reset to true.

The last layer, `RegionHeap`, maintains a linked list of allocated objects and provides a region deletion operation (`clear()`) that iterates through this list and frees the objects. We use the `RegionHeap` layer to manage memory in geometrically-increasing chunks of at least 8K, making `reapFreeAll` efficient.

6. Evaluating Custom Memory Allocators

We provide allocation statistics for our benchmarks in Table 3. Many of the general-purpose allocation benchmarks are not allocation-intensive, but we include them for completeness. In particular, 181.mcf, 186.crafty, 252.eon and 254.gap allocate only a few objects over their entire lifetime, including one or more very large objects. Certain trends appear from the data. In general, programs using general-purpose allocators spend relatively little time in the memory allocator (on average, around 3%), while programs using custom allocators spend on average 16% of their time in memory operations. Programs that use custom allocators also tend to allo-

cate many small objects. This kind of allocation behavior stresses the memory allocator, and demonstrates that programmers using custom memory allocators were generally correct in pinpointing the memory manager as a significant factor in the performance of their applications.

6.1 Evaluating Regions

While we have identified four custom memory management policies (same API, regions, nesting, and multiple areas), regions are unique in requiring the programmer to tailor their program to their choice of allocation policy.⁶ By using regions, programmers give up the ability to delete individual objects. When all objects in a region die at the same time, this restriction does not affect memory consumption. However, the presence of just one live object ties down an entire region, potentially leading to a considerable amount of wasted memory. We explore the impact on memory consumption of this inability to reclaim dead objects in Section 7.3.

We measure the impact of using regions by using a binary instrumentation tool we wrote using the Vulcan binary instrumentation system [38]. We link each program with our region emulator and instrument them using our tool to track both allocations and accesses to every heap object. When an object is actually deleted (explicitly by a `free` or by a region deletion), the tool outputs a record indicating when the object was last touched, in allocation time. We post-process the trace to compute the amount of memory the program would use if it had freed each individual object as soon as possible. This highly-aggressive freeing may be reasonable, as we show below with measurements of programs using general-purpose memory allocators.

7. Results

This section compares execution times of different memory allocation policies on a number of programs; it compares the original custom allocators with general purpose allocation and reaps. We find that only region allocators consistently outperform the `Lea` general-purpose allocator. We also compare the space consumption of these options, and find that regions may unnecessarily consume additional space. A few other systems provide similar semantics to reaps, and we show that our implementation of reaps is faster. Finally, we present a case study that shows that by combining object deletion and region semantics, reaps can yield reduced memory consumption while maintaining the software engineering and protection benefits of regions.

All runtimes are the best of three runs at real-time priority after one warm-up run; variation was less than one percent. All programs were compiled with Visual C++ 6.0 and run on a 600 MHz Pentium III system with 320MB of RAM, a unified 256K L2 cache, and 16K L1 data and instruction caches, under Windows XP. We compare the custom allocators to the Windows XP memory allocator, which we refer to in the graphs as “Win32”, to version 2.7.0 of Doug Lea’s allocator, which we refer to as “DLmalloc”, and to our implementation of reaps. For the non-region applications and 176.gcc, we use reaps as a substitute for `malloc` and `free` (with region emulation for 176.gcc). For the remaining benchmarks, we use reaps as a direct replacement for regions.

7.1 Runtime Performance

To compare runtime performance of custom allocation to general-purpose allocation, we simply reroute custom allocator calls to the general-purpose allocator or reaps, using region emulation when needed. For this study, we compare custom allocators to the Win-

⁶Nesting also requires a different programming style, but in our experience, nesting only occurs in conjunction with regions.

Benchmark Statistics							
Benchmark	Total objects	Max objects in use	Average object size (in bytes)	Total memory (in bytes)	Max in use (in bytes)	Total/max	Mem mgmt. ops. (% of runtime)
custom allocation							
<i>197.parser</i>	9,334,022	230,919	38	351,772,626	3,207,529	109.7	41.8%
<i>boxed-sim</i>	52,203	4,865	15	777,913	301,987	2.6	0.2%
<i>C-Breeze</i>	5,090,805	2,177,173	23	118,996,917	60,053,789	1.9	17.4%
<i>175.vpr</i>	3,897	3,813	44	172,967	124,636	1.4	0.1%
<i>176.gcc</i>	9,065,285	2,538,005	54	487,711,209	112,753,774	4.3	6.7%
<i>apache</i>	149,275	3,749	208	30,999,123	754,492	41	0.1%
<i>lcc</i>	1,465,416	92,696	57	83,217,416	3,875,780	21.5	24.2%
<i>mudlle</i>	1,687,079	38,645	29	48,699,895	662,964	73.5	33.7%
general-purpose allocation							
<i>espresso</i>	4,483,621	4,885	249	1,116,708,854	373,348	2991.1	10.8%
<i>lindsay</i>	108,861	297	64	6,981,030	1,509,088	4.6	2.8%
<i>164.gzip</i>	1,307	72	6108	7,983,304	6,615,288	1.2	0.1%
<i>181.mcf</i>	54	52	1,789,028	96,607,514	96,601,049	1.0	1.5%
<i>186.crafty</i>	87	86	10,206	887,944	885,520	1.0	0.0%
<i>252.eon</i>	1,647	803	31	51,563	33,200	1.6	0.4%
<i>253.perlbmk</i>	8,888,870	5,813	16	144,514,214	284,029	508.8	12.6%
<i>254.gap</i>	50	48	1,343,614	67,180,715	67,113,782	1.0	0.0%
<i>255.vortex</i>	186,483	53,087	357	66,617,881	17,784,239	3.7	1.9%
<i>300.twolf</i>	9,458	1,725	56	532,177	66,891	8.0	0.9%

Table 3: Statistics for our benchmarks, replacing custom memory allocation by general-purpose allocation. We compute the runtime percentage of memory management operations with the default Windows allocator.

dows XP memory allocator, version 2.7.0 of the Lea allocator, and our implementation of reaps.

In Figure 5(a), the second bar shows that the Windows allocator degrades performance considerably for most programs. In particular, *197.parser* and *mudlle* run more than 60% slower when using the Windows allocator than when using the original custom allocator. Only *boxed-sim*, *175.vpr*, and *Apache* run less than 10% slower when using the Windows allocator. These results, taken on their own, would more than justify the use of custom allocators for most of these programs.

However, the picture changes when we look at the third bar, showing the results of replacing the custom allocators with the Lea allocator (DLmalloc). For six of the eight applications, the Lea allocator provides nearly the same performance as the original custom allocators (less than 2% slower on average). The Lea allocator actually slightly improved performance for *C-Breeze* when we turned off STL’s internal custom allocators. Only two of the benchmarks, *lcc* and *mudlle*, still run much faster with their region-based custom allocators than with the Lea allocator. This result shows that a state-of-the-art general-purpose allocator eliminates most of the performance advantages of custom allocators.

The fourth bar in Figure 5(a) shows the results for reaps. The results show that even when reaps are used for general-purpose allocation, which is not their intended role, they perform quite well, nearly matching the Lea allocator for all but *197.parser* and *C-Breeze*. However, for the two remaining benchmarks (*lcc* and *mudlle*), reaps nearly match the performance of the original custom allocators, running under 8% slower (as compared with the Lea allocator, which runs 21–47% slower). These results show that reaps achieve performance comparable to region-based allocators while providing the flexibility of individual object deletion.

7.2 Memory Consumption

We measure the memory consumed by the various memory allocators by running the benchmarks, with custom allocation, the Lea allocator and with reaps, all linked with a slightly modified version

of the Lea allocator. We modify the *sbrk* and *mmap* emulation routines to keep track of the high water mark of memory consumption. We do not include the Windows XP allocator in this study because it does not provide an equivalent way to keep track of memory consumption. In these experiments, the reap versions do not use individual object deletion but rather each reap acts as a region. Modifying all the programs to use deletion requires application-specific information and is very labor-intensive. Section 7.5 shows a case study for reaps that uses individual object deletion thus reducing the memory requirement for a new *Apache* module.

Figure 5(b) shows our results for memory consumption, which are quite mixed. Neither custom allocators, the Lea allocator, nor reaps consistently yield a space advantage. *176.gcc* allocates many small objects, so the per-object overhead of both reaps and the Lea allocator (8 bytes) leads to increased memory consumption. Despite their overhead, reaps and the Lea allocator often *reduce* memory consumption, as in *197.parser*, *C-Breeze* and *Apache*. The custom memory allocator in *197.parser* allocates from a fixed-sized chunk of memory (a compile-time constant, set at 30MB), while the Lea allocator and reaps use just 15% of this memory. Worse, this custom allocator is brittle; requests beyond the fixed limit result in program termination. *Apache*’s region allocator is less space-efficient than reaps or our region emulator, accounting for the difference in space consumption. On the other hand, our use of increasing chunk sizes in reaps causes increased memory consumption for *mudlle*.

Of the two allocators implicitly or explicitly intended to reduce memory consumption, *176.gcc*’s *obstacks* achieves its goal, saving 32% of memory compared to the Lea allocator, while *175.vpr*’s provides only an 8% savings. Custom allocation does not necessarily provide space advantages over the Lea allocator or reaps, which is consistent with our observation that programmers generally do not use custom allocation to reduce memory consumption.

Our results show that most custom allocators achieve neither performance nor space advantages. However, region-based allocators

(a) Normalized runtimes (smaller is better). Custom allocators often outperform the Windows allocator, but the Lea allocator is as fast as or faster than most of the custom allocators. For the region-based benchmarks, reaps come close to matching the performance of the custom allocators.

(b) Normalized space (smaller is better). We omit the Windows allocator because we cannot directly measure its space consumption. Custom allocators provide little space benefit and occasionally consume much more memory than either general-purpose allocators or reaps.

Figure 5: Normalized runtime and memory consumption for our custom allocation benchmarks, comparing the original allocators to the Windows and Lea allocators and reaps.

Figure 6: Normalized runtimes (smaller is better). Reaps are almost as fast as the original custom allocators and much faster than previous allocators with similar semantics.

can provide both advantages (see `lcc` and `mudlle`). These space advantages are somewhat misleading. While the Lea allocator and reaps add a fixed overhead to each object, regions can tie down arbitrarily large amounts of memory because programmers must wait until all objects are dead to free their region. In the next section, we measure this hidden space cost of using the region interface.

7.3 Evaluating Region Allocation

Using the binary instrumentation tool we describe in Section 6.1, we obtained two curves over allocation time [22] for each of our benchmarks: memory consumed by the region allocator, and memory required when dead objects are freed immediately after their last access. Dividing the areas under these curves gives us *total drag*, a measure of the average ratio of heap sizes with and without

immediate object deallocation. A program that immediately frees every dead object thus has the minimum possible total drag of 1. Intuitively, the higher the drag, the further the program’s memory consumption is from ideal.

Figure 7(a) shows drag statistics for a wide range of benchmarks, including programs using general-purpose memory allocators. Programs using non-region custom allocators have minimal drag, as do the bulk of the programs using general-purpose allocation, indicating that programmers tend to be aggressive about reclaiming memory. The drag results for `255.vortex` show either that some programmers are not so careful, or that some programming practices may preclude aggressive reclamation. The programs with regions consistently exhibit more drag, including `176.gcc` (1.16), and `mudlle` (1.23), and `lcc` has very high drag (3.34). This drag corresponds to an average of three times more memory consumed than required.

In many cases, programmers are more concerned with the peak memory (footprint) consumed by an application rather than the average amount of memory over time. Table 4 shows the footprint when using regions compared to immediately freeing objects after their last reference. The increase in peak caused by using regions ranges from 6% for `175.vpr` to 63% for `lcc`, for an average of 23%. Figure 7(b) shows the memory requirement profile for `lcc`, demonstrating how regions influence memory consumption over time. These measurements confirm the hypothesis that regions can lead to substantially increased memory consumption. While programmers may be willing to give up this additional space in exchange for programming convenience, we believe that they should not be forced to do so.

7.4 Experimental Comparison to Previous Work

In Figure 6, we present results comparing the previous allocators that provide semantics similar to those provided by reaps (see Section 2). Windows Heaps are a Windows-specific interface providing multiple (but non-nested) heaps, and `Vmalloc` is a custom allocation infrastructure that provides the same functionality. We present results for `lcc` and `mudlle`, which are the most allocation intensive of our region benchmarks. Using Windows Heaps in

(a) Drag statistics for applications using general-purpose memory allocation (average 1.1), non-regions (average 1.0) and region custom allocators (average 1.6, 1.1 excluding lcc).

(b) Memory requirement profile for lcc. The top curve shows memory required when using regions, while the bottom curve shows memory required when individual objects are freed immediately.

Figure 7: The effect on memory consumption of not immediately freeing objects. Programs that use region allocators are especially draggy. Lcc in particular consumes up to 3 times as much memory over time as required and 63% more at peak.

Peak memory			
Benchmark	With regions	Immediate free	% Increase
175.vpr	131,274	123,823	6%
176.gcc	67,117,548	56,944,950	18%
apache	564,440	527,770	7%
lcc	4,717,603	2,886,903	63%
mudlle	662,964	551,060	20%

Table 4: Peak memory (footprint) for region-based applications, in bytes. Using regions leads to an increase in footprint from 6% to 63% (average 23%).

place of regions makes lcc take twice as long, and makes mudlle take almost 68% longer to run. Using Vmalloc slows execution for lcc by four times and slows mudlle by 43%. However, reaps slow execution by just under 8%, showing that reaps are the best implementation of this functionality of which we are aware.

For six of our eight benchmarks, replacing the custom allocator with the Lea allocator yields comparable performance. Using reaps imposes a runtime penalty from 0% to 8% compared to the original region-based allocators. In addition, reaps provide a more flexible interface than regions that permits programmers to reclaim unused memory. We believe that, for most applications, the greater flexibility of reaps justifies their small overhead.

7.5 Reaps in Apache

As a case study, we built a new Apache module to demonstrate the space consumption advantages provided by allowing individual object deletion within a region allocation framework. Using Apache’s module API [36], we incorporated bc, an arbitrary-precision calculator language [27] that uses malloc/free. Apache implements its own pool (region) API, including pool allocation, creation, and destruction. We reroute these calls to use reap (reapMalloc, reapCreate, and reapDestroy) and add a ap_pfree call routed to reapFree, thus enabling Apache modules to utilize the full range of reap functionality. In this way, all existing Apache modules use reap, but naturally do not yet take advantage of individual object deletion.

Using preprocessor directives, we redefined the calls to malloc and free in bc to ap_palloc and ap_pfree. This modification affected just 20 lines out of 8,000 lines total in bc. We then incorporated bc into a module called mod_bc. Using this module, clients can execute bc programs directly within Apache, while benefiting from the usual memory leak protection provided by pools. We then compared memory consumption with and without ap_pfree on a few test cases. For example, computing the 1000th prime consumes 7.4 megabytes of memory without ap_pfree. With ap_pfree, this calculation consumes only 240 kilobytes.

This experiment shows that we can have the best of both approaches. The reap functionality prevents memory leaks and offers module protection, as does the region interface currently in Apache, and furthermore, reaps enable a much richer range of application memory usage paradigms. Reaps make it practical to use standard malloc/free programs within Apache modules with only minor modifications.

8. Discussion

We have shown that performance frequently motivates the use of custom memory allocators and that they do not provide the performance they promise. Below we offer some explanations of why programmers used custom allocators to no effect.

Recommended practice

One reason that we believe programmers use custom allocators to improve performance is because it is recommended by so many influential practitioners and because of the perceived inadequacies of system-provided memory allocators. Examples of this use of allocators are the per-class allocators used by boxed-sim and lcc.

Premature optimization

During software development, programmers often discover that custom allocation outperforms general-purpose allocation in micro-benchmarks. Based on this observation, they may put custom allocators in place, but allocation may eventually account for a tiny percentage of application runtime.

Drift

In at least one case, we suspect that programmers initially made the *right* decision in choosing to use custom allocation for performance, but that their software evolved and the custom allocator no longer has a performance impact. The obstack allocator used by 176.gcc performs fast object reallocation, and we believe that this made a difference when parsing dominated runtime, but optimization passes now dominate 176.gcc's runtime.

Improved competition

Finally, the performance of general-purpose allocators has continued to improve over time. Both the Windows and Lea allocators are optimized for good performance for a number of programs and therefore work well for a wide range of allocation behaviors. For instance, these memory allocators perform quite well when there are many requests for objects of the same size, rendering per-class custom allocators superfluous (including those used by the Standard Template Library). While there certainly will be programs with unusual allocation patterns that might lead these allocators to perform poorly, we suspect that such programs are increasingly rare. We feel that programmers who find their system allocator to be inadequate should try using reaps or the Lea allocator rather than writing a custom allocator.

9. Conclusions

Despite the widespread belief that custom allocators improve performance, we come to a different conclusion. In this paper, we examine eight benchmarks using custom memory allocators, including the Apache web server and several applications from the SPECint2000 benchmark suite. We find that the Lea allocator is as fast as or even faster than most custom allocators. The exceptions are region-based allocators, which often outperform general-purpose allocation.

We show that regions can come at an increased cost in memory consumption and do not support common programming idioms. We present reaps, a generalization of general-purpose and region-based allocators. Reaps are a combination of regions and heaps, providing a full range of region semantics with the addition of individual object deletion. We demonstrate that reaps combine increased flexibility with high performance, outperforming other allocators with region-like semantics. We show that using reaps can yield significant reductions in memory consumption.

We plan to build on this work in several ways. Because reaps are currently limited to single-threaded use, we plan to integrate reaps with our Hoard scalable memory allocator [3]. We believe that such an extended and scalable memory allocator would eliminate the need for most custom memory allocators. We are also investigating the integration of reaps into a garbage-collected setting.

10. Acknowledgements

Thanks to Stephen Cheney for making the boxed-sim benchmark available to us, to both Sam Guyer and Daniel A. Jiménez for providing access to and guidance in using their C-Breeze compiler infrastructure, and to David Gay for making available his region benchmarks. Thanks also to Dave Hanson, Martin Hirzel, Doug Lea, Ken Pierce, and Ran Shaham for helpful discussions, and to Andy Edwards and the rest of the Vulcan team.

The software and benchmarks described in this paper may be downloaded from <http://www.cs.umass.edu/~emery>.

11. References

- [1] Apache Foundation. Apache Web server. <http://www.apache.org>.
- [2] William S. Beebe and Martin C. Rinard. An implementation of scoped memory for Real-Time Java. In *EMSOFT*, pages 289–305, 2001.
- [3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.
- [4] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124, Snowbird, Utah, June 2001.
- [5] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [6] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) / Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311, Ottawa, Canada, 1990. ACM Press.
- [7] Dov Bulka and David Mayhew. *Efficient C++*. Addison-Wesley, 2001.
- [8] Trishul Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 191–202, Snowbird, Utah, June 2001.
- [9] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Atlanta, GA, May 1999.
- [10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
- [11] Boris Fomitchev. STLport. <http://www.stlport.org/>.
- [12] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.
- [13] Free Software Foundation. GCC Home Page. <http://gcc.gnu.org/>.
- [14] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 313 – 323, Montreal, Canada, June 1998.
- [15] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 70 – 80, Snowbird, Utah, June 2001.
- [16] Wolfram Gloger. Dynamic memory allocator implementations in Linux system libraries. <http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html>.
- [17] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, Berlin, Germany, June 2002.

- [18] Sam Guyer, Daniel A. Jiménez, and Calvin Lin. The C-Breeze compiler infrastructure. Technical Report UTCS-TR01-43, The University of Texas at Austin, November 2001.
- [19] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. In *Software Practice & Experience*, number 20(1), pages 5–12. Wiley, January 1990.
- [20] David R. Hanson. *C Interfaces and Implementation*. Addison-Wesley, 1997.
- [21] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX 1992 Conference*, pages 125–136, December 1992.
- [22] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *International Symposium on Memory Management*, pages 26–36, Vancouver, B.C., Canada, 1998.
- [23] Doug Lea. A memory allocator. <http://g.oswego.edu/dl/html/malloc.html>.
- [24] Scott Meyers. *Effective C++*. Addison-Wesley, 1996.
- [25] Scott Meyers. *More Effective C++*. Addison-Wesley, 1997.
- [26] Bartosz Milewski. *C++ In Action: Industrial-Strength Programming Techniques*. Addison-Wesley, 2001.
- [27] Philip A. Nelson. bc - An arbitrary precision calculator language. <http://www.gnu.org/software/bc/bc.html>.
- [28] Jeffrey Richter. *Advanced Windows: the developer's guide to the Win32 API for Windows NT 3.5 and Windows 95*. Microsoft Press.
- [29] Gustavo Rodriguez-Rivera, Mike Spertus, and Charles Fiterman. Conservative garbage collection for general memory allocators. In *International Symposium on Memory Management*, Minneapolis, Minnesota, 2000.
- [30] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, 1967.
- [31] Colin Runciman and Niklas Rojemo. Lag, drag and postmortem heap profiling. In *Implementation of Functional Languages Workshop*, Bastad, Sweden, September 1995.
- [32] SGI. The Standard Template Library for C++: Allocators. <http://www.sgi.com/tech/stl/Allocators.html>.
- [33] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 104–113, Snowbird, Utah, June 2001.
- [34] Standard Performance Evaluation Corporation. SPEC2000. <http://www.spec.org>.
- [35] Standard Performance Evaluation Corporation. SPEC95. <http://www.spec.org>.
- [36] Lincoln Stein, Doug MacEachern, and Linda Mui. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, 1999.
- [37] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. (Addison-Wesley), 1991.
- [38] Suzanne Pierce. PPRC: Microsoft's Tool Box. <http://research.microsoft.com/research/pprc/mstoolbox.asp>.
- [39] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
- [40] Dan N. Truong, François Bodin, and André Sez nec. Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 322–329, October 1998.
- [41] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. In *Software Practice & Experience*, number 26, pages 1–18. Wiley, 1996.
- [42] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime approach to interoperability. In *Twelfth ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.
- [43] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 986, 1995.
- [44] Benjamin G. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23(7):733–756, 1993.