Reconsidering Custom Memory Allocation

Emery D. Berger
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
emery@cs.utexas.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

Many programmers use custom memory allocators hoping to achieve performance improvements. This first in-depth study examines eight applications that use custom allocators. Surprisingly, for six of these applications, a state-of-the-art general-purpose allocator performs as well as or better than the custom allocators. The two exceptions use regions, which deliver higher performance (up to 44% faster). Regions also reduce programmer burden and help eliminate a source of memory leaks. We show, however, that the inability of programmers to free individual objects within regions can lead to a substantial increase in memory consumption (up to 230% more).

To eliminate this excessive memory consumption, we develop the Reap extended general-purpose memory allocator, which combines region semantics with individual object deletion. Reap outperforms similar allocators and comes close to matching region performance while providing the potential for reduced memory consumption. We thus demonstrate an implementation of an extended memory allocation interface that eliminates the need for most custom memory allocators and the programmer effort needed to build and maintain them.

1. Introduction

Programmers seeking to improve performance often incorporate custom memory allocators into their applications. Custom allocators aim to take advantage of application-specific patterns of memory usage to manage memory more efficiently than a general-purpose memory allocator. For instance, 197.parser runs over 60% faster with its custom allocator than with the Windows XP allocator [2]. Numerous books and articles recommend custom allocators as an optimization technique [4, 20, 21]. The use of custom memory allocators is widespread, including the Apache web server [1], the GCC compiler [9], three of the SPECint2000 benchmarks [28] and the C++ Standard Template Library [7, 26], all of which we examine here.

The key contributions of this work are the following. We perform what we believe to be the first comprehensive evaluation of custom allocation. We survey a wide range of custom allocators and 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 [18]) yields equivalent performance for six of our eight benchmarks. The exceptions both use *region* custom allocators.

Regions provide high-performance but force the programmer to retain all memory associated with a region until the last object in the region dies [10, 11, 14, 24, 32]. We believe we are the first to show that the performance gains of regions (up to 44%) can come at the expense of excessive memory consumption (up to 230%). We measure the memory cost of regions using a binary instrumentation tool that determines the distance between the last reference to an object within a region and when the program frees the region.

We develop Reap, a high-performance extended general-purpose allocator that provides the performance and semantics of regions while allowing programmers to delete individual objects. The Reap allocator provides a reusable library solution for region allocation with competitive performance and the potential for reduced memory consumption, making conventional region allocators obsolete. We believe this general-purpose library eliminates the need for most programmers to write and maintain custom allocators.

The remainder of this paper is organized as follows. We discuss related work in Section 2. We describe our benchmarks in Section 3. Section 4 analyzes the structure of custom memory allocators used by our benchmark applications. We describe our experimental infrastructure and methodology in Section 5 and present experimental results in Section 6. Section 7 explores the cost in excess memory consumption of region-based custom allocators. We next describe the Reap extended general-purpose memory allocator in detail and present experimental results showing that Reap captures the performance of regions while allowing individual object deletion. We discuss our results in Section 9 and then conclude.

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 [4]. Meyers describes in detail the use of a freelist-based per-class custom allocator in "Effective C++" [19] and returns to the topic of custom allocators in the sequel [20]. Milewski also discusses per-class allocators as an optimization technique [21]. Hanson devotes a chapter to an im-

This work is supported by NSF ITR grant CCR-0085792, NSF grant ACI-9982028, NSF grant EIA-9726401, Darpa grant 5-21425, Darpa grant F33615-01-C-1892, and IBM. Part of this work was done while Emery Berger was at Microsoft Research. 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.
Submitted to PLDI 2002
Copyright 2002 ACM 0-89791-88-6/97/05 ..$5.00

plementation of regions ("arenas"), citing both the speed and software engineering benefits of regions as motivation [15]. Ellis and Stroustrup describe the syntactic facilities that allow overloading `operator new`, simplifying the use of custom allocators in C++ [6], and Stroustrup describes per-class allocators that use these facilities [30]. 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.

Region allocation, variously known as arenas, groups, and zones [14, 24] 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 [10, 11, 14, 24, 32]. Tofte and Talpin present a system that provides automatic region-based memory management for ML [32]. Aiken and Gay 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 [10, 11]. While these authors present only the benefits of regions, we investigate the hidden memory consumption cost of regions and present an alternative that avoids this cost by combining individual object deletion with the benefits of regions.

To compute the memory cost of region allocation, we measure the elapsed time between last use and reclamation of an object. This metric is known as "object drag". Our definition differs slightly from the original use of the term by Runciman and Rojemo [25], where 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 [27]. We use binary instrumentation to determine when objects are last referenced and post-process a combined allocation-reference trace to obtain object drag.

The literature on general-purpose memory allocators is extensive [34]. Here we describe the Windows XP and Lea allocators [18, 22], which we use in this study because of their widespread use (the Lea allocator forms the basis of the Linux memory allocator [12]). 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 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 large objects into smaller ones) and approximates best-fit. Larger 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 [17].

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 (not including nested regions) along with individual object deletion [23]. Vmalloc, a memory allocation infrastructure, also provides (non-nested) regions that permit individual object deletion [33]. We show in Section 7 that neither of these implementations match the performance of regions, while Reap adds the full range of region semantics and provides similar performance.

The only previous work evaluating the impact of custom memory allocators is by one of the authors (Zorn). In a paper on conservative garbage collection, Zorn compares custom ("domain-specific") allocators to general-purpose memory allocators [35]. He analyzed the performance of four benchmarks (cfrac, 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.

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 leads to a substantial increase in memory consumption. We develop a memory allocator that preserves the high performance of regions while providing individual object deletion to eliminate their excessive memory consumption.

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. Most of our benchmarks come from the SPECint2000 benchmark suite [28]. For the custom allocation benchmarks, we include a number of programs used in prior work on memory allocation. These programs include those used by Aiken and Gay (Apache, lcc, and mudlle) [10, 11], and boxed-sim, used by Chilimbi [5]. We also use the C-Breeze compiler infrastructure [13]. 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) [7, 26].

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, 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 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 Aiken and Gay 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 of custom allocation. Second, using the same benchmark as Aiken and Gay facilitates comparison with their work.

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

Benchmarks		
custom allocation		
175.vpr	FPGA placement & routing [28]	test placement
boxed-sim	Balls-in-box simulator [5]	*-n 3 -s 1*
197.parser	English parser [28]	*test.in*
c-breeze	C-to-C optimizing compiler [13]	*espresso.c*
lcc	Retargetable C compiler [8]	*scilab.i*
176.gcc	Optimizing C compiler [28]	*scilab.i*
apache	Web server [1]	see Section 5
mudlle	MUD compiler/interpreter [10]	*time.mud*
general-purpose allocation		
164.gzip	GNU zip data compressor [28]	*test/input.compressed 2*
181.mcf	Vehicle scheduler [28]	*test-input.in*
186.crafty	Chess program [28]	*test-input.in*
252.eon	Ray tracer [28]	*test/chair.control.cook*
253.perlbmk	Perl interpreter [28]	*perfect.pl b 3*
254.gap	Groups language interpreter [28]	*test.in*
255.vortex	Object-oriented DBM [28]	*test/lendian.raw*
300.twolf	CAD placement & routing [28]	*test.net*
espresso	Optimizer for PLAs [29]	*test2*
lindsay	Hypercube simulator [34]	*script.mine*

Table 1: Benchmarks used in this paper, with descriptions and inputs used. We include the general-purpose benchmarks for comparison with custom allocation in Section 7.

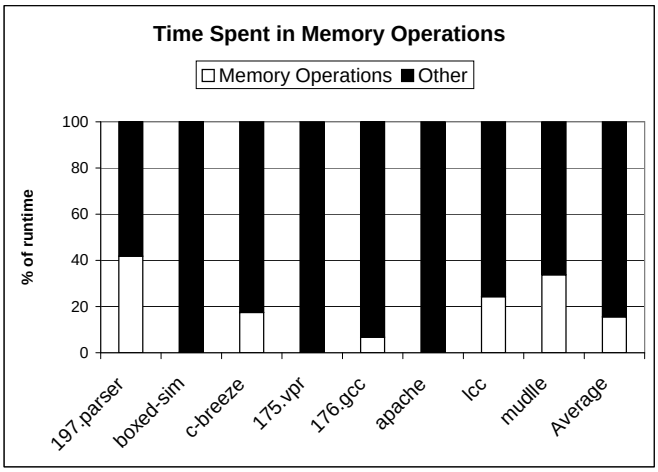


Figure 1: Time spent in memory operations for eight applications using custom memory allocators.

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.

While custom memory allocation could denote any mechanism different from direct use of the general-purpose allocator, we use the term in a more proscribed way to denote any memory allocation mechanism that differs from general-purpose allocation in at least one of two ways. A custom allocator provides more than one object for every object allocated from the system, or it does not immediately return objects to the system.[2] For instance, a custom allocator can 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 indefinitely.

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. Because the per-operation cost of most system general-purpose memory allocators is an order of magnitude higher than that of custom memory allocators, programs that make intensive use of the allocator may see performance improvements by using custom allocators.

Figure 1 shows the amount of time spent in memory operations on eight applications using a wide range of custom memory allocators, with the custom memory allocator replaced by the Windows allocator[3]. 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, among other considerations. This goal is often explicitly stated in the documentation or source code. For instance, the Apache API 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"[4] and mudlle's region-based allocator is "fast and easy". Because Hanson cites performance benefits for regions in his book, 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.[5] The per-class freelist-based custom allocator for boxed-sim also appears intended to improve performance. Only 175.vpr was explicitly optimized not for performance but for space (see Section 6.2).

While comments and documentation make it clear that programmers use custom allocators to improve performance, programmers generally do not appear to use them to reduce memory consumption. Only one of our benchmarks, 175.vpr, explicitly mentions this goal, stating that the custom allocator "should be used for allocating fairly small data structures where memory-efficiency is crucial."[6] Implicitly, the use of obstacks might be at least 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.[7] We are not aware of any discussion in the trade press of using custom allocators to improve memory efficiency.

While writing custom code to replace the general-purpose allocator is generally not a good software engineering practice, custom memory allocators can provide some software engineering benefits. The use of region-based custom allocators in parsers and compil-

[2] This definition of custom allocators excludes, *inter alia*, wrappers that perform certain tests (e.g., for null return values) before returning objects obtained from the general-purpose allocator.

[3] When needed, we use a *region emulator* that matches the semantics of the custom allocator (see Section 5.1).

[4] From the documentation on obstacks in the GNU C library.

[5] Hanson, in a private communication, indicated that the only intent of the per-class allocator was performance. We disabled this custom allocator to isolate the impact of region-based allocation.

[6] See the comment for `my_chunk_malloc` in `util.c`.

[7] "And the only space overhead per object is the padding needed to start each object on a suitable boundary. "

		Motivation			*Policy*				*Mechanism*		
Benchmark	Allocator	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	✓		✓		✓		✓	✓		

Table 2: Characteristics of the custom allocators in the benchmarks used in this paper. All but one of the custom allocators were motivated by performance concerns, while only two were (possibly) motivated by space concerns (see Sections 6.1 and 6.2). "Same API" means that the allocator allows only individual object allocation and deallocation, and "chunks" mean the custom allocator obtains large blocks of memory from the general-purpose allocator for its own use (see Section 4.2).

ers (e.g., 176.gcc, lcc, and mudlle) simplifies memory management [15]. Regions provide separate memory areas that can be disposed of with one operation, which is useful for server applications like the Apache web server. 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 potentially lead to excessive memory consumption, as we explore in Section 7.

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 can use regions to manage objects that all die at the same time. Programmers can also write custom allocators to take advantage of other object 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 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 (application-programmer interface) as `malloc` and `free`, i.e., they provide individual object allocation and deletion, but are optimized for only one type.

region Regions allocate objects by incrementing a pointer into large chunks of memory. Programmers can only delete regions in their entirety. A region allocator includes a `regionDelete` 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 aid in reducing memory leaks, but do not allow individual objects to be deleted.

obstack region An *obstack* is an extended version of a region allocator that adds deletion of every object allocated after a certain object [34]. This extension supports object allocation that follows a stack discipline (hence the name, which comes from "object stack").

nested region Nested regions are an extension of regions that 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 per-connection memory region.

custom pattern 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 can exhibit unbounded memory consumption.

As this discussion shows, programmers use a wide range of custom allocators, and in the next section, we evaluate whether they achieved their goals. In particular, we quantitatively analyze the performance of custom memory allocators in terms of runtime performance and space, and compare these to the Windows XP and Lea allocators.

5. 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 ones. 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 using custom allocators also tend to allocate many small objects. This kind of allocation behavior places considerable stress on the memory allocator.

5.1 Emulating Regions

Because custom allocators can support semantics that differ from the C memory allocation interface, we need to emulate regions with `malloc`/`free` as the underlying allocation mechanism. We wrote and tuned a region emulator that provides the full range of region semantics used by our benchmark applications, including nesting and obstacks. We record a pointer to every allocated object and when the region is deleted, we call `free` on each one. We record pointer information in a dynamic array associated with each region.

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	Memory operations (% of runtime)
custom allocation							
175.vpr	3,897	3,813	44	172,967	124,636	1.4	0.1%
boxed-sim	52,203	4,865	15	777,913	301,987	2.6	0.2%
197.parser	9,334,022	230,919	38	351,772,626	3,207,529	109.7	41.8%
c-breeze	5,090,805	2,177,173	23	118,996,917	60,053,789	1.9	17.4%
lcc	1,465,416	92,696	57	83,217,416	3,875,780	21.5	24.2%
176.gcc	9,065,285	2,538,005	54	487,711,209	112,753,774	4.3	6.7%
apache	149,275	3,749	208	30,999,123	754,492	41	0.1%
mudlle	1,687,079	38,645	29	48,699,895	662,964	73.5	33.7%
general-purpose allocation							
espresso	4,483,621	4,885	249	1,116,708,854	373,348	2991.1	10.8%
lindsay	108,861	297	64	6,981,030	1,509,088	4.6	2.8%
164.gzip	1,307	72	6108	7,983,304	6,615,288	1.2	0.1%
181.mcf	54	52	1,789,028	96,607,514	96,601,049	1.0	1.5%
186.crafty	87	86	10,206	887,944	885,520	1.0	0.0%
252.eon	1,647	803	31	51,563	33,200	1.6	0.4%
253.perlbmk	8,888,870	5,813	16	144,514,214	284,029	508.8	12.6%
254.gap	50	48	1,343,614	67,180,715	67,113,782	1.0	0.0%
255.vortex	186,483	53,087	357	66,617,881	17,784,239	3.7	1.9%
300.twolf	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 operations with the default Windows allocator. Programs using custom allocators spend on average 16% of their time in memory operations and use many small objects, while programs using general-purpose allocators spend relatively little time in memory operations (on average, 3%) and use few, large 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. 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 as we show in Section 7, the inability to free individual objects may have a much greater impact on space.

6. Results

In this section, we present our experimental results on runtime and memory consumption, discussing the programmers' goals for their custom allocators and whether they were met. All runtimes are the best of three runs after one warm-up run[8]; 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 build 2526. We compare the custom allocators to the Windows XP memory allocator, which we refer to in the graphs as "Win32", and to version 2.7.0 of Doug Lea's allocator, which we refer to as "DLmalloc".

6.1 Runtime Performance

To compare runtime performance of custom allocation to general-purpose allocation, we simply reroute custom allocator calls to the general-purpose allocator. For this study, we compare custom allocators to both the Windows XP memory allocator and to version 2.7.0 of the Lea allocator. Figure 3 shows that for four of the programs, the Windows allocator provides performance that comes within 10% of the original custom allocators: boxed-sim, 175.vpr, 176.gcc, and Apache (around 6% slower on average). Replacing the custom allocators with the Lea allocator (DLmalloc) provides nearly identical performance for six of the eight applications (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. This result shows that a good general-purpose allocator eliminates the performance advantages of custom allocators for most of our benchmarks.

However, the two remaining benchmarks (lcc and mudlle), do show runtime improvement when using their custom allocator, on average cutting runtime by 35%. Both lcc and mudlle use region-based allocators, which we examine in Section 7.

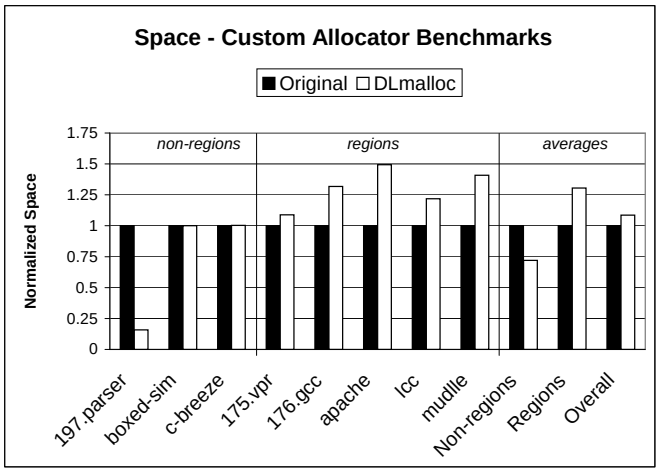
[8] We needed a longer warm-up period (five runs) for lcc.

6.2 Memory Consumption

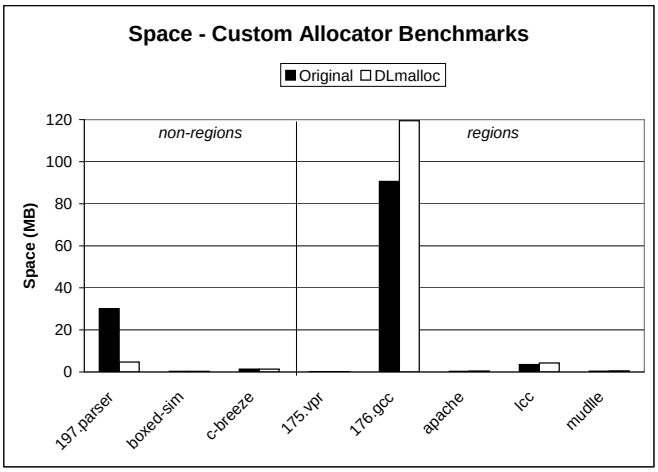
We also measured the memory consumed by the various memory allocators by running the benchmarks, with and without custom allocation, linked with a slightly modified version of the Lea allocator. We modified the `sbrk` and `mmap` emulation routines to keep track of the high water mark of memory consumption. We were unable to include the Windows XP allocator because it does not provide a way to keep track of memory consumption. While the measurements given in Table 3 include all memory allocated, in these results we include only memory consumed by the custom allocator or its replacement.

Regions complicate our measurement of memory consumption. We can simply directly compare the amount of memory consumed by the original region and by the general-purpose allocators combined with region emulation. However, we want to understand the impact of using the traditional region interface on memory consumption. By using regions, programmers give up the ability to delete individual objects. We explore the impact of this restriction on memory consumption.

We wrote a tool using the Vulcan binary instrumentation system [31] to track object drag, the time elapsed between the last access to an object and its reclamation [25, 27]. We link each program with our region emulator and instrument them to track both alloca-



(a) Normalized memory consumption.



(b) Actual memory consumption in megabytes.

Figure 2: Memory consumption for custom memory allocators compared to the Lea allocator with region emulation. Except for 197.parser, non-region custom allocators consume on average 3% less space than the Lea allocator. Region-based allocators consume on average 35% less memory. Only 197.parser and 176.gcc consume a significant amount of memory on modern hardware.

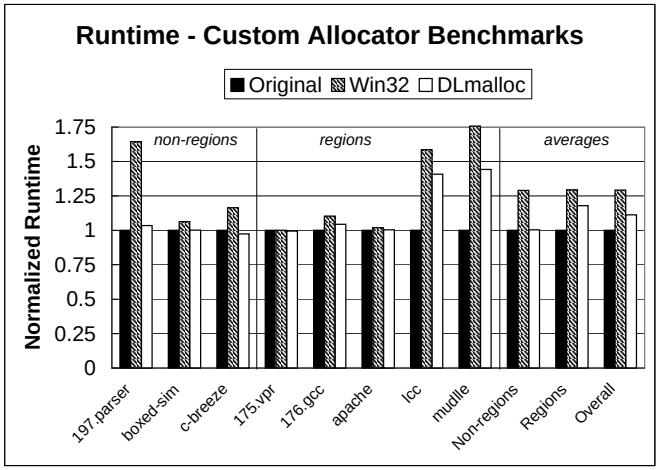


Figure 3: Normalized runtimes (smaller is better), replacing custom allocation with both the Windows and Lea allocators combined with region emulation. For six of the eight applications, the Lea allocator performs nearly as well as or better than the original custom allocators (less than 2% slower on average).

tions and accesses to every heap object. When an object is actually deleted (explicitly by a `free` or implicitly by a region deletion), the tool outputs when the object was last touched, in allocation time. We post-process the trace to compute object drag and thus compare the amount of memory that would have been used had the programmer freed each individual object as soon as possible. This highly-aggressive freeing is not always unrealistic, as we show in Section 7 with drag measurements of programs using general-purpose memory allocators.

Figure 2(a) shows our results for memory consumption. Excluding one outlier (197.parser), general-purpose allocators consume from 0% to 49% more memory than the custom allocators (on average 12%). Most of this increase arises in region-based allocators, where per-object overhead and region-emulation data structures combine to consume an average of 34% more memory. In one case, using general-purpose allocation dramatically *reduces* memory consumption. 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 achieves nearly the same performance with just 15% of the memory. Worse, this custom allocator is brittle; requests beyond the fixed limit result in program termination.

Region allocators show savings in memory consumption (from 8% to 49%) over general-purpose allocation with region emulation, while the other allocators only yield an average 3% savings, excluding 197.parser, whose custom allocator consumes 85% *more* memory. Of the two allocators implicitly or explicitly intended to reduce memory consumption, 176.gcc's obstacks achieves its goal, saving 32% of memory, while 175.vpr's provides only an 8% savings. While the performance improvements of lcc and mudlle could be attributed to improved locality due to avoiding per-object overhead, we show in Section 8.2 that these applications perform better simply because they allocate and delete faster.

Most of the space savings of region allocators is due to the overhead of region emulation plus general-purpose allocation. While these space savings can be substantial, it is worth noting that most of these benchmarks consume relatively little memory, although different inputs may increase memory consumption. For instance, the application whose custom allocator saves the most memory by percentage, Apache, only requires 450K of memory for its regions. Figure 2(b) shows the actual amount of space required by these benchmarks. In the context of most current hardware, only 197.parser and 176.gcc require large amounts of memory (30MB and 90MB, respectively), but still much less than typical RAM sizes of modern desktop PCs (256MB) and servers (1GB).

Our results show that while most custom allocators achieve neither performance nor space advantages, region-based allocators frequently provide both over general-purpose allocation combined with region emulation. These space advantages are misleading. While region emulation adds a fixed overhead to each object, regions can tie down arbitrarily large amounts of memory because program-

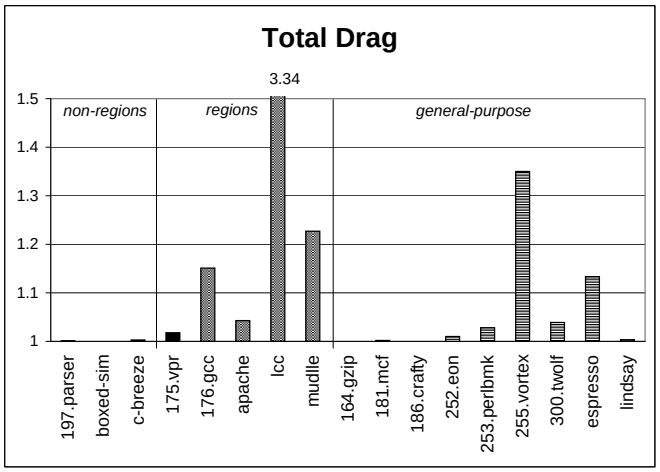


Figure 4: Drag statistics for applications using general-purpose memory allocation (average 1.1), non-regions (average 1.0) and region custom allocators (average 1.6, 1.1 excluding lcc).

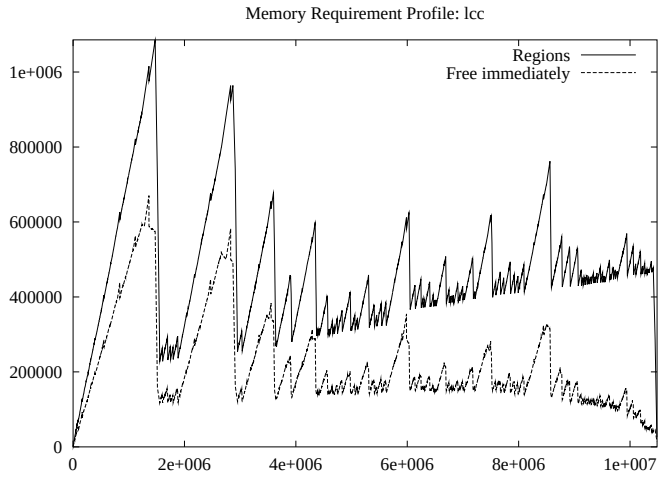


Figure 5: Memory requirement profile for lcc. The Y-axis is memory required, and the X-axis is allocation time. The top curve shows memory required when using regions, while the bottom curve shows memory required if individual objects are freed immediately.

mers 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. Evaluating Region Allocation

Regions achieve high performance by allocating memory by bumping a pointer through reasonably large chunks of memory, typically from 8K to 32K. Threading a linked list through these chunks permits region deletion of many small objects with a few operations. Regions also confer some software engineering benefits, because they simplify memory management and avoid memory leaks.

However, the region interface is severely constrained. A programmer can only free a region in its entirety and cannot free individual objects within a region [10, 11]. 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 calculate the effect of this restriction by comparing actual memory consumed to the amount that would have been required had the programmer immediately reclaimed dead objects. We could not measure this effect directly by modifying the source of our region-based benchmarks to use `malloc` and `free`, which would have required a complete rewrite of these applications. As we describe in Section 6.2, we collect allocation and reference traces using a binary instrumentation tool and subsequently post-process this trace. We obtain two curves over allocation time [16]: 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 program that immediately frees every dead object thus has the minimum possible total drag of 1.

Figure 4 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 that programmers are occasionally less aggressive. The programs with regions consistently exhibit more drag, including 176.gcc (1.16), and mudlle (1.23), and lcc has high drag (3.34). In other words, lcc's regions keep around 234% more memory than needed over time. The peak memory required with regions for lcc is approximately 1MB, while the peak when objects are immediately freed is 670K, an increase of 62%. Figure 5 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 excessive 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 in order to achieve high performance.

Region-based allocation clearly has both performance and software engineering advantages over general-purpose memory allocation, but can lead to considerable increases in memory consumption. Ideally, we would like to combine general-purpose allocation with region semantics, allowing for multiple allocation areas that can be cheaply deleted en masse. We would also like to extend region semantics with individual object deletion in order to address the problem of excessive memory consumption.

Both Vmalloc and the Windows "Heap" family of memory allocation functions come close to these mixed semantics. One key feature lacking in both is nested regions, which Apache requires. More importantly, however, neither implementation provides the performance we need, as we show in Section 8.2.

Providing a general-purpose allocation library with a wide range of region semantics, including nested heaps, would eliminate the need for programmers to roll their own region implementations. Extending such an interface to provide individual object deletion could yield significant memory savings. A high-performance implementation of such an allocator would make conventional regions obsolete. These are the goals of the extended general-purpose allocator that we describe in the next section.

8. The Reap Memory Allocator

We have designed and implemented a memory allocator which we call Reap (region + heap). Reap is an extended general-purpose memory allocator that provides a standard `malloc`/`free` interface. In addition, Reap adds the semantics of regions. Figure 6

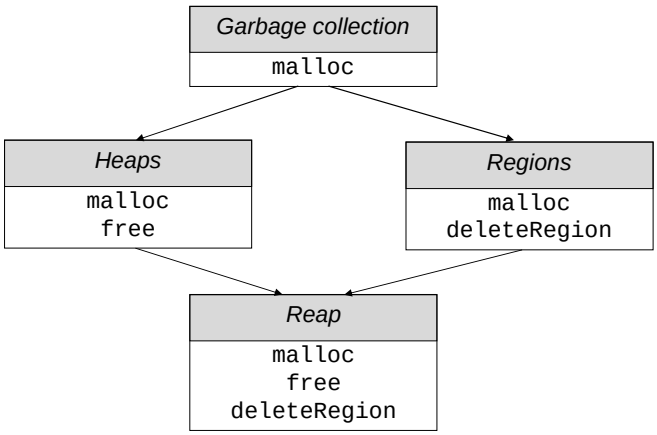


Figure 6: A lattice of APIs, showing how Reap combines the semantics of regions and heaps.

depicts a lattice of our interfaces. Reap provides a full range of region semantics, including nested regions, along with individual object deletion. We show that this allocator provides the extended semantics we want along with high performance. We provide a C-based interface to region allocation, including operations for region creation and destruction, deletion (freeing of every object in a region without destroying the region data structure) and individual object allocation and deallocation:

```
void regionCreate (void ** region, void ** parent);
void regionDelete (void ** region);
void regionDestroy (void ** region);
void * regionAllocate (void ** region, size_t size);
void regionFree (void ** region, void * object);
```

We implemented Reap using Heap Layers [2]. The Heap Layers infrastructure allows programmers to compose allocator "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* [3], using classes whose superclass is a template argument. With mixins, the programmer creates allocators from composable layers that a compiler implements efficiently.

8.1 Reap Design and Implementation

In order to add region semantics to a general-purpose allocator, each heap must manage its own memory and quickly discard memory. We adapt LeaHeap, a heap layer that approximates the behavior of the Lea allocator, in order to take advantage of LeaHeap's 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, helps us optimize for the case when no memory has yet been freed by allowing us to allocate memory by bumping a pointer. ClearOptimizedHeap takes two heaps as arguments and maintains a boolean flag, `noMemoryFreed`, which is initially true. While this flag is true, ClearOptimizedHeap allocates memory from its first heap argument. When an object is freed, `noMemoryFreed` is set to false. ClearOptimizedHeap then allocates memory from its second heap. Deleting a region resets the `noMemoryFreed` flag to true. In Reap, we use ClearOptimizedHeap to obtain memory directly from the system via CoalesceableHeap, which adds necessary header information so we can later free this memory (and coalesce adjacent free objects). Bypassing the LeaHeap for this case has no impact

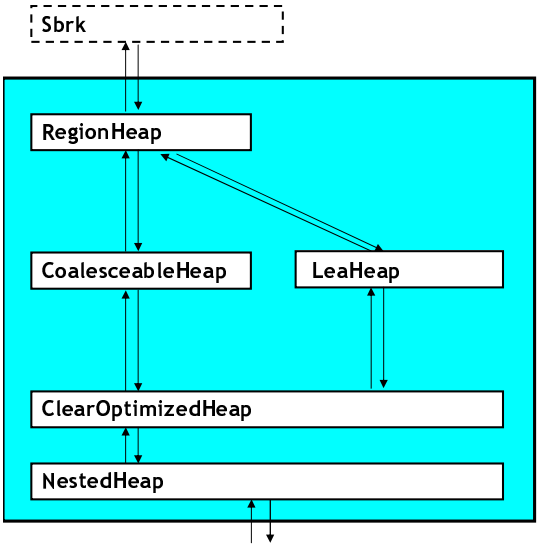


Figure 7: A diagram of the individual heap layers that comprise the Reap allocator. ClearOptimizedHeap bypasses the LeaHeap when no memory has yet been freed, dramatically improving the performance of region allocation.

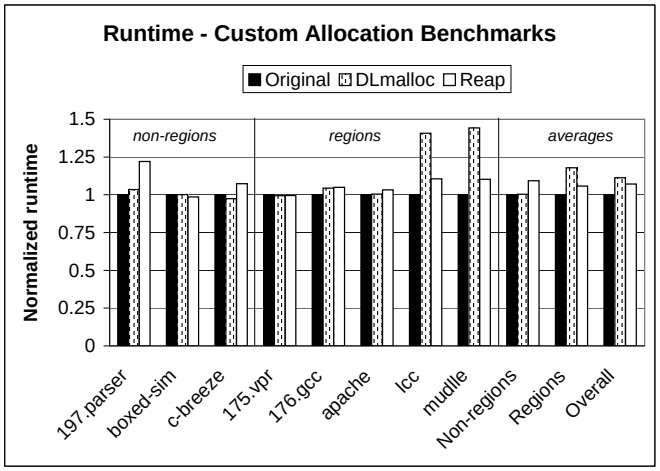
on general-purpose memory allocation, but dramatically improves the performance of region allocation.

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. In Reap, we use the RegionHeap layer to manage memory in chunks of at least 8K, making `regionDelete` efficient. In total, Reap required less than 150 lines of new code. Figure 7 depicts the design of Reap in graphical form.

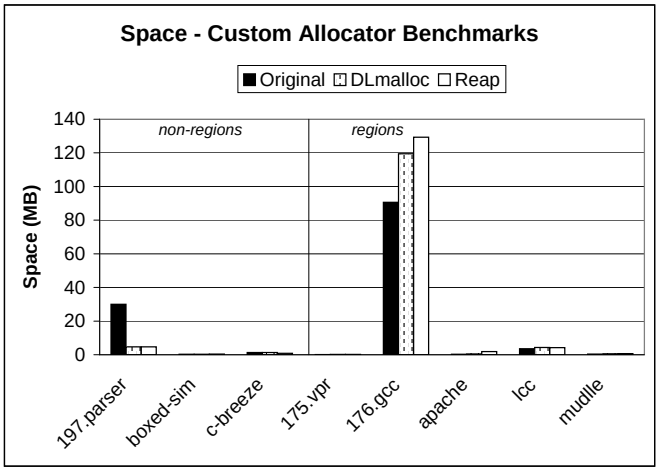
8.2 Experimental Results for Reap

In this section, we present experimental results for Reap. We compare Reap to the Windows XP and Lea allocators, combined with region emulation where appropriate. For the two applications where region-based allocation provided performance improvements, lcc and mudlle, we present additional results comparing Reap with two allocators that provide region semantics, Windows Heaps and Vmalloc (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.

Figure 9 shows our results for lcc and mudlle. Using Windows Heaps in place of regions makes lcc run 217% slower, and mudlle 68% slower. Using Vmalloc results in 407% slower execution for lcc and 43% slower execution for mudlle. The performance results for Windows Heaps and Vmalloc suggest that neither of them were designed with intensive region allocation and deletion in mind. Reap yields only 10% overhead for both lcc and mudlle, achieving performance comparable to the original custom allocators while providing the flexibility of individual object deletion. In Figure 8(a), we compare Reap to the original custom allocators and to the Lea allocator across all of our custom allocation benchmarks. Except for 197.parser, Reap comes within 10% of the performance of the original custom allocators. In Figure 8(b), we present our space results. 176.gcc allocates many small objects, so Reap's per-object overhead (8 bytes) leads to somewhat increased memory consumption compared to the Lea allocator (whose per-



(a) Normalized runtimes (smaller is better) for applications using custom memory allocators, replacing custom allocation with both the Lea allocator with region emulation and Reap.



(b) Actual space for applications using custom memory allocators, replacing custom allocation with both the Lea allocators and Reap.

Figure 8: Runtime and memory consumption for custom memory allocators compared to the Lea allocator with region emulation and Reap. Except for 197.parser, Reap performs within 10% of the original custom allocators.

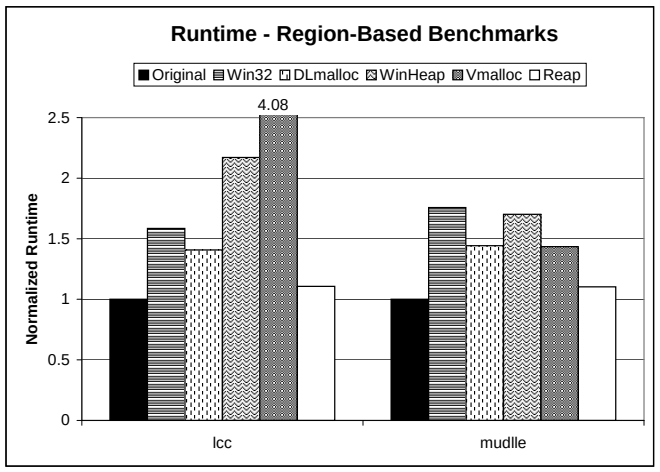


Figure 9: Normalized runtimes (smaller is better). Reap performs almost as quickly as the original custom allocators (within 10%), much faster than region emulation or other region-based allocators.

object overhead is 4 bytes). Reap is close in memory efficiency to the Lea allocator for most of the rest of the benchmarks.

We have shown that Reap is highly effective as a replacement for region-based custom allocators. For six of our eight benchmarks, replacing the custom allocator with the Lea allocator yields comparable performance. Reap provides similar performance for the other two. We see that using the Lea allocator for some benchmarks and Reap for others yields performance comparable to the original custom allocators. In addition, Reap provides a more flexible interface that permits programmers to reclaim unused memory.

9. 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 finding their system allocator to be inadequate should consider an off-the-shelf solution like Reap or the Lea allocator rather than writing a custom allocator.

10. Conclusions

Despite the widespread belief that custom allocators should be used in order to 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 exception is region allocators, which often outperform general-purpose allocation.

We show that regions can come at an excessive cost in memory consumption. With our implementation of the Reap allocator, we demonstrate an extended general-purpose memory allocator that provides a broad range of region semantics while achieving the performance of region-based allocators and the flexibility of general-purpose allocation.

We believe custom allocators are suitable for managing special kinds or specific ranges of memory, like DMA, shared or embedded memory. But our results indicate that an extension of the general-purpose memory management interface combined with an efficient implementation effectively removes custom allocators as an appropriate optimization technique.

11. Acknowledgements

Thanks to Stephen Chenney 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 the Vulcan team (Andy Edwards in particular).

12. References

[1] Apache Foundation. Apache Web server. http://www.apache.org.

[2] 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)*, Snowbird, Utah, June 2001.

[3] 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.

[4] Dov Bulka and David Mayhew. *Efficient C++*. Addison-Wesley, 2001.

[5] 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)*, Snowbird, Utah, June 2001.

[6] Margaret A. Ellis and Bjarne Stroustrop. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[7] Boris Fomitchev. STLport. http://www.stlport.org/.

[8] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[9] Free Software Foundation. GCC Home Page. http://gcc.gnu.org/.

[10] 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.

[11] 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.

[12] Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries. http://www.dent.med.uni-muenchen.de/˜ wmglo/malloc-slides.html.

[13] 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.

[14] 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.

[15] David R. Hanson. *C Interfaces and Implementation*. Addison-Wesley, 1997.

[16] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[17] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *International Symposium on Memory Management*, Vancouver, B.C., Canada, 1998.

[18] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[19] Scott Meyers. *Effective C++*. Addison-Wesley, 1996.

[20] Scott Meyers. *More Effective C++*. Addison-Wesley, 1997.

[21] Bartosz Milewski. *C++ In Action: Industrial-Strength Programming Techniques*. Addison-Wesley, 2001.

[22] J. Richter. *Advanced Windows: The Developer's Guide to the Win32 API for Windows NT and Windows 95*. Microsoft Press, 1995.

[23] Jeffrey Richter. *Advanced Windows: the developer's guide to the Win32 API for Windows NT 3.5 and Windows 95*. Microsoft Press, Bellevue, WA, USA, 1995.

[24] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, 1967.

[25] Colin Runciman and Niklas Rojemo. Lag, drag and postmortem heap profiling. In *Implementation of Functional Languages Workshop*, Bastad, Sweden, September 1995.

[26] SGI. The Standard Template Library for C++: Allocators. http://www.sgi.com/tech/stl/Allocators.html.

[27] 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.

[28] Standard Performance Evaluation Corporation. SPEC2000. http://www.spec.org.

[29] Standard Performance Evaluation Corporation. SPEC95. http://www.spec.org.

[30] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. (Addison-Wesley), 1991.

[31] Suzanne Pierce. PPRC: Microsoft's Tool Box. http://research.microsoft.com/research/pprc/mstoolbox.asp.

[32] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[33] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. In *Software Practice & Experience*, number 26, pages 1–18. Wiley, 1996.

[34] 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.

[35] Benjamin G. Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23(7):733–756, 1993.