The Dissertation Committee for Emery D. Berger

certifies that this is the approved version of the following dissertation:

# Memory Management for High-Performance Applications

Committee:

Kathryn S. McKinley, Supervisor

James C. Browne

Michael D. Dahlin

Stephen W. Keckler

Benjamin G. Zorn

# Memory Management for High-Performance Applications

by

**Emery D. Berger, M.S., B.S.**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

August 2002

# Memory Management for High-Performance Applications

Publication No. _____

Emery D. Berger, Ph.D.

The University of Texas at Austin, 2002

Supervisor: Kathryn S. McKinley

Fast and effective memory management is crucial for many applications, including web servers, database managers, and scientific codes. Unfortunately, current memory managers often do not meet these requirements. These memory managers also do not provide the support required to avoid memory leaks in server applications, and prevent multithreaded applications from scaling on multiprocessor systems. In fact, current memory managers cause some applications to slow down with the addition of more processors.

In this thesis, we address these memory management problems for high-performance applications. We develop a memory management infrastructure called ***heap layers*** that allows us to compose efficient general-purpose memory managers from components. We show that most *custom* memory managers achieve slight or no performance improvements over current general-purpose memory managers. We build a hybrid memory manager called ***reap*** that combines region-based and general-purpose memory management, simplifying memory management for server applications. We show that previous memory managers suffer from serious problems when running on multiprocessors, including allocator-induced false sharing and a blowup in memory consumption. We present a scalable concurrent memory manager called ***Hoard*** that provably avoids these problems and significantly improves application performance.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Memory management is one of the most enduring problems in computer science. The first papers on the subject appeared in the early sixties. Wilson's memory management survey papers from the nineties include over 220 citations [80, 81], and the subject remains an area of active research. Much current research on memory management focuses on automatic storage allocation (garbage collection). While garbage collection represents a software engineering advance over explicit storage allocation, the performance of garbage-collected languages (like Java) continues to lag behind those that use explicit memory management (like C and C++) [25, 58, 77]. Because we are concerned here with high-performance applications, we will focus exclusively on explicit memory management (which we hereafter refer to simply as "memory management" or "memory allocation").

Despite the long history of memory management research, memory managers continue to be a source of performance and robustness problems for application software. First, the time spent allocating and freeing objects accounts for a significant proportion of the runtime of today's increasingly object-oriented applications (as high as 40%). In an effort to improve the performance of these allocation-intensive applications, programmers frequently write *ad hoc* or ***custom*** memory managers [18, 54, 56] that take advantage of certain allocation behaviors known to the programmer. Second, lock contention in current memory managers prevents many multithreaded programs from scaling [48]. In fact, these memory managers cause some applications to *slow down* with additional processors. Finally, general-purpose memory managers do not

support the *teardown* of objects (bulk deletion) associated with cancelled transactions or connections. This support is crucial in order to avoid memory leaks in some applications [2].

In this thesis, we examine each of these problems in turn. Constructing high-quality memory managers presents a number of significant engineering challenges. We evaluate infrastructures designed to simplify memory manager construction but find that these incur unnecessary function call and virtual dispatch overhead. We develop a C++ infrastructure called ***heap layers*** that is both fast and flexible [12]. Heap layers permits the construction of high-quality memory managers by mixing and matching components using templated classes known as "mixins". We build both custom and general-purpose allocators using heap layers and we show that the resultant memory managers match or exceed the performance and memory efficiency of their hand-coded counterparts. Heap layers thus provides a framework for composing high-quality memory managers from well-tested components.

We then conduct a comprehensive evaluation of the performance of applications that use custom memory managers. Contrary to popular belief, we find that the Lea allocator [50], a high-quality general-purpose memory manager, provides nearly the same performance as custom memory managers for most (but not all) of the applications using custom memory managers that we tested. These results argue for solving memory management problems in a general-purpose framework.

To address the special needs of server applications, we develop a new memory management abstraction called ***reaps*** [13]. Reaps are a hybrid of heaps and *regions*, which permit only bulk deletion of all objects within separate areas of memory. We verify that programs using regions can achieve significant performance gains over general-purpose memory management. We show that our implementation of reaps nearly matches the performance of regions. More importantly, we show that reaps provide greater flexibility for managing memory and simplify the coding of server applications and offer the opportunity to reduce memory consumption.

We then address the problem of scalable concurrent memory management for multiprocessors. For multiprocessor applications, we find that there are serious problems with current general-purpose memory managers. We identify three key problems: ***lock contention*** in the memory manager, ***allocator-induced false sharing*** of heap objects, which can signficantly degrade performance, and a ***blowup*** in memory con-

sumption that can range from $P$ (the number of processors) to *unbounded* memory consumption, which causes programs to fail by exhausting all available memory.

We develop ***Hoard***, a fast, highly scalable concurrent memory manager that largely avoids false sharing and is memory efficient [11]. Hoard is the first allocator to simultaneously provide these features. Hoard combines one global heap and per-processor heaps with a novel discipline that provably bounds memory consumption and has very low synchronization costs in the common case. Our results on eleven programs demonstrate that Hoard yields low average fragmentation and improves overall program performance over the standard Solaris memory manager by up to a factor of 60 on 14 processors, and up to a factor of 18 over the next best memory manager we tested. These results show that Hoard is a scalable concurrent memory manager that can dramatically improve application performance.

## 1.1 Summary of contributions

In this thesis, we both identify and solve a number of memory management problems for high-performance applications. We develop a memory management infrastructure that allows us to compose efficient general-purpose memory managers from components. We show that most previous custom memory managers achieve slight or no performance improvements over current general-purpose memory managers. We build a hybrid memory manager that allows server applications to easily avoid memory leaks. We show that previous memory managers suffer from a number of problems when used by concurrent applications running on multiprocessors, and we present a concurrent memory manager that provably avoids these problems.

## 1.2 Thesis Outline

This thesis is organized as follows. In Chapter 2, we present background material and discuss recent related work in memory management, describing three modern memory managers and focusing on memory management infrastructures and custom memory management. We then describe our experimental methodology in Chapter 3. In Chapter 4, we present the heap layers infrastructure and use it to build two high-performance general-purpose memory managers that perform comparably to their highly-tuned C counterparts. We then

compare custom and general-purpose memory managers in Chapter 5 and demonstrate that using custom memory managers generally does not yield significant performance improvements for uniprocessor applications. In Chapter 6, we address the special needs of server applications with reaps, and demonstrate that these provide increased flexibility and comparable performance to regions. We then focus on multiprocessor memory management in Chapter 7. We first discuss previous work and then describe Hoard in detail. Finally, we summarize our contributions and discuss future research directions in Chapter 8.

# Chapter 2

# Background and Related Work

In this chapter, we first give a brief introduction to general-purpose memory managers. We describe three representative memory managers in order to introduce some key memory management concepts. We then present related work in two of the three areas of memory management that are the subject of this thesis: memory management infrastructures and custom memory managers. We describe related work on concurrent memory management in Chapter 7 where we discuss allocator-induced false sharing and blowup in detail.

## 2.1 Basic Concepts

Most programs rely on dynamic memory management, the creation and deletion of objects at runtime (as opposed to static memory management, e.g. for fixed-sized arrays). C programmers call `void * ptr = malloc(s)` to obtain a pointer to `s` bytes of memory, and call `free(ptr)` to release this memory for future requests. The C++ interface to the memory manager is type-safe: the programmer calls `Object * p = new Object` to simultaneously allocate and construct an `Object`, and calls `delete p` to finalize and deallocate it.

Programmers expect memory managers to be both fast and memory-efficient, consuming as little memory as possible while rapidly satisfying all requests for memory. If the memory manager were free

to relocate already-allocated objects, it would always be able to tightly bound memory consumption [24]. However, the memory models of languages like C and C++ do not permit the underlying memory manager to move allocated objects. These languages therefore require *non-moving* memory managers (often referred to here and elsewhere in the literature as ***memory allocators***, or simply *allocators*). All such memory managers can suffer from ***fragmentation***, or wasted memory. In the worst-case, this fragmentation can be as high as a factor of the logarithm of the ratio of the largest object size divided by the smallest object size [60]. This bound means that a program that manages 8-byte and 8K objects could consume 10 times as much memory as required ($\log(8192/8) = 10$). However, the average fragmentation induced by a number of memory management algorithms on real applications is low (around 1.1) [46].

Drawing from experience and empirical studies [34, 46, 80], most current memory managers perform an approximation of ***best-fit***, which provides both speed and reasonably low fragmentation for most applications. These memory managers attempt to satisfy memory requests with best-fitting chunks of memory – chunks that are the same size or slightly larger than the requests. In an attempt to maximize memory utilization, many memory managers perform ***splitting*** (breaking large objects into smaller ones) and ***coalescing*** (combining adjacent free objects). Splitting reduces ***internal fragmentation***, wasted space inside allocated objects, while coalescing can reduce ***external fragmentation*** (all other wasted space).

The language specifications of C and C++ impose additional requirements on the memory manager. Objects must be double-word aligned in order to be able to hold double-precision numbers (a requirement for many architectures). Therefore, the minimum object size is generally eight bytes. All object requests from 1 to 8 bytes therefore belong to the same ***size class***, or range of object sizes that the memory manager treats identically.

## 2.2 General-Purpose Memory Management

Rather than discussing the vast number of general-purpose memory management algorithms[1], we focus on three representative memory managers in order to introduce some key concepts. Here we describe the

[1] See Wilson *et al.* for an extensive survey [80].

Kingsley allocator used in BSD 4.2 [80], the Windows XP allocator [59], and the Lea allocator [50]. These allocators are in widespread use [2] and span the spectrum between maximizing speed and minimizing memory consumption.

The Kingsley allocator is a power-of-two ***segregated fits*** allocator: all allocation requests are rounded up to the next power of two, and objects from different size classes are never combined. This rounding can lead to severe internal fragmentation, because in the worst case, it allocates twice as much memory as requested. Further, an object allocated for a given size cannot be reused for another size: the allocator performs no splitting or coalescing. This algorithm is well known to be among the fastest memory allocators (avoiding relatively expensive splitting and coalescing operations) although it is among the worst in terms of fragmentation [46].

The Lea allocator is an approximate best-fit allocator that manages objects differently based on their size. The Lea allocator manages small objects (smaller than 64 bytes) using exact-size quicklists (one linked list of freed objects for each multiple of 8 bytes). Requests for a medium-sized object (64 bytes to 128K) and certain other events trigger the Lea allocator to coalesce all of the objects in these quicklists in the hope that this reclaimed space can be reused for the medium-sized object. In other words, the coalescing of small objects is *deferred* until this trigger condition occurs. The Lea allocator performs *immediate* coalescing and splitting of medium-sized objects to approximate best-fit. Objects larger than 128K are allocated and freed using the virtual memory mapping functions. The Lea allocator is the best overall allocator (in terms of the combination of speed and memory usage) of which we are aware [46].

The Windows XP allocator is a best-fit allocator with 127 exact-size quicklists (one linked list of freed objects for each multiple of 8 bytes). Objects larger than 1024 bytes are obtained from a sorted linked list, sacrificing speed for a good fit. When using the multithreaded version of the library, the allocator manages quicklists using atomic operations rather than locks.

[2] The Linux allocator (in GNU libc) is based on the Lea allocator[31].

## 2.3 Memory Management Infrastructures

We know of only two previous infrastructures for building memory managers: *vmalloc*, by Vo, and *CMM*, by Attardi, Flagella, and Iglio. We describe the key differences between their systems and ours, focusing on the performance and flexibility advantages that heap layers provide.

### 2.3.1 Vmalloc

The most successful customizable memory manager of which we are aware is the *vmalloc* allocator [78]. Vmalloc lets the programmer define multiple regions (distinct heaps) with different disciplines for each. The programmer performs customization by supplying user-defined functions and `structs` that manage memory. By chaining these together, vmalloc does provide the possibility of composing heaps. Each abstraction layer pays the penalty of a function call. This approach often prevents many useful optimizations, in particular method inlining. The vmalloc infrastructure limits the programmer to a small set of functions for memory allocation and deallocation; a programmer cannot add new functionality or new methods as we describe in Section 4.4.1. Vmalloc does not provide a way to delete heaps and reclaim all of their memory in one step. These limitations dramatically reduce vmalloc's usefulness as an extensible infrastructure.

### 2.3.2 CMM

Attardi, Flagella, and Iglio created an extensive C++-based system called the Customizable Memory Management (CMM) framework [4, 5]. The primary focus of the CMM framework is garbage collection. The only non-garbage collected heaps provided by the framework are a single "traditional manual allocation discipline" heap (whose policy the authors do not specify) called UncollectedHeap and a zone allocator called TempHeap. A programmer can create separate regions by subclassing the abstract class CmmHeap, which uses virtual methods to obtain and reclaim memory. For every memory allocation, deallocation, and crossing of an abstraction boundary, the programmer must thus pay the cost of one virtual method call. As in vmalloc, this approach often prevents compiler optimizations across method boundaries. The virtual method approach also limits flexibility. In CMM, subclasses cannot implement functions not already provided by

virtual methods in the base heap. Also, since class hierarchies are fixed, it is not possible to have one class (such as FreelistHeap, described in Section 4.1.1) with two different parent heaps in different contexts.

## 2.4 Custom Memory Management

### 2.4.1 Construction and Use of Custom Memory Managers

Most academic research on special-purpose (custom) allocation has focused on profile-based optimization of general-purpose allocation. Grunwald and Zorn's CustoMalloc builds memory allocators from allocation traces, optimizing the allocator based on the range of object sizes and their frequency of usage [34]. Other profile-based allocators use lifetime information to improve performance and reference information to improve locality for explicit memory management [7, 62].

Two custom memory allocators are especially popular and merit special attention. ***Freelist-based allocators*** [53, 56, 71] keep same-sized objects on a linked-list, yielding very fast allocation and deallocation of these objects (avoiding splitting, coalescing, and size calculations). ***Region allocators*** [29, 30, 37, 61, 73] allocate space for objects from large chunks of memory obtained from the general-purpose memory manager. Object allocation in regions is very fast, consisting of bumping a pointer and checking to ensure that the current chunk still has space (getting a new one from the general-purpose memory manager if needed). A region allocator cannot free objects within a region. Rather, the region allocator deletes all of the chunks at once when the region as a whole is no longer needed.

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 [18]. Meyers describes in detail the use of a freelist-based per-class custom allocator in "Effective C++" [53] and returns to the topic of custom allocators in the sequel [54]. Milewski also discusses per-class allocators as an optimization technique [56]. Hanson devotes a chapter to an implementation of regions ("arenas"), citing both the speed and software engineering benefits of regions as motivation [38]. Ellis and Stroustrup describe the syntactic facilities that allow overloading `operator new`, simplifying the use of custom allocators in C++ [23], and Stroustrup describes per-class allocators

that use these facilities [71]. 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 [37, 61] has recently attracted attention as an alternative to garbage collection. Tofte and Talpin present a system that provides automatic region-based memory management for ML [73]. 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 [29, 30]. 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.

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 [59]. Vmalloc, a memory allocation infrastructure that we describe above, also provides (non-nested) regions that permit individual object deletion [78]. We show in Section 5.4.3 that neither of these implementations match the performance of regions or reaps, and reaps capture the same semantics.

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 [16]. These regions, while somewhat different from traditional region-based allocators in that they are associated with one or more computations [10], 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.

### 2.4.2 Evaluation of Custom Memory Management

The only previous work evaluating the impact of custom memory allocators is by Zorn. Zorn compared custom ("domain-specific") allocators to general-purpose memory allocators [82]. 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 that performed in our work 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.[3] 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.

In this section, we have discussed several general-purpose memory managers, existing memory management infrastructures, and custom memory managers. In Chapter 4, we present our heap layers infrastructure and show how it improves on past work. We perform a detailed evaluation of custom memory managers in Chapter 5, comparing these to general-purpose memory managers, which we find perform identically or nearly as well in most cases. However, existing general-purpose memory managers do not provide adequate support for server-style and multithreaded applications, which we address in Chapters 6 and 7. In the next chapter, we present our experimental methodology that we use in the remainder of this thesis.

[3] These allocators are all variants of what we call per-class allocators in Section 5.2.2.

# Chapter 3

# Experimental Methodology

To evaluate memory managers, we use analysis whenever possible but also rely on a large number of experiments. Here we describe the different sets of benchmarks we use in this thesis, our hardware platforms and our experimental methodology.

## 3.1 Benchmarks

We have gathered two suites of benchmarks that we use to evaluate a wide range of memory management characteristics. We call these the Memory-Intensive and General-Purpose benchmark suites, and use them in this thesis to measure different aspects of memory management.

### 3.1.1 Memory-Intensive Benchmarks

The Memory-Intensive Benchmark suite comprises a number of memory-intensive programs, most of which were described by Zorn and Wilson [35, 46] and shown in Table 3.1. A memory-intensive program has at least one of the following characteristics: it allocates and frees many objects, it consumes a significant amount of memory, or it spends a lot of its time performing memory operations.

The suite includes the following programs: *cfrac* factors arbitrary-length integers, *espresso* is an optimizer for programmable logic arrays, *lindsay* is a hypercube simulator, *LRUsim* analyzes locality in

| Memory-Intensive Benchmarks | | |
|---|---|---|
| Benchmark | Description | Input |
| *cfrac* | factors numbers | a 36-digit number |
| *espresso* | optimizer for PLAs | *test2* |
| *lindsay* | hypercube simulator | *script.mine* |
| *LRUsim* | a locality analyzer | an 800MB trace |
| *Perl* | Perl interpreter | *perfect.in* |
| *roboop* | Robotics simulator | included benchmark |

Table 3.1: Memory-intensive benchmarks.

| Memory-Intensive Benchmark Statistics | | | | | | |
|---|---|---|---|---|---|---|
| Benchmark | Objects | Total memory | Max in use | Avg. size | Memory ops | Memory ops/sec |
| *cfrac* | 10,890,166 | 222,745,704 | 176,960 | 20 | 21,780,289 | 1,207,862 |
| *espresso* | 4,477,737 | 1,130,107,232 | 389,152 | 252 | 8,955,367 | 218,276 |
| *lindsay* | 108,862 | 7,418,120 | 1,510,840 | 68 | 217,678 | 72,300 |
| *LRUsim* | 39,139 | 1,592,992 | 1,581,552 | 41 | 78,181 | 94 |
| *perl* | 8,548,435 | 162,451,960 | 293,928 | 19 | 17,091,308 | 257,809 |
| *roboop* | 9,268,221 | 332,058,248 | 16,376 | 36 | 18,536,397 | 1,701,786 |

Table 3.2: Statistics for the memory-intensive benchmarks. We divide by runtime with the Lea allocator to obtain memory operations per second.

reference traces, *perl* is the Perl interpreter included in SPEC2000 (253.perlbmk), and *roboop* is a robotics simulator. As Table 3.2 shows, these programs exercise memory allocator performance in both speed and memory efficiency. This table also includes the number of objects allocated and their average size. The programs' footprints range from just 16K (for *roboop*) to over 1.5MB (for *LRUsim*). For all of the programs except *lindsay* and *LRUsim*, the ratio of total memory allocated to the maximum amount of memory in use is large, showing that they allocate and free many objects. The programs' rates of memory allocation and deallocation (memory operations per second) range from under one hundred to almost two million per second. Except for *LRUsim*, memory operations account for a significant portion of the runtime of these programs.

### 3.1.2 General-Purpose Benchmarks

The General-Purpose Benchmark suite comprises programs drawn from the integer SPEC95 and SPEC2000 benchmark suites [65]. The SPEC benchmarks are CPU-intensive and so are useful for measuring CPU

| General Benchmarks | | |
|---|---|---|
| *164.gzip* | GNU zip data compressor [65] | *test/input.compressed 2* |
| *181.mcf* | Vehicle scheduler [65] | *test-input.in* |
| *186.crafty* | Chess program [65] | *test-input.in* |
| *252.eon* (C++) | Ray tracer [65] | *test/chair.control.cook* |
| *253.perlbmk* | Perl interpreter [65] | *perfect.pl b 3* |
| *254.gap* | Groups language interpreter [65] | *test.in* |
| *255.vortex* | Object-oriented DBM [65] | *test/lendian.raw* |
| *300.twolf* | CAD placement & routing [65] | *test.net* |
| *espresso* | Optimizer for PLAs [66] | *test2* |
| *lindsay* (C++) | Hypercube simulator [80] | *script.mine* |

Table 3.3: General-purpose benchmarks and inputs. Programs not written in C++ are written in C.

| Benchmark Statistics | | | | | | |
|---|---|---|---|---|---|---|
| Benchmark | Total objects | Max objects in use | Avg obj. size (in bytes) | Total memory (in bytes) | Max in use (in bytes) | Mem. operations (% of runtime) |
| **general-purpose allocation** | | | | | | |
| *164.gzip* | 1,307 | 72 | 6108 | 7,983,304 | 6,615,288 | 0.1% |
| *181.mcf* | 54 | 52 | 1,789,028 | 96,607,514 | 96,601,049 | 1.5% |
| *186.crafty* | 87 | 86 | 10,206 | 887,944 | 885,520 | 0.0% |
| *252.eon* | 1,647 | 803 | 31 | 51,563 | 33,200 | 0.4% |
| *253.perlbmk* | 8,888,870 | 5,813 | 16 | 144,514,214 | 284,029 | 12.6% |
| *254.gap* | 50 | 48 | 1,343,614 | 67,180,715 | 67,113,782 | 0.0% |
| *255.vortex* | 186,483 | 53,087 | 357 | 66,617,881 | 17,784,239 | 1.9% |
| *300.twolf* | 9,458 | 1,725 | 56 | 532,177 | 66,891 | 0.9% |
| *espresso* | 4,483,621 | 4,885 | 249 | 1,116,708,854 | 373,348 | 10.8% |

Table 3.4: Statistics for the General-Purpose Benchmark suite.

performance. We use these programs as a baseline for understanding the behavior of memory managers on programs that do not generally make intensive use of the memory allocator.

## 3.2 Platforms

For uniprocessor experiments, we use Intel-based systems running Windows. Programs were compiled with Visual C++ 6.0 and run on one of two dedicated personal computers, PC Platform 1 and 2. Table 3.5 describes all of our platforms in detail.

We conducted multiprocessor experiments on the Sun platform, a dedicated Enterprise E5000. Nearly all programs (including the allocators) were compiled using the GNU C++ compiler version 2.80

| *Platform* | *CPU* | *OS* | *RAM* | *Cache sizes* |
|---|---|---|---|---|
| PC Platform 1 | Pentium II, 366 MHz (1) | Windows 2000 | 128 MB | L2: 256K (unified), L1: 16K |
| PC Platform 2 | Pentium III, 600 MHz (1) | Windows XP | 320 MB | L2: 256K (unified), L1: 16K |
| Sun Platform | UltraSparc, 400 MHz (14) | Solaris 7 | 2 GB | L2: 4MB (unified), L1: 16K |

Table 3.5: Platform characteristics. The number in parenthesis after CPU clock speed indicates the number of processors.

at the highest possible optimization level (`-O6`). We use GNU C++ because we encountered errors when we used high optimization levels for the vendor compiler (Sun Workshop compiler version 5.0). However, we did use the vendor compiler for the one benchmark (Barnes-Hut), which ran considerably faster than the GNU C++ version.

## 3.3 Execution Environment

In all cases, we performed experiments on dedicated machines. For runtimes, we report the arithmetic mean of at least three runs, after one warm-up run. On the PC platforms, we run programs at real-time priority, preventing all background applications from running at all. These steps ensure that variation in runtime remains minimal (below 1%).

For most of the experiments in this thesis, we substitute memory allocators in existing applications by statically linking in replacement allocators. That is, all calls to `malloc`, etc., are routed away from the system library to our replacement allocator. This approach intercepts all memory operations performed by the application, including those made by library code (e.g., `printf`) and initialization code. We link in a memory allocation tracer that logs all memory operations to gather allocation statistics, including those in the tables above and subsequently in the remainder of this thesis.

# Chapter 4

# Composing High-Performance Memory Managers

Building high-quality general-purpose memory managers presents numerous software engineering challenges. These memory managers must simultaneously be very fast and keep memory consumption as low as possible. Balancing these goals is difficult. The approach used by the Lea allocator is to implement memory operations with large, monolithic C functions (hundreds of lines long) and employing heavy use of macros to avoid function call overhead. This approach yields suitably fast code but at the considerable expense of sacrificing modularity, extensibility, and maintainability.

To address these problems, we present a flexible and efficient infrastructure for building memory managers called *heap layers*. Heap layers provide a foundation for composing memory managers from a collection of reusable components. This infrastructure is based on a combination of C++ templates and inheritance called *mixins* [17]. Mixins are classes whose superclass may be changed. Using mixins allows the programmer to code memory managers as composable layers that a compiler can implement with efficient code. Unlike previous approaches, we show that this technique allows programmers to write highly modular and reusable code with no abstraction penalty. We describe a number of high-performance custom allocators that we built by mixing and matching heap layers. We show that these allocators match or improve performance when compared with their hand-tuned, monolithic C counterparts on a selection of C and C++

programs.

We demonstrate that this infrastructure can be used effectively to build high-performance, general-purpose allocators. We evaluate two general-purpose allocators we developed using heap layers over a period of three weeks, and compare their performance to the Kingsley allocator, one of the fastest general-purpose allocators, and the Lea allocator, an allocator that is both fast and memory-efficient. While the current heap layers allocator does not quite achieve the fragmentation and performance of the Lea allocator, it comes close. The Lea allocator is highly tuned and has undergone many revisions over a period of more than seven years [50].

The remainder of this chapter is organized as follows. In Section 4.1, we describe how we use mixins to build heap layers and demonstrate how we can mix and match a few simple heap layers to build and combine allocators. We briefly discuss our experimental methodology in Section 4.2. In Section 4.3, we show how we implement some real-world custom allocators using heap layers and present performance results. Section 4.4 then describes two general-purpose allocators built with heap layers and compares their runtime and memory consumption to the Kingsley and Lea allocators. We describe some of the software engineering benefits of heap layers in Section 4.5, and in Section 4.6, we show how heap layers provide a convenient infrastructure for memory allocation experiments. We use this infrastructure to build and explore allocator performance and memory utilization in the remainder of the thesis.

## 4.1 Heap Layers

While programmers often write memory allocators as monolithic pieces of code, they tend to think of them as consisting of separate pieces. Most general-purpose allocators treat objects of different sizes differently. The Lea allocator uses one algorithm for small objects, another for medium-sized objects, and yet another for large objects. Conceptually at least, these heaps consist of a number of separate heaps that are combined in a hierarchy to form one big heap.

The standard way to build components like these in C++ uses virtual method calls at each abstraction boundary. The overhead caused by virtual method dispatch is significant when compared with the cost of

memory allocation. This implementation style also greatly limits the opportunities for optimization since the compiler often cannot optimize across method boundaries. Building a class hierarchy through inheritance also fixes the relationships between classes in a single inheritance structure, making reuse difficult.

To address these concerns, we use *mixins* to build our heap layers. Mixins are classes whose superclass may be changed (they may be reparented) [17]. The C++ implementation of mixins [75] consists of a templated class that subclasses its template argument:

```
template <class Super>
class Mixin : public Super {};
```

Mixins overcome the limitation of a single class hierarchy, enabling the reuse of classes in different hierarchies. For instance, we can use `Child` in two different hierarchies, `Child` $\rightarrow$ `Parent1` and `Child` $\rightarrow$ `Parent2` (where the arrow means "inherits from"), by defining `Child` as a mixin and composing the classes as follows:

```
class Composition1 : public Child<Parent1> {};
class Composition2 : public Child<Parent2> {};
```

A heap layer is a mixin that provides a `malloc` and `free` method and that follows certain coding guidelines. The `malloc` function returns a memory block of the specified size, and the `free` function deallocates the block. As long as the heap layer follows the guidelines we describe below, programmers can easily compose heap layers to build heaps. One layer can obtain memory from its parent by calling `SuperHeap::malloc()` and can return it with `SuperHeap::free()`. Heap layers also implement thin wrappers around system-provided memory allocation functions like `malloc`, `sbrk`, or `mmap`. We term these thin-wrapper layers *top heaps*, because they appear at the top of any hierarchy of heap layers.

We require that heap layers adhere to the following coding guidelines in order to ensure composability. First, `malloc` must correctly handle NULLs returned by `SuperHeap::malloc()` to allow an out-of-memory condition to propagate through a series of layers or to be handled by an exception-handling layer. Second, the layer's destructor must free any memory held by the layer. This action allows heaps

composed of heap layers to be deleted in their entirety in one step.[1]

### 4.1.1 Example: Composing a Per-Class Allocator

One common way of improving memory allocation performance is to allocate all objects from a highly-used class from a per-class pool of memory. Because all such objects are the same size, memory can be managed by a simple singly-linked freelist [48]. Programmers often implement these per-class allocators in C++ by overloading the `new` and `delete` operators for the class.[2]

Below we show how we can combine two simple heap layers to implement per-class pools without changing the source code for the original heap layer class. We first define a utility class called PerClassHeap that allows a programmer to adapt a class to use any heap layer as its allocator:

```
template <class Object, class SuperHeap>
class PerClassHeap : public Object {
public:
  inline void * operator new (size_t sz) {
    return getHeap().malloc (sz);
  }
  inline void operator delete (void * ptr) {
    getHeap().free (ptr);
  }
private:
  static SuperHeap& getHeap (void) {
    static SuperHeap theHeap;
    return theHeap;
  }
};
```

We build on the above with a very simple heap layer called FreelistHeap. This layer implements a linked list of free objects of the same size. `Malloc` removes one object from the freelist if one is available, and `free` places memory on the freelist for later reuse. This approach is a common idiom in allocators because it provides fast allocation and freeing and reuses the most-recently freed memory which may provide good locality. However, it is limited to handling only one size of object. The code for FreelistHeap appears in

[1] This functionality will prove useful for the development of region-like allocators.

[2] We show in Chapter 5 that such custom memory managers are generally a waste of time but use them only as a simple demonstration of heap layers.

Figure 4.1 without the error checking included in the actual code to guarantee that all objects are the same size.

```
template <class SuperHeap>
class FreelistHeap : public SuperHeap {
public:
  FreelistHeap (void)
    : myFreeList (NULL)
  {}
  ~FreelistHeap (void) {
    // Delete everything on the freelist.
    void * ptr = myFreeList;
    while (ptr != NULL) {
      void * oldptr = ptr;
      ptr = (void *) ((freeObject *) ptr)->next;
      SuperHeap::free (oldptr);
    }
  }
  inline void * malloc (size_t sz) {
    // Check the freelist first.
    void * ptr = myFreeList;
    if (ptr == NULL) {
      ptr = SuperHeap::malloc (sz);
    } else {
      myFreeList = myFreeList->next;
    }
    return ptr;
  }
  inline void free (void * ptr) {
    // Add this object to the freelist.
    ((freeObject *) ptr)->next = myFreeList;
    myFreeList = (freeObject *) ptr;
  }
private:
  class freeObject {
  public:
    freeObject * next;
  };
  freeObject * myFreeList;
};
```

Figure 4.1: The implementation of FreelistHeap.

We can now combine PerClassHeap and FreelistHeap with mallocHeap (a thin layer over the system-supplied `malloc` and `free`) to make a subclass of *Foo* that uses per-class pools.

```
class SameSizeFoo :
public
  PerClassHeap<Foo, FreelistHeap<mallocHeap> >{};
```

### 4.1.2 A Library of Heap Layers

We have built a comprehensive library of heap layers that allows programmers to build a range of memory allocators with minimal effort by composing these ready-made layers. Figure 4.4 lists a number of these layers, which we group into the following categories:

**Top heaps.** A "top heap" is a heap layer that provides memory directly from the system and at least one appears at the top of any hierarchy of heap layers. These thin wrappers over system-based memory allocators include mallocHeap (which uses the system `malloc` and `free`) mmapHeap (which uses `mmap` and `munmap`), and sbrkHeap (which uses `sbrk()` for UNIX systems and an `sbrk()` emulator for Windows).

**Building-block heaps.** Programmers can use these simple heaps in combination with other heaps described below to implement more complex heaps. We provide an adapter called AdaptHeap that lets us embed a dictionary data structure inside freed objects so we can implement variants of FreelistHeap, including DLList, a FIFO-ordered, doubly-linked freelist that allows constant-time removal of objects from anywhere in the freelist. This heap supports CoalesceHeap, which performs splitting and coalescing of adjacent objects belonging to different freelists into one object.

**Combining heaps.** These heaps combine a number of heaps to form one new heap. These include two segregated-fits layers, SegHeap and StrictSegHeap (described in Section 4.4.1), and HybridHeap, a heap that uses one heap for objects smaller than a given size and another for larger objects.

**Utility layers.** Utility layers include ANSIWrapper, which provides ANSI-C compliant behavior for `malloc` and `free` to allow a heap layer to replace the system-supplied allocator. A number of layers supply multithreaded support, including LockedHeap, which *code-locks* a heap for thread safety (acquires a

lock, performs a `malloc` or `free`, and then releases the lock), and ThreadHeap and PHOThreadHeap, which implement finer-grained multithreaded support. Error handling is provided by ThrowExceptionHeap, which throws an exception when its superheap is out of memory. We also provide heap debugging support with DebugHeap, which tests for multiple frees and other common memory management errors.

**Object representation.** SizeHeap maintains object size in a header just preceding the object. CoalesceableHeap does the same but also records whether each object is free in the header of the next object in order to facilitate coalescing.

**Special-purpose heaps.** We provide a number of heaps optimized for managing objects with known lifetimes, including two heaps for stack-like behavior (ObstackHeap and XallocHeap, described in Sections 4.3.1 and 4.3.2) and a region-based allocator (ZoneHeap).

**General-purpose heaps.** We also implement two heap layers useful for general-purpose memory allocation: KingsleyHeap and LeaHeap, described in Sections 4.4.1 and 4.4.2.

## 4.2 Experimental Methodology

We wrote these heap layers in C++ and implemented them as a series of include files. We then used these heap layers to replace a number of allocators. For C++ programs, we used these heap layers directly (e.g., `kHeap.free(p)`). When replacing custom allocators in C programs, we wrapped the heap layers with a C API. When replacing the general-purpose allocators, we redefined `malloc` and `free` and the C++ operators `new` and `delete` to refer to the desired allocator. We executed these programs on PC Platform 1 (see 3.2).

## 4.3 Building Special-Purpose Allocators

In this section, we investigate the performance implications of building allocators using heap layers. Specifically, we evaluate the performance of two applications (197.parser and 176.gcc from the SPEC2000 bench-

mark suite) that make extensive use of custom allocators, as described in Chapter **??**. We compare the performance of the original carefully-tuned allocators against versions of the allocators that we wrote with heap layers. In Section 4.4, we show similar results for general-purpose memory managers.

### 4.3.1 197.parser

The 197.parser benchmark is a natural-language parser for English written by Sleator and Temperley. It uses a custom allocator the authors call *xalloc* which is optimized for stack-like behavior. This allocator uses a fixed-size region of memory (in this case, 30MB) and always 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. Xalloc can free the entire heap quickly by setting the pointer to the start of the memory region. This allocator is a good example of appropriate use of a custom allocator. As in most custom allocation strategies, it is not appropriate for general-purpose memory allocation. For instance, if an application never frees the last block in use, this algorithm would exhibit unbounded memory consumption.

We replaced xalloc with a new heap layer, XallocHeap. This layer, which we put on top of Mmap-Heap, is the same as the original allocator, except that we replaced a number of macros by inline static functions. We did not replace the general-purpose allocator which uses the Windows 2000 heap. We ran 197.parser against the SPEC test input to measure the overhead that heap layers added. Figure 4.2 presents these results. We were quite surprised to find that using layers actually slightly *reduced* runtime (by just over 1%), although this reduction is barely visible in the graph. The source of this small improvement is due to the increased opportunity for code reorganization that layers provide. When using layers, the compiler can schedule code with much greater flexibility. Since each layer is a direct procedure call, the compiler can decide what pieces of the layered code are most appropriate to inline at each point in the program. The monolithic implementations of `xalloc`/`xfree` in the original can only be inlined in their entirety. Table 4.1 shows that the executable sizes for the original benchmark are the smallest when the allocation functions are not declared inline and the largest when they are inlined, while the version with XallocHeap lies in between (the compiler inlined the allocation functions with XallocHeap regardless of our use of

| 197.parser variant | Executable size |
|---|---|
| original | 211,286 |
| original (inlined) | 266,342 |
| XallocHeap | 249,958 |
| XallocHeap (inlined) | 249,958 |

Table 4.1: Executable sizes for variants of 197.parser.

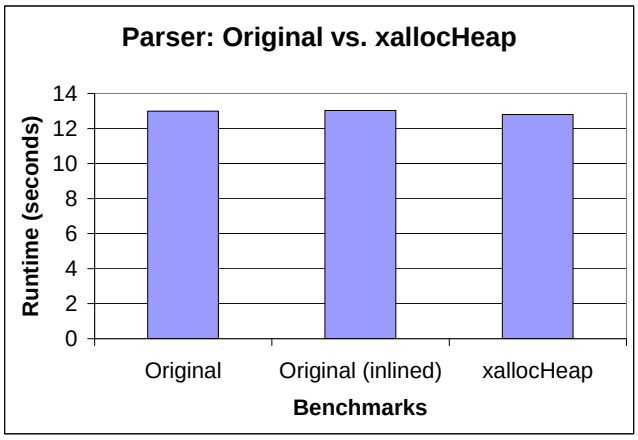


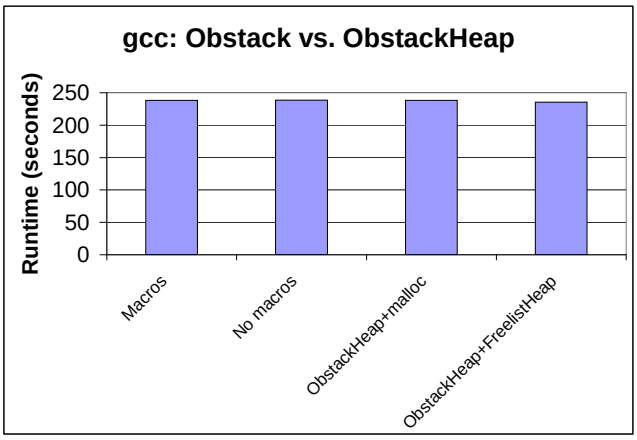

Figure 4.2: Runtime comparison of the original 197.parser custom allocator and xallocHeap.

`inline`). Inspecting the assembly output reveals that the compiler made more fine-grained decisions on what code to inline and thus achieved a better trade-off between program size and optimization opportunities to yield improved performance.

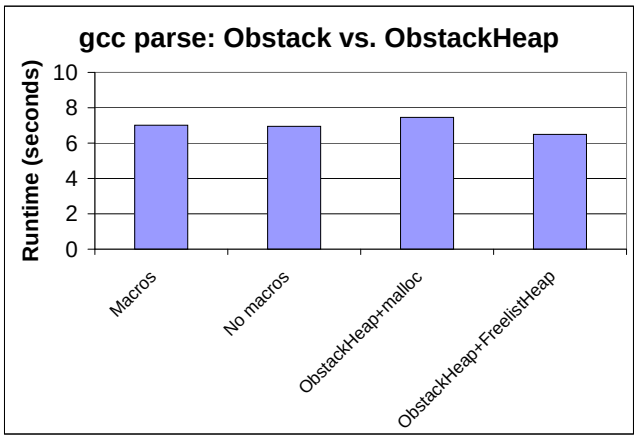
### 4.3.2 176.gcc

Gcc uses *obstacks*, a well-known custom memory allocation library [80]. Obstacks also are designed to take advantage of stack-like behavior, but in a more radical way than xalloc. Obstacks consist of a number of large memory "chunks" that are linked together. Allocation of a block bumps a pointer in the current chunk, and if there is not enough room in a given chunk, the obstack allocator obtains a new chunk from the system. Freeing an object deallocates all memory allocated after that object. Obstacks also support a `grow()` operation. The programmer can increase the size of the current block, and if this block becomes too large for the current chunk, the obstack allocator copies the current object to a new, larger chunk.

Gcc uses obstacks in a variety of phases during compilation. The parsing phase in particular uses




(a) Complete execution of gcc.




(b) gcc's parse phase only.

Figure 4.3: Runtime comparison of gcc with the original obstack and ObstackHeap.

obstacks extensively. In this phase, gcc uses the obstack grow operation for symbol allocation in order to avoid a fixed limit on symbol size. When entering each lexical scope, the parser allocates objects on obstacks. When leaving scope, it frees all of the objects allocated within that scope by freeing the first object it allocated.

Obstacks have been heavily optimized over a number of years and make extensive use of macros. We implemented ObstackHeap in heap layers and provided C-based wrapper functions that implement the obstack API. This effort required about one week and consists of 280 lines of code (around 100 are to implement the API wrappers). By contrast, the GNU obstack library consists of around 480 lines of code and was refined over a period of at least six years.

We ran gcc on one of the reference inputs (scilab.i) and compared two versions of the original gcc with two versions of gcc with ObstackHeap: the original macro-based code, the original with function calls instead of macros, the ObstackHeap version layered on top of mallocHeap, and an ObstackHeap version that uses a FreelistHeap to optimize allocation and freeing of the default chunk size and mallocHeap for larger chunks:

As with 197.parser, we did not replace the general-purpose allocator. Figure 4.3(a) shows the total execution time for each of these cases, while Figure 4.3(b) shows only the parse phase. Layering ObstackHeap on top

```
class ObstackType :
  public ObstackHeap<4096,
    HybridHeap<4096 + 8,  // Obstack overhead
               FreelistHeap<mallocHeap>,
               mallocHeap> {};
```

of FreelistHeap results in an 8% improvement over the original in the parse phase, although its improvement over the original for the full execution of gcc is minimal (just over 1%).

## 4.4 Building General-Purpose Allocators

In this section, we consider the performance implications of building general-purpose allocators using heap layers. Specifically, we compare the performance of the Kingsley and Lea allocators [50] to allocators with very similar architectures created by composing heap layers. Our goal is to understand whether the performance costs of heap layers prevent the approach from being viable for building general-purpose allocators. We map the designs of these allocators to heap layers and then compare the runtime and memory consumption of the original allocators to our heap layer implementations, KingsleyHeap and LeaHeap. To evaluate allocator runtime performance and fragmentation, we use the Memory-Intensive benchmark suite we describe in Section 3.1.1. Memory operations account for a significant portion of their runtime for these benchmarks except for *LRUsim*, and exercise both the speed and memory efficiency of memory allocators.

### 4.4.1 The Kingsley Allocator

We first show how we can build KingsleyHeap, a complete general-purpose allocator using the FreelistHeap layer described in Section 4.1.1 composed with one new heap layer. We show that KingsleyHeap, built using heap layers, performs as well as the Kingsley allocator.

The Kingsley allocator needs to know the sizes of allocated objects so it can place them on the appropriate free list. An object's size is often kept in metadata just before the object itself, but it can be represented in other ways. We can abstract away object representation by relying on a `getSize()` method that must be implemented by a superheap. SizeHeap is a layer that records object size in a header

immediately preceding the object.

```
template <class SuperHeap>
class SizeHeap : public SuperHeap {
public:
  inline void * malloc (size_t sz) {
    // Add room for a size field.
    freeObject * ptr = (freeObject *)
      SuperHeap::malloc (sz + sizeof(freeObject));
    // Store the requested size.
    ptr->sz = sz;
    return (void *) (ptr + 1);
  }
  inline void free (void * ptr) {
    SuperHeap::free ((freeObject *) ptr - 1);
  }
  inline static size_t getSize (void * ptr) {
    return ((freeObject *) ptr - 1)->sz;
  }
private:
  union freeObject {
    size_t sz;
    double _dummy; // for alignment.
  };
};
```

Figure 4.4: The implementation of SizeHeap.

StrictSegHeap provides a general interface for implementing strict segregated fits allocation. Segregated fits allocators divide objects into a number of *size classes*, which are ranges of object sizes that are grouped together (e.g., all objects between 32 and 36 bytes are treated as 36-byte objects). Memory requests for a given size are satisfied directly from the "bin" corresponding to the requested size class. The heap returns deallocated memory to the appropriate bin. StrictSegHeap's arguments include the number of bins, a function that maps object size to size class and size class to maximum size, the heap type for each bin, and the parent heap (for bigger objects). The implementation of StrictSegHeap is 32 lines of C++ code. The class definition appears in Figure 4.8.

We now build KingsleyHeap using these layers. First, we implement helper functions that support power-of-two size classes (integer $\log$ function and exponentiation functions). We can now define KingsleyHeap. We implement KingsleyHeap as a StrictSegHeap with 29 bins and power-of-two size classes

(supporting an object size of up to $2^{32} - 1$ bytes). Each size class is implemented using a FreelistHeap that gets memory from SbrkHeap (a thin layer over `sbrk()`).

```
class KingsleyHeap :
public StrictSegHeap<29, pow2getSizeClass,
         pow2getClassMaxSize,
         SizeHeap<FreelistHeap<SbrkHeap> >,
         SizeHeap<FreelistHeap<SbrkHeap> > > {};
```

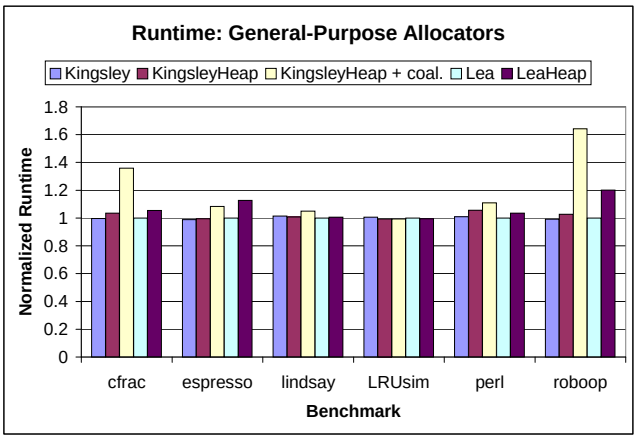
A C++ programmer now uses this heap by declaring it as an object and directly using the `malloc` and `free` calls.

```
KingsleyHeap kHeap;
void * ptr = kHeap.malloc (20);
kHeap.free (ptr);
```

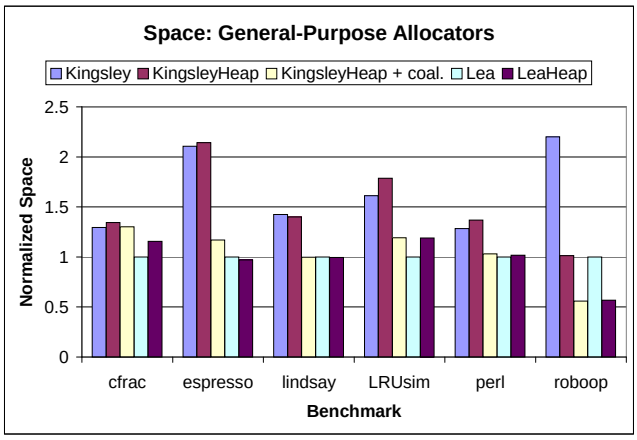
### 4.4.2 The Lea Allocator

Version 2.7.0 of the Lea allocator is a hybrid allocator with different behavior for different object sizes. For small objects ($\leq$ 64 bytes), the allocator uses quick lists; for large objects ($\geq$ 128K bytes), it uses virtual memory (`mmap`), and for medium-sized objects, it performs approximate best-fit allocation [50]. The strategies it employs are somewhat intricate but it is possible to decompose these into a hierarchy of layers.

Figure 4.6 shows the heap layers representation of LeaHeap, which is closely modeled after the Lea allocator. The shaded area represents LeaHeap, while the Sbrk and Mmap heaps depicted at the top are parameters. At the bottom of the diagram, object requests are managed by a SelectMmapHeap, which routes large size requests to be eventually handled by the Mmap parameter. Smaller requests are routed to ThresholdHeap, which both routes size requests to a small and medium heap and in certain instances (e.g., when a sufficiently large object is requested), frees all of the objects held in the small heap. We implemented coalescing and splitting using two layers. CoalesceHeap performs splitting and coalescing, while CoalesceableHeap provides object headers and methods that support coalescing. SegHeap is a more general version




(a) Runtime normalized to the Lea allocator.




(b) Space (memory consumption) normalized to the Lea allocator.

Figure 4.5: Runtime and space comparison of the original Kingsley and Lea allocators and their heap layers counterparts.

of StrictSegHeap described in Section 4.4.1 that searches through all of its heaps for available memory. Not shown in the picture are AdaptHeap and DLList. AdaptHeap lets us embed a dictionary data structure within freed objects, and for LeaHeap, we use DLList, which implements a FIFO doubly-linked list. While LeaHeap is not a complete implementation of the Lea allocator (which includes other heuristics to further reduce fragmentation), it is a faithful model that implements most of its important features, including the hierarchy described here.

We built LeaHeap in a total of three weeks. We were able to reuse a number of layers, including SbrkHeap, MmapHeap, and SegHeap. The layers that implement coalescing (CoalesceHeap and CoalesceableHeap) are especially useful and can be reused to build other coalescing allocators, as we show in Section 4.6. The new layers constitute around 500 lines of code, not counting comments or white space, while the Lea allocator is over 2,000 lines of code. LeaHeap is more flexible than the original Lea allocator. For instance, a programmer can use multiple instances of LeaHeaps to manage distinct ranges of memory and thus provide some memory protection, something that is not possible with the original. Similarly, we can make these heaps thread-safe when needed by wrapping them with a LockedHeap layer. Because of this flexibility of heap layers, we can easily include *both* a thread-safe and non-thread-safe version of the same

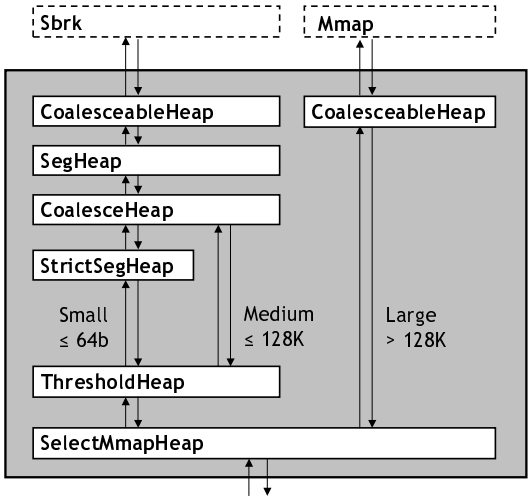



Figure 4.6: A diagram of LeaHeap's architecture.

| Runtime for General-Purpose Allocators | | | | | |
|---|---|---|---|---|---|
| Benchmark | Kingsley | KingsleyHeap | KHeap + coal. | Lea | LeaHeap |
| *cfrac* | 19.02 | 19.75 | 25.94 | 19.09 | 20.14 |
| *espresso* | 40.66 | 40.91 | 44.56 | 41.12 | 46.33 |
| *lindsay* | 3.05 | 3.04 | 3.16 | 3.01 | 3.03 |
| *LRUsim* | 836.67 | 827.10 | 826.44 | 831.98 | 828.36 |
| *perl* | 66.94 | 70.01 | 73.61 | 66.32 | 68.60 |
| *roboop* | 10.81 | 11.19 | 17.89 | 10.89 | 13.08 |

Table 4.2: Runtime (in seconds) for the general-purpose allocators described in this paper.

allocator in the same application so that an application only incurs the cost of locking when necessary.

### 4.4.3 Experimental Results

We ran the benchmarks in Table 3.1 with the Kingsley allocator, KingsleyHeap, KingsleyHeap plus coalescing (which we discuss in Section 4.6), the Lea allocator, and LeaHeap. In Figure 4.5(a) we present a comparison of the runtimes of our benchmark applications normalized to the original Lea allocator (we present the data used for this graph in Table 4.2). The average increase in runtime for KingsleyHeap over the Kingsley allocator is just below 2%. For the two extremely allocation-intensive benchmarks, *cfrac* and *roboop*, the increase in runtime is just over 3%, demonstrating that the overhead of heap layers has minimal impact. Despite being cleanly decomposed into a number of layers, KingsleyHeap performs nearly as well

| Memory Consumption for General-Purpose Allocators | | | | | |
|---|---|---|---|---|---|
| Benchmark | Kingsley | KingsleyHeap | KHeap + coal. | Lea | LeaHeap |
| *cfrac* | 270,336 | 280,640 | 271,944 | 208,896 | 241,272 |
| *espresso* | 974,848 | 992,032 | 541,696 | 462,848 | 448,808 |
| *lindsay* | 2,158,592 | 2,120,752 | 1,510,688 | 1,515,520 | 1,506,720 |
| *LRUsim* | 2,555,904 | 2,832,272 | 1,887,512 | 1,585,152 | 1,887,440 |
| *perl* | 425,984 | 454,024 | 342,344 | 331,776 | 337,408 |
| *roboop* | 45,056 | 20,760 | 11,440 | 20,480 | 11,616 |

Table 4.3: Memory consumption (in bytes) for the general-purpose allocators described in this paper.

as the original hand-coded Kingsley allocator. Runtime of LeaHeap is between $1/2$% faster and 20% slower than the Lea allocator (an average of 7% slower).

Figure 4.5(b) shows memory consumption for the same benchmarks normalized to the Lea allocator (we present the data used for this graph in Table 4.3). We define memory consumption as the high-water mark of memory requested from the operating system. For the Kingsley and Lea allocators, we used the amount reported by these programs; for the heap layers allocators, we directly measured the amount requested by both SbrkHeap and MmapHeap. KingsleyHeap's memory consumption is between 54% less and 11% more (on average 5.5% less), while LeaHeap's memory consumption is between 44% less and 19% more (on average 2% less) than the Lea allocator. The outlier is *roboop*, which has an extremely small footprint (just 16K) that exaggerates the memory efficiency of the heap layers allocators. Excluding *roboop*, the average increase in memory consumption for KingsleyHeap is 4% and for LeaHeap is 6.5%.

This investigation provides several insights. First, we have demonstrated that the heap layers framework is sufficiently robust that we can use it to develop quite sophisticated allocator implementations. Furthermore, we have shown that we can quickly (in a matter of weeks) assemble an allocator that is structurally similar to one of the best general-purpose allocators available. In addition, its performance and fragmentation are comparable to the original allocator.

## 4.5 Software Engineering Benefits

Our experience with building and using heap layers has been quite positive. Some of the software engineering advantages of using mixins to build software layers (e.g., heap layers) have been discussed previously,

especially focusing on ease of refinement [8, 19, 64]. We found that using heap layers as a means of stepwise refinement greatly simplified allocator construction. We also found the following additional benefits of using layers.

Because we can generally use any single layer to replace an allocator, we are often able to test and debug layers in isolation, making building allocators a much more reliable process. By adding and removing layers, we can find buggy layers by process of elimination. To further assist in layer debugging, we built a simple DebugHeap layer (shown in Figure 4.7) that checks for a variety of memory allocation errors, including invalid and multiple `free`s. During development, we insert this layer between pairs of layers as a sanity check. DebugHeap is also useful as a layer for finding errors in client applications. By using it with our heap layers allocators, we discovered a number of serious allocation errors (multiple `free`s) in *p2c*, a program we had previously planned to use as a benchmark.

The combination of error-checking in heap layers with compiler elimination of layer overhead encourages the division of allocators into many layers. When porting our first version of the LeaHeap to Solaris, we found that one of our layers, CoalesceSegHeap, contained a bug. This heap layer provided the functionality of SegHeap as well as coalescing, splitting and adding headers to allocated objects. This bug motivated us to break out coalescing and header management into different layers (CoalesceHeap and CoalesceableHeap). By interposing DebugHeap, we found the bug quickly.

## 4.6 Heap Layers as an Experimental Infrastructure

Because heap layers simplify the creation of memory allocators, we can use them to perform a wide range of memory allocation experiments that previously would have required a substantial programming effort. In this section, we describe one such experiment that demonstrates the use of heap layers as an experimental infrastructure.

As Figures 4.5(a) and 4.5(b) demonstrate, the Kingsley allocator is fast but suffers from excessive memory consumption. Wilson and Johnstone attribute this effect to the Kingsley allocator's lack of coalescing or splitting that precludes reuse of objects for different-sized requests [46]. A natural question is to what

extent adding coalescing remedies this problem and what impact it has on performance. Using heap layers, we just add coalescing and splitting with the layers we developed for LeaHeap.

We ran our benchmarks with this coalescing Kingsley heap and report runtime and performance numbers in the figures and tables as "KHeap + coal." Coalescing has a dramatic effect on memory consumption, bringing KingsleyHeap fairly close to the Lea allocator. Coalescing decreases memory consumption by an average of 50% (as little as 3% and as much as 80%). For most of the programs, the added cost of coalescing has little impact, but on the extremely allocation-intensive benchmarks (*cfrac* and *roboop*), this cost is significant. This experiment demonstrates that coalescing achieves effective memory utilization, even for an allocator with high internal fragmentation. It also shows that the performance impact of immediate coalescing is significant for allocation-intensive programs, in contrast to the Lea allocator which defers coalescing to certain circumstances, as described in Section 2.2.

## 4.7 Conclusion

In this chapter, we describe a framework in which custom and general purpose allocators can be effectively constructed from composable, reusable parts. Our framework, heap layers, uses C++ templates and inheritance to allow high-performance memory managers to be rapidly created. Even though heap layers introduce many layers of abstraction into an implementation, building allocators using heap layers can actually match or improve the performance of monolithic allocators. This non-intuitive result occurs, as we show, because heap layers expand the flexibility of compiler-directed inlining.

Based on our design, we implement a library of reusable heap layers: layers specifically designed to combine heaps, layers that provide heap utilities such as locking and debugging, and layers that support application-specific semantics such as region allocation and stack-structured allocation. We also demonstrate how these layers can be easily combined to create special and general purpose allocators.

To evaluate the cost of building allocators using heap layers, we present a performance comparison of two custom allocators found in SPEC2000 programs (197.parser and 176.gcc) against an equivalent implementation based on heap layers. In both cases, we show that the use of heap layers improves performance

slightly over the original implementation. This surprising result demonstrates the software engineering benefits described above have no performance penalty for these programs. We also compare the performance of a general-purpose allocator based on heap layers against the performance of the Lea allocator, widely considered to be among the best uniprocessor allocators available. While the allocator based on heap layers currently requires more CPU time (7% on average), we anticipate that this difference will shrink as we spend more time tuning our implementation. Furthermore, because our implementation is based on layers, we can easily provide an efficient scalable version of our allocator for multithreaded programs, whereas the Lea allocator requires significant effort to rewrite for this case.

Our results suggest a number of additional research directions. First, because heap layers are so easy to combine and compose, they provide an excellent infrastructure for doing comparative performance studies. Questions like the cache effect of size tags, or the locality effects of internal or external fragmentation can be studied easily using heap layers. Second, we anticipate growing our library of standard layers to increase the flexibility with which high-performing allocators can be composed. Finally, we believe that heap layers greatly simplify the creation of new general-purpose memory managers. In the remainder of this thesis, we use heap layers as a foundation for building these better general-purpose memory managers.

| A Library of Heap Layers | |
|---|---|
| **Top Heaps** | |
| mallocHeap | A thin layer over `malloc` |
| mmapHeap | A thin layer over the virtual memory manager |
| sbrkHeap | A thin layer over `sbrk` (contiguous memory) |
| **Building-Block Heaps** | |
| AdaptHeap | Adapts data structures for use as a heap |
| BoundedFreelistHeap | A freelist with a bound on length |
| ChunkHeap | Manages memory in chunks of a given size |
| CoalesceHeap | Performs coalescing and splitting |
| FreelistHeap | A freelist (caches freed objects) |
| **Combining Heaps** | |
| HybridHeap | Uses one heap for small objects and another for large objects |
| SegHeap | A general segregated fits allocator |
| StrictSegHeap | A strict segregated fits allocator |
| **Utility Layers** | |
| ANSIWrapper | Provides ANSI-malloc compliance |
| DebugHeap | Checks for a variety of allocation errors |
| LockedHeap | Code-locks a heap for thread safety |
| PerClassHeap | Use a heap as a per-class allocator |
| PHOThreadHeap | A private heaps with ownership allocator [11] |
| ProfileHeap | Collects and outputs fragmentation statistics |
| ThreadHeap | A pure private heaps allocator [11] |
| ThrowExceptionHeap | Throws an exception when the parent heap is out of memory |
| TraceHeap | Outputs a trace of allocations |
| UniqueHeap | A heap type that refers to one heap object |
| **Object Representation** | |
| CoalesceableHeap | Provides support for coalescing |
| SizeHeap | Records object sizes in a header |
| **Special-Purpose Heaps** | |
| ObstackHeap | A heap optimized for stack-like behavior and fast resizing |
| ZoneHeap | A zone (“region”) allocator |
| XallocHeap | A heap optimized for stack-like behavior |
| **General-Purpose Heaps** | |
| KingsleyHeap | Fast but high fragmentation |
| LeaHeap | Not quite as fast but low fragmentation |

Table 4.4: A library of heap layers, divided by category.

```
template <class SuperHeap>
class DebugHeap : public SuperHeap {
private:
  // A freed object has a special (invalid) size.
  enum { FREED = -1 };
  // "Error messages", used in asserts.
  enum { MALLOC_RETURNED_ALLOCATED_OBJECT = 0,
         FREE_CALLED_ON_INVALID_OBJECT = 0,
         FREE_CALLED_TWICE_ON_SAME_OBJECT = 0 };
public:
  inline void * malloc (size_t sz) {
    void * ptr = SuperHeap::malloc (sz);
    if (ptr == NULL)
      return NULL;
    // Fill the space with a known value.
    memset (ptr, 'A', sz);
    mapType::iterator i = allocated.find (ptr);
    if (i == allocated.end()) {
      allocated.insert (pair<void *, int>(ptr, sz));
    } else {
      if ((*i).second != FREED) {
        assert (MALLOC_RETURNED_ALLOCATED_OBJECT);
      } else {
       (*i).second = sz;
      }
    }
    return ptr;
  }
  inline void free (void * ptr) {
    mapType::iterator i = allocated.find (ptr);
    if (i == allocated.end()) {
      assert (FREE_CALLED_ON_INVALID_OBJECT);
      return;
    }
    if ((*i).second == FREED) {
      assert (FREE_CALLED_TWICE_ON_SAME_OBJECT);
      return;
    }
    // Fill the space with a known value.
    memset (ptr, 'F', (*i).second);
    (*i).second = FREED;
    SuperHeap::free (ptr);
  }
private:
  typedef map<void *, int> mapType;
  // A map of tuples (obj address, size).
  mapType allocated;
};
```

Figure 4.7: The implementation of DebugHeap.

```
template <int NumBins,
          int (*getSizeClass) (size_t),
          size_t (*getClassMaxSize) (int),
          class LittleHeap,
          class BigHeap>
class StrictSegHeap : public BigHeap {
public:
  inline void * malloc (size_t sz) {
    void * ptr;
    int sizeClass = getSizeClass (sz);
    if (sizeClass >= NumBins) {
      // This request was for a "big" object.
      ptr = BigHeap::malloc (sz);
    } else {
      size_t ssz = getClassMaxSize(sizeClass);
      ptr = myLittleHeap[sizeClass].malloc (ssz);
    }
    return ptr;
  }
  inline void free (void * ptr) {
    size_t objectSize = getSize(ptr);
    int objectSizeClass
      = getSizeClass (objectSize);
    if (objectSizeClass >= NumBins) {
      BigHeap::free (ptr);
    } else {
      while (getClassMaxSize(objectSizeClass)
             > objectSize) {
        objectSizeClass--;
      }
      myLittleHeap[objectSizeClass].free (ptr);
    }
  }
private:
 LittleHeap myLittleHeap[NumBins];
};
```

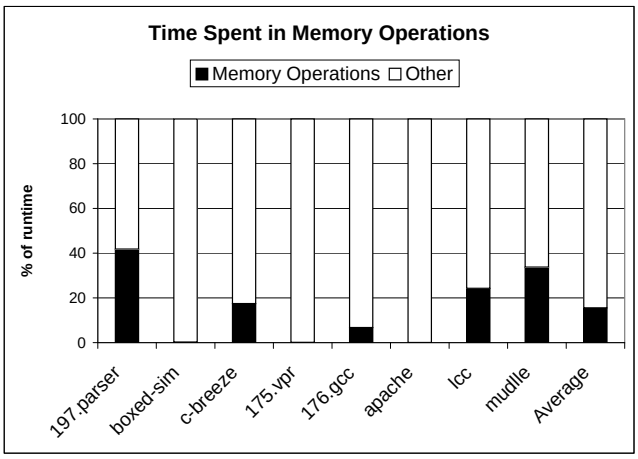
Figure 4.8: The implementation of StrictSegHeap.

# Chapter 5

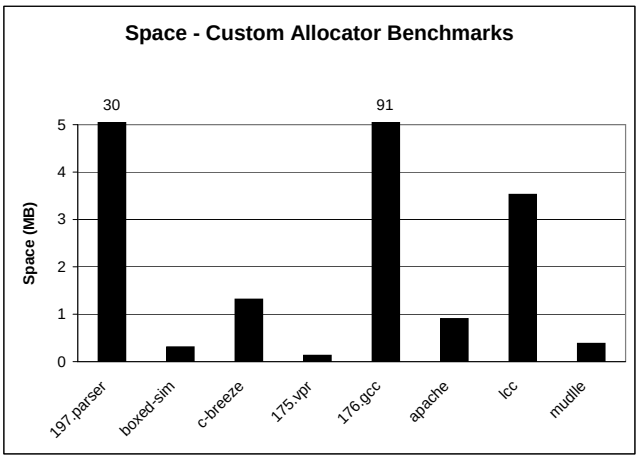
# Reconsidering Custom Memory Management

Programmers seeking to improve performance often incorporate custom memory managers into their applications. Custom memory managers aim to take advantage of application-specific patterns of memory usage to manage memory more efficiently than a general-purpose memory manager. For instance, the SPEC2000 benchmark 197.parser runs over 60% faster with its custom memory manager than with the Windows XP memory allocator [12]. Numerous books and articles recommend custom memory managers as an optimization technique [18, 54, 56]. The use of custom memory managers is widespread, including the Apache web server [2], the GCC compiler [28], three of the SPECint2000 benchmarks [65], and the C++ Standard Template Library [26, 63], all of which we examine here. The C++ language provides language constructs that directly support custom memory management (overloading `operator new` and `delete`) [23].

In this chapter, we perform a comprehensive evaluation of custom allocation. We survey a variety of applications that use a wide range of custom memory managers. We compare their performance and memory consumption to general-purpose memory managers. 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 memory manager (the Lea allocator [50]) yields performance equivalent to custom memory management for six of our eight benchmarks. These




(a) Time spent in memory operations for eight custom allocation benchmarks, with their memory managers replaced by the Windows allocator (see Section 5.1.1). Memory 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 memory managers*. Most of these consume relatively small amounts of memory on modern hardware, suggesting little opportunity for reducing memory consumption.

Figure 5.1: Runtime and space consumption for eight custom allocation benchmarks.

results suggest that most programmers seeking faster memory allocation should use the Lea allocator rather than writing their own custom memory manager.

The remainder of this chapter is organized as follows. We describe our benchmarks in Section 5.1. In Section 5.2, we analyze the structure of custom memory managers used by our benchmark applications. We describe our experimental infrastructure and methodology in Section 5.3 and present experimental results in Section 5.4. We discuss our results in Section 5.5, explaining why we believe programmers used custom memory managers despite the fact that these do not provide the performance they promise.

## 5.1 Benchmarks

We list the benchmarks we use in this paper in Table 5.1, including general-purpose allocation benchmarks that we use for comparison with custom allocation in Section 5.4.3. Most of our benchmarks come from the SPECint2000 benchmark suite [65]. 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) [29, 30], and boxed-sim, used by Chilimbi [20]. We also use the C-Breeze

compiler infrastructure [36]. C-Breeze makes intensive use of the C++ Standard Template Library (STL), and most implementations of the STL use custom memory managers, including the one we use in this study (STLport, officially recommended by IBM) [26, 63].

| **Benchmarks** | | |
|---|---|---|
| **custom allocation** | | |
| *197.parser* | English parser [65] | *test.in* |
| *boxed-sim* | Balls-in-box simulator [20] | *-n 3 -s 1* |
| *c-breeze* (C++) | C-to-C optimizing compiler [36] | *espresso.c* |
| *175.vpr* | FPGA placement & routing [65] | test placement |
| *176.gcc* | Optimizing C compiler [65] | *scilab.i* |
| *apache* | Web server [2] | see Section 5.3 |
| *lcc* | Retargetable C compiler [27] | *scilab.i* |
| *mudlle* | MUD compiler/interpreter [29] | *time.mud* |

Table 5.1: Benchmarks and inputs. Programs not written in C++ are written in C.

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 5.1 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, isolating performance from the usual network and disk I/O bottlenecks magnifies the performance impact of custom allocation. Second, using the same benchmark as Gay and Aiken facilitates comparison with their work.

### 5.1.1 Emulating Regions

Because custom memory managers often support semantics that differ from the C memory allocation interface, we need to emulate them with `malloc`/`free` as the underlying allocation mechanism. We wrote and tuned a region emulator to provide the full range of region semantics used by our benchmark applications, including nesting and obstacks (see Section 5.2.2). The region emulator uses the general-purpose memory

manager 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 this pointer information 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, which we measure in Section 5.4.3. 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 5.3.1.

## 5.2 Custom Memory Managers

In this section, we explain exactly what we mean by custom memory memory managers. We discuss the reasons why programmers use them and survey a wide range of custom memory managers, 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 memory manager may provide more than one object for every allocated chunk of memory obtained from the general-purpose memory manager. Second, it may not immediately return objects to the system or to the general-purpose memory manager.[1] For instance, a custom memory manager may obtain large chunks of memory from the general-purpose memory manager which it carves up into a number of objects. A custom memory manager might also defer object deallocation, returning objects to the system much later than when the object is last used or becomes unreachable.

[1]This definition of custom memory managers excludes, among others, wrappers that perform certain tests (e.g., for null return values) before returning objects obtained from the general-purpose memory manager.

### 5.2.1 Why Programmers Use Custom Memory Managers

There are a variety of reasons why programmers use custom memory managers. Runtime performance is the principal reason cited by programmers and authors of books on programming [18, 38, 53, 54, 56, 71]. Because the per-operation cost of most system general-purpose memory managers is an order of magnitude higher than that of custom memory managers, programs that make intensive use of the memory manager may see performance improvements by using custom memory managers.

**Improving performance.**

Figure 5.1(a) shows the amount of time spent in memory operations on eight applications using a wide range of custom memory managers, with the custom memory manager replaced by the Windows allocator[2]. Many of these applications spend a large percentage of their runtime in the memory manager (16% on average), demonstrating an opportunity to improve performance by optimizing memory memory management.

Nearly all of our benchmarks use custom memory managers 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 memory manager `ap_palloc` "is generally faster than malloc." The STLport implementation of STL (used in our runs of C-Breeze) refers to its custom memory manager as an "optimized node allocator engine", while 197.parser's memory manager 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"[3] and mudlle's region-based memory manager is "fast and easy". Because Hanson cites performance benefits for regions in his book [38], we assume that they intended the same benefit. Lcc also includes a per-class custom memory manager, intended to improve performance, which had no observable performance impact.[4] The per-class freelist-based custom memory manager for boxed-sim also appears intended to improve performance.

[2]For 176.gcc, Apache, lcc, and mudlle, we use a *region emulator* that matches the semantics of the custom memory manager (see Section 5.1.1).

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

[4]Hanson, 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 memory manager to isolate the impact of its region-based memory manager.

| Benchmark | *Motivation* perf. | space | s/w eng. | *Policy* same API | `region-Delete` | nested lifetimes | multiple areas | *Mechanism* chunks | stack optimized | same-type optimized |
|---|---|---|---|---|---|---|---|---|---|---|
| **custom pattern** | | | | | | | | | | |
| *197.parser* | ✓ | | | ✓ | | | | ✓ | ✓ | |
| **per-class** | | | | | | | | | | |
| *boxed-sim* | ✓ | | | ✓ | | | ✓ | | | ✓ |
| *c-breeze* (STL) | ✓ | | | ✓ | | | ✓ | | | ✓ |
| **region** | | | | | | | | | | |
| *175.vpr* | | ✓ | | | ✓ | | ✓ | ✓ | | |
| *176.gcc* (obstack) | ✓ | ✓ | ✓ | | ✓ | | ✓ | ✓ | ✓ | |
| *apache* (nested) | ✓ | | ✓ | | ✓ | ✓ | ✓ | ✓ | | |
| *lcc* | ✓ | | ✓ | | ✓ | | ✓ | ✓ | | |
| *mudlle* | ✓ | | ✓ | | ✓ | | ✓ | ✓ | | |

Table 5.2: Characteristics of the custom memory managers in our benchmarks. Performance motivates all but one of the custom memory managers, while only two were (possibly) motivated by space concerns (see Section 5.2.1). "Same API" means that the memory manager allows individual object allocation and deallocation, and "chunks" means the custom memory manager obtains large blocks of memory from the general-purpose memory manager for its own use (see Section 5.2.2).

**Reducing memory consumption.**

While programmers primarily use custom memory managers 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 memory manager "should be used for allocating fairly small data structures where memory-efficiency is crucial."[5] 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.[6] Figure 5.1(b) shows the amount of memory consumed by custom memory managers 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.

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

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

**Improving software engineering.**

Writing custom code to replace the general-purpose memory manager is generally not a good software engineering practice. Memory allocated via a custom memory manager cannot be managed later by another custom memory manager or 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 memory manager. Custom memory managers also can make it difficult to understand the sources of memory consumption in a program. Using custom memory managers often precludes the use of memory leak detection tools like Purify [39].

However, custom memory managers can provide some important software engineering benefits. The use of region-based custom memory managers in parsers and compilers (e.g., 176.gcc, lcc, and mudlle) simplifies memory management [38]. Regions provide separate memory areas which a single call deletes in its entirety. Multithreaded server applications use regions to isolate the memory spaces of separate threads (*sandboxing*), reducing the likelihood that one thread will accidentally overwrite another thread's data. Server applications like the Apache web server also use regions to prevent memory leaks, tearing down all memory associated with a terminated 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 5.4.3.

### 5.2.2 A Taxonomy of Custom Memory Managers

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

We break down the memory managers from our custom allocation benchmarks in terms of several

characteristics in Table 5.2. We divide these into three categories: the *motivation* behind the programmer's use of a custom memory manager, the *policies* they implement, 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 memory managers 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 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. Allocation and freeing are thus as fast as possible. A region-based memory manager 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.

**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.

**obstack region** An *obstack* is an extended version of a region-based memory manager that adds deletion of every object allocated after a certain object [80]. This extension supports object allocation that follows a stack discipline (hence the name, which comes from "object stack").

**custom pattern** This catch-all category refers to what is essentially a general-purpose memory manager 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

| Benchmark Statistics | | | | | | |
|---|---|---|---|---|---|---|
| Benchmark | Total objects | Max objects in use | Avg obj. size (in bytes) | Total memory (in bytes) | Max in use (in bytes) | Mem. operations (% of runtime) |
| **custom allocation** | | | | | | |
| *197.parser* | 9,334,022 | 230,919 | 38 | 351,772,626 | 3,207,529 | 41.8% |
| *boxed-sim* | 52,203 | 4,865 | 15 | 777,913 | 301,987 | 0.2% |
| *c-breeze* | 5,090,805 | 2,177,173 | 23 | 118,996,917 | 60,053,789 | 17.4% |
| *175.vpr* | 3,897 | 3,813 | 44 | 172,967 | 124,636 | 0.1% |
| *176.gcc* | 9,065,285 | 2,538,005 | 54 | 487,711,209 | 112,753,774 | 6.7% |
| *apache* | 149,275 | 3,749 | 208 | 30,999,123 | 754,492 | 0.1% |
| *lcc* | 1,465,416 | 92,696 | 57 | 83,217,416 | 3,875,780 | 24.2% |
| *mudlle* | 1,687,079 | 38,645 | 29 | 48,699,895 | 662,964 | 33.7% |

Table 5.3: Statistics for our custom allocation benchmarks, replacing custom memory allocation by general-purpose allocation. We compute the runtime percentage of memory operations with the default Windows allocator.

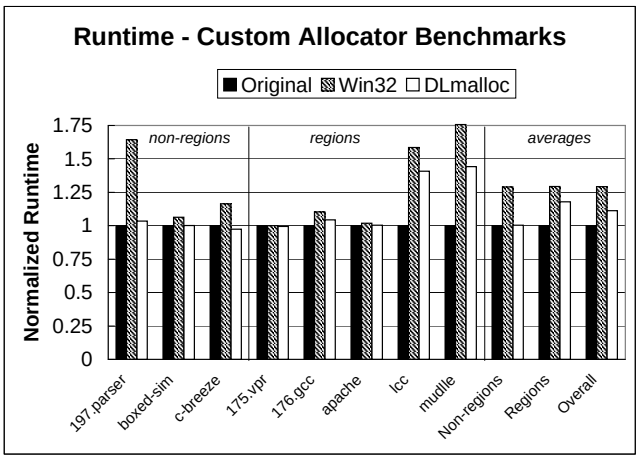
to the new last block in use. This allocator is fast for 197.parser's stack-like use of memory, but if object lifetimes do not follow a stack-like discipline, it exhibits unbounded memory consumption.

## 5.3 Evaluating Custom Memory Managers

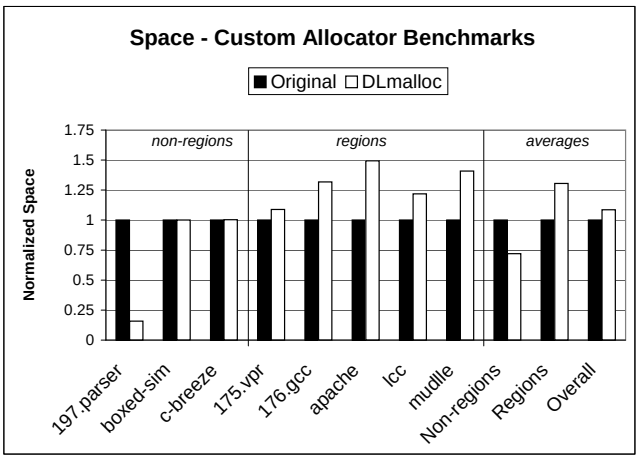
We provide allocation statistics for our benchmarks in Tables 3.4 and 5.3. Many of the general-purpose allocation benchmarks are not allocation intensive, but we include them for completeness. In particular, 181.mcf, 186.crafty, 252.eon and 254.gap allocate only a few objects over their entire lifetime, including one or more very large objects. Certain trends appear from the data. In general, programs using general-purpose memory managers spend relatively little time in the memory manager (on average, around 3%), while programs using custom memory managers spend on average 16% of their time in memory operations. Programs using custom memory managers also tend to allocate many small objects. This kind of allocation behavior stresses the memory manager.

### 5.3.1 Evaluating Regions

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




(a) Normalized runtimes (smaller is better). Custom memory managers often outperform the Windows allocator, but the Lea allocator is as fast as or faster than most of the custom memory managers.




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

Figure 5.2: Normalized runtime and memory consumption for our custom allocation benchmarks, comparing the original custom memory managers to the Windows and Lea allocators.

one live object ties down an entire region, potentially leading to a considerable amount of wasted memory. We want to explore the impact on memory consumption of this inability to reclaim dead objects.

We do not undertake the rewriting of region-based programs like lcc or Apache (60K – 100K lines of code) to use explicit object deallocation, which requires considerable application expertise and is very time-intensive. Instead, we measure the impact of using regions by using a binary instrumentation tool we wrote using the Vulcan binary instrumentation system [72]. We link the programs 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 is not unrealistic, as we show below with measurements of programs using general-purpose memory managers.

## 5.4 Results

In this section, we present our experimental results on runtime and memory consumption, discussing the programmers' goals for their custom memory managers and whether they were met. All runtimes are the best of three runs at real-time priority after one warm-up run; variation was less than one percent. We executed these programs on PC Platform 2 (see 3.2). We compare the custom memory managers to the Windows XP allocator, which we refer to in the graphs as "Win32", to version 2.7.0 of Doug Lea's memory manager, which we refer to as "DLmalloc."

### 5.4.1 Runtime Performance

To compare runtime performance of custom allocation to general-purpose allocation, we simply reroute custom memory manager calls to the general-purpose memory manager, using region emulation when needed. For this study, we compare custom memory managers to the Windows XP allocator, and version 2.7.0 of the Lea allocator.

In Figure 5.2(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 memory manager. 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 memory managers for most of these programs.

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

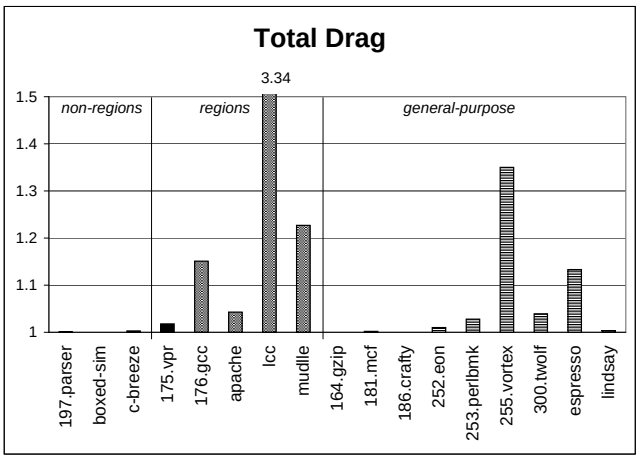
### 5.4.2 Memory Consumption

We measured the memory consumed by the various memory managers by running the benchmarks 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 an equivalent way to keep track of memory consumption.

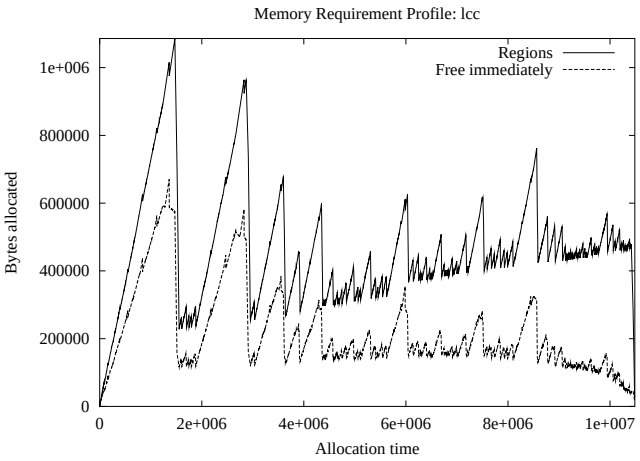
Figure 5.2(b) shows our results for memory consumption, which are quite mixed. Neither custom memory managers nor the Lea allocator consistently yield a space advantage. 176.gcc allocates many small objects, so the per-object overhead of the Lea allocator (8 bytes) lead to increased memory consumption. Despite its overhead, the Lea allocator often *reduces* memory consumption, as in 197.parser, c-breeze and Apache. The custom memory manager in 197.parser allocates from a fixed-sized chunk of memory (a compile-time constant, set at 30MB), while the Lea allocator uses just 15% of this memory. Worse, this custom memory manager is brittle; requests beyond the fixed limit result in program termination. Apache's region allocator is less space-efficient than our region emulator, accounting for the difference in space consumption.

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, which is consistent with our observation that programmers generally do not use custom allocation to reduce memory consumption.

Our results show that most custom memory managers achieve neither performance nor space advantages. However, region-based allocators can provide both advantages (see lcc and mudlle). These space advantages are somewhat misleading. While the Lea allocator adds 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.




(a) Drag statistics for applications using general-purpose memory allocation (average 1.1), non-regions (average 1.0) and region custom memory managers (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 5.3: 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.

### 5.4.3 Evaluating Region Allocation

Using the binary instrumentation tool we describe in Section 5.3.1, we obtained two curves over allocation time [46] 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 farther the program's memory consumption is from ideal.

Figure 5.3(a) shows drag statistics for a wide range of benchmarks, including programs using general-purpose memory managers. Programs using non-region custom memory managers 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 either 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 5.4 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 5.3(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.

| **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% |
| **Average** | | | 23% |

Table 5.4: Peak memory (footprint) for region-based applications, in bytes. Using regions leads to an increase in footprint from 6% to 63% (average 23%).

## 5.5 Discussion

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

**Recommended practice.**

One reason that we believe programmers use custom memory managers to improve performance is because it is recommended by so many influential practitioners and because of the perceived inadequacies of system-provided memory managers. 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 memory manager 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 memory managers 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 managers 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 a high-quality general-purpose memory manager like the Lea allocator rather than writing a custom memory manager.

## 5.6 Conclusions

Despite the widespread belief that custom memory managers should be used in order to improve performance, we come to a different conclusion. In this chapter, we examine eight benchmarks using custom

memory managers, including the Apache web server and several applications from the SPECint2000 benchmark suite. We find that the Lea memory manager is as fast as or even faster than most custom memory managers. The exceptions are region-based memory managers, which often outperform general-purpose memory management.

The results in this chapter indicate that, for many applications, a good general-purpose memory manager can provide excellent performance. However, programmers use region-based memory managers to achieve both performance and software engineering benefits. We show in the next chapter how to capture the benefits of both regions and general-purpose memory management in a hybrid memory manager called *reap* that is especially well-suited for certain types of server applications, including Apache. In Chapter 7, we show that current general-purpose memory managers do not provide satisfactory performance for applications running on multiprocessors, and present our solution.

# Chapter 6

# Memory Management for Servers

In the previous chapter, we describe region-based custom allocators that some applications use to improve performance. However, server applications (e.g., Apache) use regions because they need additional memory management support beyond that provided by the general-purpose memory manager. These applications require *sandboxing*, or isolating the memory spaces of separate threads, in order to reduce the likelihood of one thread accidentally or maliciously overwriting another thread's data. Second, and often more importantly, server applications need support for connection (or transaction) *teardown*. When a connection is terminated or fails, the server must be able to tear down all memory associated with the connection. By associating separate regions with every connection or transaction, the programmer can achieve both sandboxing and rapid teardown. In addition, regions can also provide higher performance than general-purpose memory managers. However, regions force the programmer to retain all memory associated with a region until the last object in the region dies [29, 30, 37, 61, 73]. Beyond the causing drag (see Section 5.4.3), this limitation has serious software engineering implications for server applications, which we describe in detail below.

The rest of this chapter is organized as follows. First, we discuss the drawbacks of regions. We then present a generalization of regions and heaps we call *reaps*. We show that our implementation of reaps provides the performance and semantics of regions while allowing programmers to delete individual objects. We do not undertake the addition of individual object deletion calls to existing region-based programs

because it requires both application expertise and a considerable investment of time. However, we show that reaps nearly match the speed of regions when used in the same way, and provide additional semantics and generality. We argue that reaps provide a reusable library solution for region allocation with competitive performance, the potential for reduced memory consumption, and greater flexibility than regions.

## 6.1 Drawbacks of Regions

In Section 5.4.3, 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, however, 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. For instance, using regions is not possible for many programs using `malloc` and `free`, producer-consumer allocation patterns, or dynamic arrays because the inability to free individual objects in regions could lead to unbounded memory consumption. 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 adoption of regions to satisfy Apache's needs of sandboxing, heap teardown, and high performance.

## 6.2 Desiderata

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 survey in

Chapter 5 (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.

## 6.3 Reaps: Generalizing Regions and Heaps

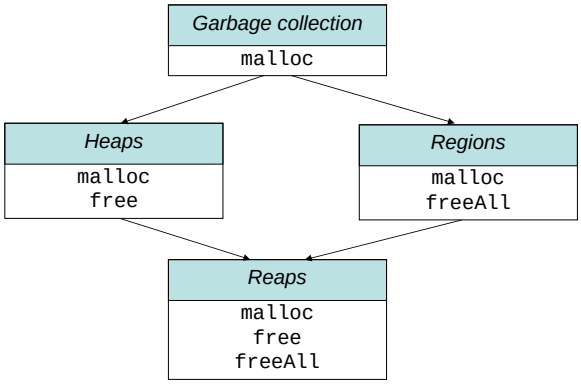
We have designed and implemented a generalization of regions and general-purpose memory allocators (heaps) that we call *reaps*. Reaps provide a full range of region semantics, including nested regions, but also include individual object deletion. Figure 6.1(a) depicts a lattice of API's, showing how reaps combine the semantics of regions and heaps. We provide a C-based interface to reap allocation, including operations for reap creation and destruction, deletion (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 reapDelete (void ** reap);
void reapDestroy (void ** reap);
void * reapAllocate (void ** reap, size_t size);
void reapFree (void ** reap, void * object);
```

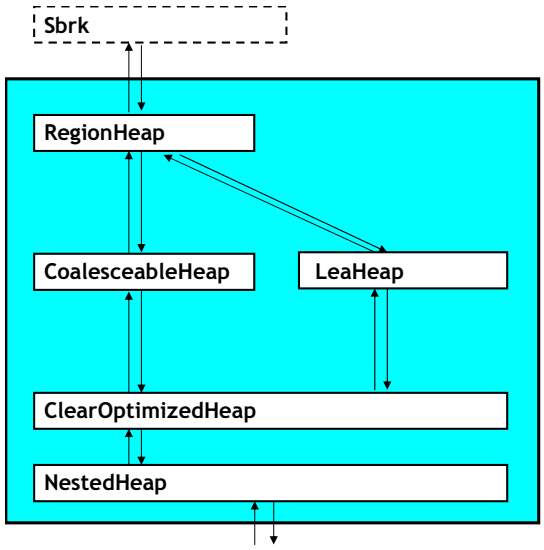
### 6.3.1 Design and Implementation

Our implementation of reaps, which we built using heap layers, 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 allocate memory like regions, bumping a pointer through geometrically increasing large chunks of memory (initially 8K), which are threaded into a doubly-linked list. Unlike regions, however, we 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 we revert to region mode.

Figure 6.1(b) depicts the design of reaps in graphical form, using Heap Layers. Memory requests




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

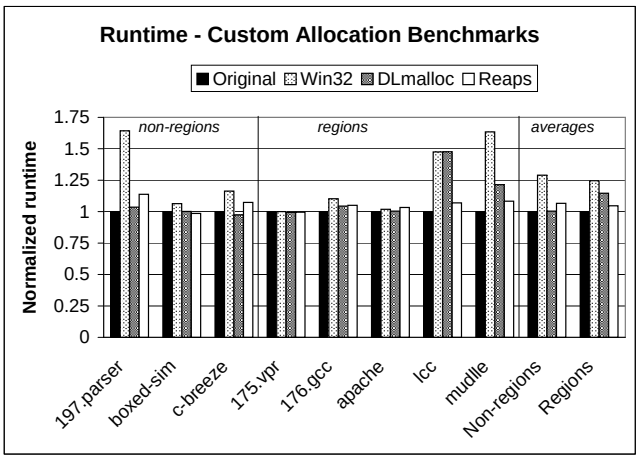



(b) A diagram of the heap layers that comprise our implementation of reaps. Reaps adapt to their use, acting either like regions or heaps (see Section 6.3).

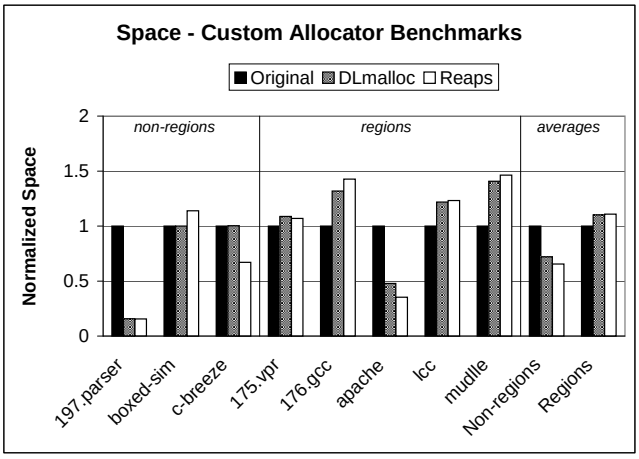
Figure 6.1: A description of the API and implementation of reaps.

(`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 argument, bumping a pointer and adding per-object metadata. When an object is freed, `nothingOnHeap` 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 set to true. We use ClearOptimizedHeap to obtain memory directly from the system via CoalesceableHeap, which adds the necessary header information so we can later free this memory (and coalesce adjacent free objects). Bypassing the LeaHeap for this case has little impact on general-purpose memory allocation,




(a) Normalized runtimes (smaller is better). Reaps are almost as fast as or faster than most of the custom memory managers. In particular, reaps nearly match the performance of region-based custom memory managers.




(b) Normalized space (smaller is better). We omit the Windows allocator because we cannot directly measure its space consumption. Reaps generally consume less memory than non-region custom memory managers and more than region-based memory managers.

Figure 6.2: Normalized runtime and memory consumption for our custom allocation benchmarks, comparing the original allocators to the Windows and Lea allocators and to reaps.

speeding up only the initial allocation of heap items, but it 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. We use the RegionHeap layer to manage memory in geometrically-increasing chunks of at least 8K, making `reapDelete` efficient.

## 6.4 Results

In this section, we present our experimental results on runtime and memory consumption for reaps. 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 reaps 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.”

### 6.4.1 Runtime Performance

As in Section 5.4.1, we compare runtime performance of allocators simply by rerouting custom memory manager calls to reaps, using region emulation when needed. For this study, we compare reaps to the Windows XP memory manager and to version 2.7.0 of the Lea allocator. 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.

The fourth bar in Figure 5.2(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.

### 6.4.2 Memory Consumption

As in Chapter 5, 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 because it does not provide an equivalent way to keep track of memory consumption.

Figure 5.2(b) shows our results for memory consumption. On average, reaps consume less memory than non-region custom memory managers and somewhat more than region-based memory managers. The per-object overhead of reaps (8 bytes) leads to increased memory consumption in applications that allocate many small objects, like 176.gcc. Despite this overhead, reaps often *reduce* memory consumption, as in 197.parser, c-breeze and Apache. On the other hand, our use of geometrically-increasing chunk sizes in reaps causes increased memory consumption for mudlle.

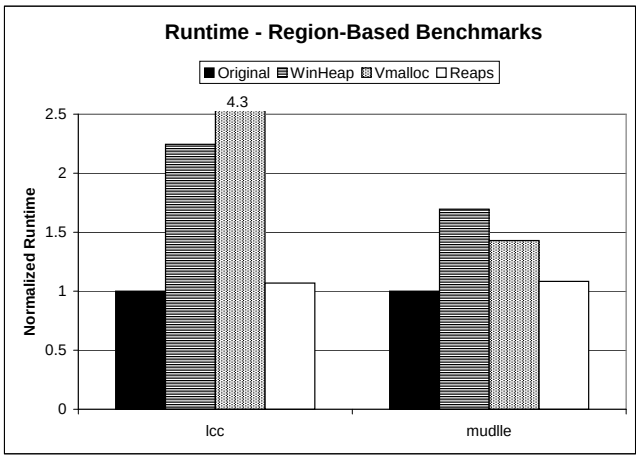



Figure 6.3: Normalized runtimes (smaller is better). Reaps are almost as fast as the original custom allocators and much faster than previous allocators with similar semantics.

### 6.4.3 Experimental Comparison to Previous Work

In Figure 6.3, we present results comparing reaps to the previous allocators that provide similar semantics (see Section 2.3). 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 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.

## 6.5 Conclusion

In this chapter, we show that regions can come at an increased cost in memory consumption and do not support common programming idioms. With our implementation of reaps, we demonstrate a memory allocator that provides region performance and extended region semantics. 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. However, reaps are not a panacea. In particular, reaps do not address the particular needs of applications running on multiprocessors, which we discuss in the next chapter.

# Chapter 7

# Scalable Concurrent Memory Management

While the general-purpose and custom allocators we have described so far are suitable for single-threaded applications, they do not provide effective support for multithreaded applications running on multiprocessors. In this chapter, we discuss general-purpose memory allocation for multithreaded applications, describe problems with existing memory allocators and present Hoard, a fast, scalable allocator that largely avoids false sharing and is memory efficient.

Parallel, multithreaded programs are becoming increasingly prevalent. These applications include web servers, database managers, news servers, as well as more traditional parallel applications such as scientific applications. For these applications, high performance is critical. They are generally written in C or C++ to run efficiently on modern shared-memory multiprocessor servers. Many of these applications make intensive use of dynamic memory allocation. Unfortunately, the memory allocator is often a bottleneck that severely limits program scalability on multiprocessor systems [11, 48].

Existing allocators suffer from problems that include poor performance and scalability, and heap organizations that introduce false sharing. Worse, many allocators exhibit a dramatic increase in memory consumption when confronted with a producer-consumer pattern of object allocation and freeing. This increase in memory consumption can range from a factor of $P$ (the number of processors) to unbounded memory consumption. These problems combine and often result in allocators that prevent applications from scaling on multiprocessors. For instance, British Telecom reports that for a proprietary middleware appli-

cation, increasing the number of CPUs in their server from 1 to 6 reduced throughput from 500 orders per hour to 300 orders per hour. Replacing the default Solaris memory allocator with Hoard raised throughput to over 1,600 orders per hour [79].

In order to achieve scalable and memory-efficient memory allocator performance, all of the following features are required:

**Speed.** A memory allocator should perform memory operations (i.e., `malloc` and `free`) about as fast as a state-of-the-art serial memory allocator. This feature guarantees good allocator performance even when a multithreaded program executes on a single processor.

**Scalability.** As the number of processors in the system grows, the performance of the allocator must scale linearly with the number of processors to ensure scalable application performance.

**False sharing avoidance.** The allocator should not introduce false sharing of cache lines in which threads on distinct processors inadvertently share data on the same cache line.

**Low fragmentation.** We define *fragmentation* as the maximum amount of memory allocated from the operating system divided by the maximum amount of memory required by the application. Excessive fragmentation can degrade performance by causing poor data locality, leading to paging.

Certain classes of memory allocators (described in Section 7.2) exhibit a special kind of fragmentation that we call ***blowup***. Intuitively, blowup is the increase in memory consumption caused when a concurrent allocator reclaims memory freed by the program but fails to use it to satisfy future memory requests. We define blowup as the maximum amount of memory allocated by a given allocator divided by the maximum amount of memory allocated by an ideal uniprocessor allocator. As we show in Section 7.1.2, the common producer-consumer programming idiom can cause blowup. In many allocators, blowup ranges from a factor of $P$ (the number of processors) to unbounded memory consumption (the longer the program runs, the more memory it consumes). Such a pathological increase in memory consumption can be catastrophic, resulting in premature application termination due to exhaustion of swap space.

We have developed an allocator called Hoard that enables parallel multithreaded programs to achieve scalable performance on shared-memory multiprocessors [11]. Hoard achieves this result by simultaneously solving all of the above problems. In particular, Hoard solves the blowup and false sharing problems, which, as far as we know, have never been addressed in the literature. As we demonstrate, Hoard also achieves nearly zero synchronization costs in practice.

Hoard maintains per-processor heaps and one global heap. When a per-processor heap's usage drops below a certain fraction, Hoard transfers a large fixed-size chunk of its memory from the per-processor heap to the global heap, where it is then available for reuse by another processor. We show that this algorithm bounds blowup and synchronization costs to a constant factor. This algorithm avoids false sharing by making it difficult for processors to allocate from the same cache line. Results on eleven programs demonstrate that Hoard scales linearly as the number of processors grows and that its fragmentation costs are low. On 14 processors, Hoard improves performance over the standard Solaris allocator by up to a factor of 60 and a factor of 18 over the next best allocator we tested. These features have led to its incorporation in a number of high-performance commercial applications, including the Twister, Typhoon, Breeze and Cyclone chat and USENET servers [9] and BEMSolver, a high-performance scientific code [21].

The remainder of this chapter is organized as follows. We describe the false sharing and blowup problems in previous work in Section 7.1. In Section 7.2, we classify previous work into a taxonomy of memory allocators, focusing on speed, scalability, false sharing, and fragmentation. We describe the algorithms used in the Hoard allocator in Section 7.4, provide a summary of analytical results in Section 7.5, and we demonstrate Hoard's scalable performance empirically in Section 7.8.

## 7.1 Motivation

In this section, we focus special attention on the issues of allocator-induced false sharing of heap objects and blowup to motivate our work. As we show in Section 7.8, these issues must be addressed to achieve efficient memory allocation for scalable multithreaded applications but have been neglected in the memory allocation literature.

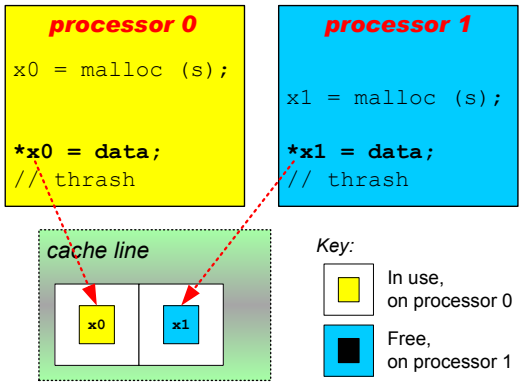



Figure 7.1: An example of allocator-induced false sharing of heap objects. The boxes correspond to allocated objects: the inside color reflects the allocating processor, and the outside color reflects the processor on which the freed object resides. Here the allocator parceled out one cache line to two processors (*actively-induced* false sharing), resulting in cache thrashing.

### 7.1.1 Allocator-Induced False Sharing of Heap Objects

***False sharing*** occurs when multiple processors share words in the same cache line without actually sharing data and is a notorious cause of poor performance in parallel applications [42, 47, 74]. Allocators can cause false sharing of heap objects by dividing cache lines into a number of small objects that distinct processors then write. A program may introduce false sharing by allocating a number of objects within one cache line and passing an object to a different thread. It is thus impossible to completely avoid false sharing of heap objects unless the allocator pads out every memory request to the size of a cache line. However, no user-level allocator we know of pads memory requests to the size of a cache line, and with good reason; padding could cause a dramatic increase in memory consumption (for instance, objects would be padded to a multiple of 64 bytes on a SPARC) and thus significantly degrade spatial locality and cache utilization.

Unfortunately, an allocator can ***actively induce*** false sharing even on objects that the program does not pass to different threads. Active false sharing is due to `malloc` satisfying memory requests by different threads from the same cache line. For instance, single-heap allocators can give many threads parts of the same cache line. Figure 7.1 demonstrates this splitting of cache lines, leading to false sharing. Here, the allocator divides a cache line into 8-byte chunks. The allocator gives each processor one chunk in turn, generating false sharing because both are on the same cache line.

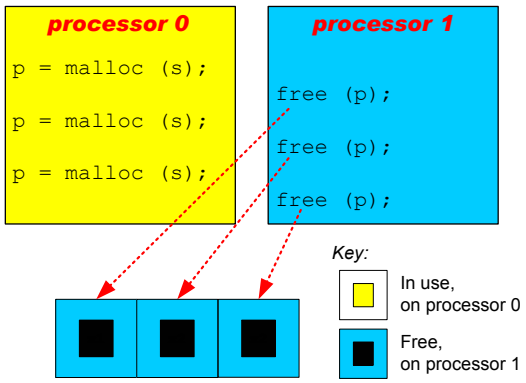



Figure 7.2: This figure demonstrates how *pure private heaps* allocators can exhibit unbounded memory consumption. Processor 0 allocates objects that processor 1 frees. However, processor 0 cannot reclaim the memory on processor 1, and so $s$ bytes "leak" on every iteration.

Allocators may also ***passively induce*** false sharing. Passive false sharing occurs when `free` allows a future `malloc` to produce false sharing. If a *program* introduces false sharing by spreading the pieces of a cache line across processors, the allocator may then passively induce false sharing after a `free` by letting each processor reuse pieces it freed, which then leads to false sharing.

### 7.1.2 Blowup

Many previous allocators suffer from blowup. As we show in Section 7.4.1, Hoard keeps blowup to a constant factor. To the best of our knowledge, papers in the literature do not address this problem, and many existing concurrent allocators suffer from a blowup problem. The worst of these is the ***pure private heaps*** algorithm, used by the Cilk and STL allocators [15, 63]. This memory manager reserves one heap for each processor: all memory allocations and frees are performed on the local heap. Except when objects are initially allocated, this approach eliminates heap contention.

Unfortunately, the pure private heaps algorithm can exhibit *unbounded* blowup: memory consumption can grow without bound, even though the memory required is fixed. Figure 7.2 shows how this blowup can occur. In this example, two processors are in a producer-consumer relationship. The producer thread allocates a block of memory and gives it to the consumer thread, which frees it. Using a pure private heaps allocator, the memory freed by the consumer is unavailable to the producer, so the program consumes more

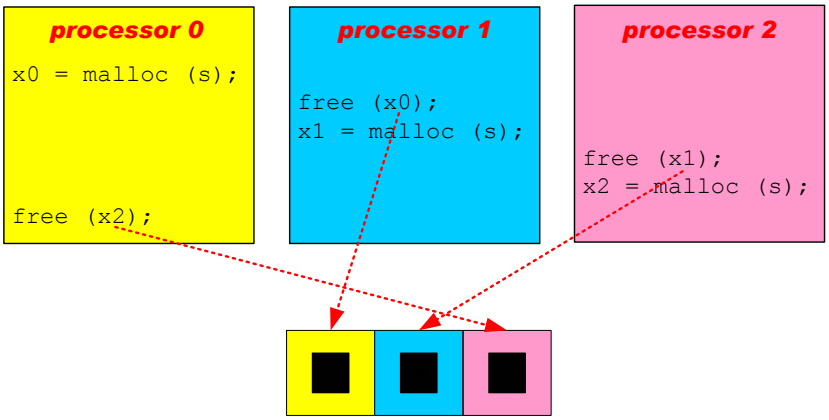



Figure 7.3: This figure demonstrates how *private heaps with ownership* allocators can exhibit a $P$-fold blowup in memory consumption, where a round-robin producer-consumer pattern spreads memory across the processors.

and more memory as it runs.

Other concurrent memory allocators suffer from a less dramatic but still serious blowup problem. ***Private heaps with ownership*** allocators return memory to the originating processor (e.g., Ptmalloc and LKmalloc [31, 49]). This approach avoids the unbounded blowup of pure private heaps allocators, but we show that it can cause memory consumption to grow linearly with $P$, the number of processors. Figure 7.3 demonstrates how such blowup can occur. Here the processors are in a round-robin producer-consumer relationship (processor $i$ mod $P$ allocates, processor $(i + 1)$ mod $P$ frees). The program requires only $s$ blocks, but the memory manager will allocate $P * s$ blocks ($s$ on all $P$ heaps) because the ownership policy makes memory systematically unavailable for reuse.

This $P$-fold increase in memory consumption is a cause for concern. On 32-bit architectures, multiplying memory consumption by a factor of $P$ can cause many programs to exhaust all available address space. A program that uses more than 128MB of memory could not run on a 16-processor machine. Further, the scheduling of multithreaded programs on multiple processors can cause these programs to require *much* more memory when run on one processor [15, 57]. Consider a program with $P$ threads. Each thread calls `x=malloc(s); free(x)`. If these threads are serialized, the total memory required is $s$. However, if they execute on $P$ processors and each call to `malloc` runs in parallel, the memory requirement increases to $P * s$. If the allocator multiplies this consumption by another factor of $P$, then memory consumption

increases to $P^2 * s$.

## 7.2 Related Work

While dynamic storage allocation is one of the most studied topics in computer science, there has been relatively little work on concurrent memory allocators. In this section, we place past work into a taxonomy of memory allocator algorithms. We address the blowup and allocator-induced false sharing characteristics of each of these algorithms and compare them to Hoard.

## 7.3 Taxonomy of Memory Allocator Algorithms

Our taxonomy consists of the following five categories:

**Serial single heap.** Only one processor may access the heap at a time (Solaris, Windows NT/2000 [48]).

**Concurrent single heap.** Many processors may simultaneously operate on one shared heap ([14, 43, 44, 40, 41]).

**Pure private heaps.** Each processor has its own heap (STL [63], Cilk [15]).

**Private heaps with ownership.** Each processor has its own heap, but memory is always returned to its "owner" processor (*MTmalloc*, *Ptmalloc* [31], *LKmalloc* [49]).

**Private heaps with thresholds.** Each processor has its own heap which can hold a limited amount of free memory (DYNIX kernel allocator [52], Vee and Hsu [76]).

Below we discuss these single and multiple-heap algorithms, focusing on the false sharing and blowup characteristics of each.

### 7.3.1 Single Heap Allocation

***Serial single heap*** allocators often exhibit extremely low fragmentation over a wide range of real programs [46] and are quite fast on uniprocessors [50]. Since they typically protect the heap with a single lock

| Allocator algorithm | fast? | scalable? | avoids false sharing? | avoids blowup? |
|---|---|---|---|---|
| serial single heap | ✓ | | | ✓ |
| concurrent single heap | | maybe | | ✓ |
| pure private heaps | ✓ | ✓ | | unbounded |
| private heaps w/ownership: | | | | |
| *Ptmalloc* [31] | ✓ | ✓ | | $O(P)$ |
| *MTmalloc* | ✓ | | | $O(P)$ |
| *LKmalloc* [49] | ✓ | ✓ | ✓ | $O(P)$ |
| private heaps w/thresholds | ✓ | ✓ | | ✓ |
| **Hoard** | ✓ | ✓ | ✓ | ✓ |

Table 7.1: A taxonomy of memory allocation algorithms discussed in this chapter.

which serializes memory operations and introduces contention, they are inappropriate for use with most parallel multithreaded programs. In multithreaded programs, contention for the lock prevents allocator performance from scaling with the number of processors. Many modern operating systems provide such memory allocators in the default library, including Solaris and IRIX. Windows NT/2000/XP uses 64-bit atomic operations on freelists rather than locks [48] which is also unscalable because the head of each freelist is a central bottleneck[1]. These allocators all actively induce false sharing.

***Concurrent single heap*** allocation implements the heap as a concurrent data structure, such as a concurrent B-tree [32, 33, 40, 41, 43, 44] or a freelist with locks on each free block [14, 22, 70]. This approach reduces to a serial single heap in the common case when most allocations are from a small number of object sizes. Johnstone and Wilson show that for every program they examined, the vast majority of objects allocated are of only a few sizes [45]. Each memory operation on these structures requires either time linear in the number of free blocks or $O(\log C)$ time, where $C$ is the number of ***size classes*** of allocated objects. A size class is a range of object sizes that are grouped together (e.g., all objects between 32 and 36 bytes are treated as 36-byte objects). Like serial single heaps, these allocators actively induce false sharing. Another problem with these allocators is that they make use of many locks or atomic update operations (e.g., `compare-and-swap`), which are quite expensive on modern architectures.

State-of-the-art serial allocators are so well engineered that most memory operations involve only

[1]The Windows allocator and some of Iyengar's allocators use one freelist for each object size or range of sizes [40, 41, 48]

a handful of instructions [50]. An *uncontended* lock acquisition and release accounts for about half of the total runtime of these memory operations. In order to be competitive, a memory allocator can only acquire and release at most two locks in the common case, or incur three atomic operations. Hoard requires only one lock for each `malloc` and two for each `free`.

### 7.3.2 Multiple Heap Allocation

In this section, we discuss multiple-heap allocators as if heaps were directly associated with processors. However, because operating systems are generally free to switch processors at any time, user-space memory allocators cannot guarantee a one-to-one connection between heaps and executing processors.

Multiple heap allocators therefore use a variety of techniques to map threads onto heaps. These techniques include assigning one heap for each thread using thread-specific data [63], by using a currently unused heap from a collection of heaps [31], round-robin heap assignment (as in *MTmalloc*, provided with Solaris 7 as a replacement allocator for multithreaded applications), or by providing a mapping function that maps threads onto a collection of heaps (*LKmalloc* [49], Hoard). For simplicity of exposition in the remainder of the thesis, we assume that there is exactly one thread bound to each processor and one heap for each of these threads. We describe Hoard's mapping strategy in Section 7.4.

We group existing multiple-heap allocators into three categories, which we describe in detail below: *pure private heaps*, *private heaps with ownership*, and *private heaps with thresholds*. STL's (Standard Template Library) *pthread_alloc*, Cilk 4.1, and many ad hoc allocators use ***pure private heaps*** allocation [15, 63]. Each processor has its own per-processor heap that it uses for every memory operation (the allocator `malloc`s from its heap and `free`s to its heap). Each per-processor heap is "purely private" because each processor never accesses any other heap for any memory operation. After one thread allocates an object, a second thread can free it; in pure private heaps allocators, this memory is placed in the second thread's heap. Since parts of the same cache line may be placed on multiple heaps, pure private-heaps allocators passively induce false sharing. Worse, pure private-heaps allocators exhibit unbounded memory consumption given a producer-consumer allocation pattern, as described in Section 7.1.2. Hoard avoids this problem by returning freed blocks to the heap that owns them.

***Private heaps with ownership*** allocators return free blocks to the heap that allocated them. This algorithm, used by *MTmalloc*, *Ptmalloc* [31] and *LKmalloc* [49], yields $O(P)$ blowup, whereas Hoard has $O(1)$ blowup. *Ptmalloc* and *MTmalloc* can actively induce false sharing (different threads may allocate from the same heap). *LKmalloc*'s permanent assignment of large regions of memory to processors and its immediate return of freed blocks to these regions, while leading to $O(P)$ blowup, should have the advantage of eliminating allocator-induced false sharing, although the authors did not explicitly address this issue. Hoard explicitly takes steps to reduce false sharing, while maintaining $O(1)$ blowup.

Both *Ptmalloc* and *MTmalloc* also suffer from scalability bottlenecks. In *Ptmalloc*, each `malloc` chooses the first heap that is not currently in use (caching the resulting choice for the next attempt). This heap selection strategy causes substantial bus traffic which limits *Ptmalloc*'s scalability to about 6 processors, as we show in Section 7.8. *MTmalloc* performs round-robin heap assignment by maintaining a "nextHeap" global variable that is updated by every call to `malloc`. This variable is a source of contention that makes *MTmalloc* unscalable and actively induces false sharing. Hoard has no centralized bottlenecks except for the global heap, which is not a frequent source of contention for reasons described in Section 7.7.1.

The DYNIX kernel memory allocator by McKenney and Slingwine [52] and the single object-size allocator by Vee and Hsu [76] employ a ***private heaps with thresholds*** algorithm. These allocators are efficient and scalable because they move large blocks of memory between a hierarchy of per-processor heaps and heaps shared by multiple processors. When a per-processor heap has more than a certain amount of free memory (the threshold), some portion of the free memory is moved to a shared heap. This strategy bounds blowup to a constant factor, since no heap may hold more than some fixed amount of free memory. The mechanisms that control this motion and the units of memory moved by the DYNIX and Vee and Hsu allocators differ significantly from those used by Hoard. Both of these allocators passively induce false sharing by making it very easy for pieces of the same cache line to be recycled. As long as the amount of free memory does not exceed the threshold, pieces of the same cache line spread across processors will be repeatedly reused to satisfy memory requests. Also, these allocators are forced to synchronize every time the threshold amount of memory is allocated or freed, while Hoard can avoid synchronization altogether while the emptiness of per-processor heaps is within the empty fraction. On the other hand, these allocators

do avoid the two-fold slowdown that can occur in the worst-case described for Hoard in Section 7.7.1.

Table 7.1 presents a summary of the above allocator algorithms, along with their speed, scalability, false sharing and blowup characteristics. As can be seen from the table, the algorithms closest to Hoard are Vee and Hsu, DYNIX, and *LKmalloc*. The first two fail to avoid passively-induced false sharing and are forced to synchronize with a global heap after each threshold amount of memory is consumed or freed, while Hoard avoids false sharing and is not required to synchronize until the emptiness threshold is crossed or when a heap does not have sufficient memory. *LKmalloc* has similar synchronization behavior to Hoard and avoids allocator-induced false sharing, but has $O(P)$ blowup.

## 7.4 The Hoard Memory Allocator

This section describes Hoard in detail. Hoard can be viewed as an allocator that generally avoids false sharing and that trades increased (but bounded) memory consumption for reduced synchronization costs.

Hoard augments per-processor heaps with a *global heap* that every thread may access (similar to Vee and Hsu [76]). Each thread can access only its heap and the global heap. We designate heap 0 as the global heap and heaps 1 through $P$ as the per-processor heaps. In the implementation we actually use $2P$ heaps (without altering our analytical results) in order to decrease the probability that concurrently-executing threads use the same heap; we use a simple hash function to map thread id's to per-processor heaps that can result in collisions. We need such a mapping function because in general there is not a one-to-one correspondence between threads and processors, and threads can be reassigned to other processors. On Solaris, however, we avoid collisions of heap assignments to threads by hashing on the light-weight process (LWP) id. The number of LWP's is usually set to the number of processors [51, 69], so each heap is generally used by no more than one LWP.

Hoard maintains *usage statistics* for each heap. These statistics are $u_i$, the amount of memory in use ("live") in heap $i$, and $a_i$, the amount of memory allocated by Hoard from the operating system held in heap $i$.

Hoard allocates memory from the system in chunks we call ***heap blocks***. Each heap block is an

array of some number of blocks (objects) and contains a free list of its available blocks maintained in LIFO order to improve locality. All heap blocks are the same size ($S$), a multiple of the system page size. Hoard manages objects larger than half the size of a heap block directly using the virtual memory system (i.e., Hoard allocates them via `mmap` and frees them using `munmap`). All of the blocks in a heap block are in the same size class. By using size classes that are a power of $b$ apart (where $b$ is greater than 1) and rounding the requested size up to the nearest size class, we bound worst-case *internal* fragmentation within a block to a factor of $b$. In order to reduce *external* fragmentation, we *recycle* completely empty heap blocks for re-use by any size class. For clarity of exposition, we assume a single size class in the discussion below.

### 7.4.1 Bounding Blowup

Each heap "owns" a number of heap blocks. When there is no memory available in any heap block on a thread's heap, Hoard obtains a heap block from the global heap if one is available. If the global heap is also empty, Hoard creates a new heap block by requesting virtual memory from the operating system and adds it to the thread's heap. Hoard does not currently return empty heap blocks to the operating system. It instead makes these heap blocks available for reuse.

Hoard moves heap blocks from a per-processor heap to the global heap when the per-processor heap crosses the ***emptiness threshold***: i.e., more than $f$, the ***empty fraction***, of its blocks are not in use ($u_i < (1-f)a_i$), and there are more than some number $K$ of heap blocks' worth of free memory on the heap ($u_i < a_i - K * S$). As long as a heap is not more than $f$ empty, and contains $K$ or fewer heap blocks, Hoard will not move heap blocks from a per-processor heap to the global heap. Whenever a per-processor heap does cross the emptiness threshold, Hoard transfers one of its heap blocks that is at least $f$ empty to the global heap. Always removing such a heap block whenever we cross the emptiness threshold maintains the following invariant on the per-processor heaps: $(u_i \geq a_i - K * S) \vee (u_i \geq (1-f)a_i)$. When we remove a heap block, we reduce $u_i$ by at most $(1-f)S$ but reduce $a_i$ by $S$, thus restoring the invariant. Maintaining this invariant bounds blowup to a constant factor, as we show in Section 7.5.

Hoard finds $f$-empty heap blocks in constant time by dividing heap blocks into a number of bins that we call "fullness groups". Each bin contains a doubly-linked list of heap blocks that are in a given

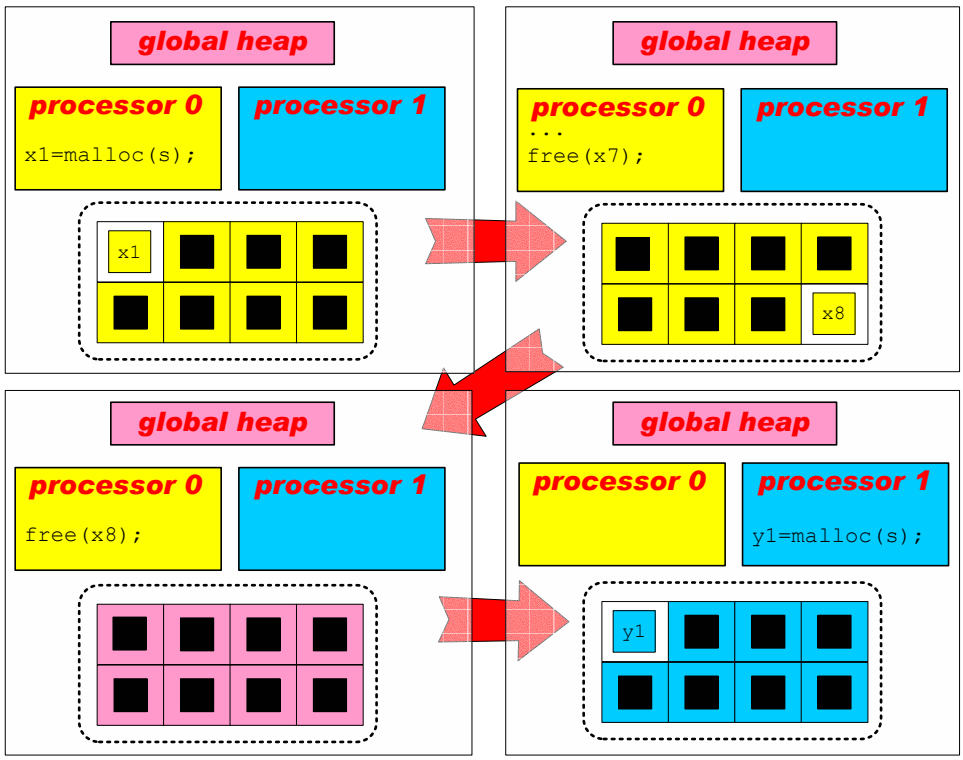



Figure 7.4: Allocation and freeing in Hoard. See Section 7.4.2 for details.

fullness range (e.g., all heap blocks that are between $3/4$ and completely empty are in the same bin). Hoard moves heap blocks from one group to another when appropriate, and always allocates from the fullest heap blocks. To improve locality, we order the heap blocks within a fullness group using a move-to-front heuristic. Whenever we free a block in a heap block, we move the heap block to the front of its fullness group. If we then need to allocate a block, we will be likely to reuse a heap block that is already in memory; because we maintain the free blocks in LIFO order, we are also likely to reuse a block that is already in cache.

### 7.4.2 Example

Figure 7.4 illustrates, in simplified form, how Hoard manages heap blocks. For simplicity, we assume there are two threads and heaps (thread $i$ maps to heap $i$). In this example, the empty fraction $f$ is $7/8$ and $K$ is 0. Initially, all heaps are empty.

The top left diagram shows the heaps after thread 1 allocates `x1`. Hoard allocates a new heap block

and assigns it to heap 1. The next diagram (upper right), shows the state of the heaps after thread 1 has allocated `x2` through `x8` and has freed `x1` through `x7`. The next free causes heap 1 to cross the emptiness threshold, resulting in a transfer of ownership of the empty heap block to the global heap (lower left). In the final diagram, thread 2 causes ownership of the heap block to transfer from the global heap to heap 2 by allocating one object.

### 7.4.3 Avoiding False Sharing

Hoard uses the combination of heap blocks and multiple heaps described above to avoid most active and passive false sharing. Only one thread may allocate from a given heap block since only one heap owns a heap block at any time. When multiple threads make simultaneous requests for memory, the requests will always be satisfied from different heap blocks, avoiding actively induced false sharing. When a program deallocates a block of memory, Hoard returns the block to its heap block. This coalescing prevents multiple threads from reusing pieces of cache lines that were passed to these threads by a user program, avoiding passively-induced false sharing.

While this strategy greatly reduces allocator-induced false sharing, it can not guarantee it will never cause false sharing. Because Hoard may move heap blocks from one heap to another, it is possible for two heaps to share cache lines. In practice, fortunately, heap block transfer is a relatively infrequent event – it occurs only when a per-processor heap drops below the emptiness threshold. We have observed that heap blocks released to the global heap are usually completely empty, eliminating the possibility of false sharing. A simple mechanism to prevent false sharing altogether prohibits allocation from partially-allocated cache lines in transferred heap blocks. This mechanism provably avoids all false sharing of heap objects. Implementing this mechanism remains future work.

## 7.5 Analytical Results

In this section, we prove bounds on blowup and synchronization for Hoard. We first define some useful notation. Let $A(t)$ and $U(t)$ denote the *maximum* amount of memory allocated and in use by the program

`malloc (sz)`

1. If sz $> S/2$, allocate the heap block from the OS
   and *return* it.
2. $i \leftarrow$ hash(the current thread).
3. Lock heap $i$.
4. Scan heap $i$'s list of heap blocks from most full to least
   (for the size class corresponding to sz).
5. If there is no heap block with free space,
6. Check heap 0 (the global heap) for a heap block.
7. If there is none,
8. Allocate $S$ bytes as heap block $s$
   and set the owner to heap $i$.
9. Else,
10. Transfer the heap block $s$ to heap $i$.
11. $u_0 \leftarrow u_0 - s.u$
12. $u_i \leftarrow u_i + s.u$
13. $a_0 \leftarrow a_0 - S$
14. $a_i \leftarrow a_i + S$
15. $u_i \leftarrow u_i +$ sz.
16. $s.u \leftarrow s.u +$ sz.
17. Unlock heap $i$.
18. Return a block from the heap block.

`free (ptr)`

1. If the block is "large",
2. Free the heap block to the operating system and *return*.
3. Find the heap block $s$ this block comes from and lock it.
4. Lock heap $i$, the heap block's owner.
5. Deallocate the block from the heap block.
6. $u_i \leftarrow u_i -$ block size.
7. $s.u \leftarrow s.u -$ block size.
8. If $i = 0$, unlock heap $i$ and the heap block
   and *return*.
9. If $u_i < a_i - K * S$ and $u_i < (1 - f) * a_i$,
10. Transfer a mostly-empty heap block $s1$
    to heap 0 (the global heap).
11. $u_0 \leftarrow u_0 + s1.u$, $u_i \leftarrow u_i - s1.u$
12. $a_0 \leftarrow a_0 + S$, $a_i \leftarrow a_i - S$
13. Unlock heap $i$ and the heap block.

Figure 7.5: Pseudo-code for Hoard's `malloc` and `free`.

("live memory") after memory operation $t$. Let $a(t)$ and $u(t)$ denote the *current* amount of memory allocated and in use by the program after memory operation $t$. We add a subscript for a particular heap (e.g., $u_i(t)$) and add a caret (e.g., $\hat{a}(t)$) to denote the sum for all heaps *except* the global heap.

## 7.6 Bounds on Blowup

We formally define the blowup for an allocator as its worst-case memory consumption divided by the ideal worst-case memory consumption for a serial memory allocator (a constant factor times its maximum memory required [60]):

**Definition 1** *blowup* $= O(A(t)/U(t))$.

By maintaining no more than a constant fraction of unused memory on each heap and moving free memory to the global heap, we can prove the following theorem:

**Theorem 1** $A(t) = O(U(t) + P)$.

By the definition of blowup above, and assuming that $P << U(t)$, Hoard's blowup is $O((U(t) + P)/U(t)) = O(1)$. This result shows that Hoard's worst case memory consumption is at worst a constant factor overhead that does not grow with the amount of memory required by the program. This result dramatically improves on the blowup for non-threshold allocators, which is $O(P)$.

### 7.6.1 Proof

We make use of the following lemma:

**Lemma 1** $A(t) = \hat{A}(t)$.

This lemma holds because these quantities are maxima, and any memory in the global heap was originally allocated into a per-processor heap. Now we prove the bounded memory consumption theorem

above ($A(t) = O(U(t) + P)$).

*Proof.* We restate the invariant from Section 7.4.1 that we maintain over all the per-processor heaps: $(a_i(t) - K * S \leq u_i(t)) \vee ((1 - f)a_i(t) \leq u_i(t))$.

The first inequality is sufficient to prove the theorem. Summing over all $P$ per-processor heaps gives us

$$\begin{aligned}
\hat{A}(t) &\leq \textstyle\sum_{i=1}^{P} u_i(t) + P * K * S && \triangleright \textit{def. of } \hat{A}(t) \\
&\leq \hat{U}(t) + P * K * S && \triangleright \textit{def. of } \hat{U}(t) \\
&\leq U(t) + P * K * S. && \triangleright \hat{U}(t) \leq U(t)
\end{aligned}$$

Since by the above lemma $A(t) = \hat{A}(t)$, we have $A(t) = O(U(t) + P)$. ∎

Because the number of size classes is constant, this theorem holds over all size classes. By the definition of blowup above, and assuming that $P << U(t)$, Hoard's blowup is $O((U(t)+P)/U(t)) = O(1)$. This result shows that Hoard's worst case memory consumption is at worst a constant factor overhead that does not grow with the amount of memory required by the program.

Our discipline for using the empty fraction ($f$) enables this proof, so it is clearly a key parameter for Hoard. For reasons we describe and validate with experimental results in Section 7.10.3, Hoard's performance is robust with respect to the choice of $f$.

## 7.7 Bounds on Synchronization

We now analyze Hoard's worst-case and discuss expected synchronization costs. Synchronization costs come in two flavors: contention for a per-processor heap and acquisition of the global heap lock. We argue that the first form of contention is not a scalability concern, and that the second form is rare. Further, for common program behavior, the synchronization costs are low over most of the program's lifetime.

### 7.7.1 Per-processor Heap Contention

The worst-case contention for Hoard arises when one thread allocates memory from the heap and all other threads free it (thus all contending for the same heap lock). If an application allocates memory in such a manner and the amount of work between allocations is so low that heap contention is an issue, then the application itself is fundamentally unscalable. Even if heap access were to be completely independent, the application itself could only achieve a two-fold speedup, no matter how many processors are available.

Since we are concerned with providing a scalable allocator for scalable applications, we can bound Hoard's worst case for such applications, which occurs when pairs of threads exhibit producer-consumer behavior. Each `malloc` and each `free` will be serialized. Modulo context-switch costs, this pattern results in at most a two-fold slowdown. This slowdown is not desirable but it is scalable as it does not grow with the number of processors (as it does for allocators with one heap protected by a single lock).

It is difficult to establish an expected case for per-processor heap contention. In our own and others' experience with multithreaded applications [49], the allocating thread exclusively uses most of its dynamically-allocated memory, and only a small fraction of allocated memory is freed by another thread. We thus find and expect per-processor heap contention to be quite low.

### 7.7.2 Global Heap Contention

Global heap contention arises when heap blocks are first created, when heap blocks are transferred to and from the global heap, and when blocks are freed from heap blocks held by the global heap. We simply count the number of times the global heap's lock is acquired by each thread, to develop an upper-bound on global heap contention. We analyze two cases: a growing phase and a shrinking phase. We show that worst-case synchronization for the growing phases is inversely proportional to the heap block size and the empty fraction. We show that the worst-case for the shrinking phase is expensive but only for a pathological case that is unlikely to occur in practice. Empirical evidence from Section 7.8 suggests that Hoard will incur low synchronization costs.

Two key parameters control the worst-case global heap contention while a per-processor heap is

growing: $f$, the empty fraction, and $S$, the size of a heap block. When a per-processor heap is growing, a thread can acquire the global heap lock at most $k/(f * S/s)$ times for $k$ memory operations, where $s$ is the object size. Whenever the per-processor heap is empty, the thread will lock the global heap and obtain a heap block with at least $f * S/s$ free blocks. If the thread then calls `malloc` $k$ times, it will exhaust its heap and acquire the global heap lock at most $k/(f * S/s)$ times.

When a per-processor heap is shrinking, a thread will first acquire the global heap lock when the release threshold is crossed. The release threshold could then be crossed on every single call to `free` if every heap block is exactly $f$ empty. Completely freeing each heap block in turn will cause the heap block to first be released to the global heap and every subsequent `free` to a block in that heap block will therefore acquire the global heap lock. Luckily, this pathological case is highly unlikely to occur since it requires an improbable sequence of operations: the program must systematically free $(1 - f)$ of each heap block and then free every block in a heap block in round-robin order.

For the common case, Hoard will incur *very low* contention costs for any memory operation. This situation holds when the amount of live memory remains within the empty fraction of the maximum amount of memory allocated (and when all `free`s are local). Johnstone [45] and Stefanović [68] show in their empirical studies of allocation behavior that for nearly every program they analyzed, the memory in use tends to vary within a range that is within a fraction of total memory currently in use, and this amount often grows steadily. Thus, in the steady state case, Hoard incurs no contention, and in gradual growth, Hoard incurs low contention.

## 7.8 Experimental Results

In this section, we investigate Hoard's performance experimentally. We performed experiments on uniprocessors and multiprocessors to demonstrate Hoard's speed, scalability, false sharing avoidance, and low fragmentation. We used the dedicated 14-processor Sun Enterprise 5000 described in Table 3.2. In the experiments below, the size of a heap block $S$ is 8K, the empty fraction $f$ is $3/4$, the number of heap blocks $K$ that must be free for heap blocks to be released is 4, and the base of the exponential for size classes $b$ is

| *multithreaded benchmarks* | |
|---|---|
| threadtest | each thread repeatedly allocates and then deallocates 100,000/$P$ objects |
| shbench [55] | each thread allocates and randomly frees random-sized objects |
| Larson [49] | simulates a server: each thread allocates and deallocates objects, and then transfers some objects to other threads to be freed |
| active-false | tests active false sharing avoidance |
| passive-false | tests passive false sharing avoidance |
| BEMengine [21] | object-oriented PDE solver |
| Barnes-Hut [1, 6] | $n$-body particle solver |

Table 7.2: Multithreaded benchmarks used in this chapter.

1.2 (bounding internal fragmentation to 1.2).

We compare Hoard (version 2.0.2) to the following single and multiple-heap memory allocators: *Solaris*, the default allocator provided with Solaris 7, *Ptmalloc* [31], the Linux allocator included in the GNU C library that extends a traditional allocator to use multiple heaps, and *MTmalloc*, a multiple heap allocator included with Solaris 7 for use with multithreaded parallel applications. (Section 7.2 includes extensive discussion of *Ptmalloc*, *MTmalloc*, and other concurrent allocators.) The latter two are the only publicly-available concurrent allocators of which we are aware for the Solaris platform (for example, *LKmalloc* is Microsoft proprietary and does not work under Solaris). We use the Solaris allocator as the baseline for calculating speedups.

To measure Hoard's performance and memory utilization for uniprocessor memory allocation, we ran several of the Memory-Intensive benchmarks (see Section 3.1.1). These include the following programs: *espresso*, an optimizer for programmable logic arrays; *Ghostscript*, a PostScript interpreter, and *LRUsim*, a locality analyzer. We chose these programs because they are allocation-intensive and have widely varying memory usage patterns. We used the same inputs for these programs as Wilson and Johnstone [46].

There is as yet no standard suite of benchmarks for evaluating multithreaded allocators. We know of no benchmarks that specifically stress multithreaded performance of server applications like web servers [2]

[2]Memory allocation becomes a bottleneck when most pages served are dynamically generated. Unfortunately, the SPECweb99 benchmark [67] performs very few requests for completely dynamically-generated pages (0.5%), and most web servers exercise dynamic memory allocation only when generating dynamic content.

| Benchmark applications | Fragmentation ($A/U$) | max in use ($U$) | max allocated ($A$) | total memory requested | # objects requested | average object size |
|---|---|---|---|---|---|---|
| *multithreaded benchmarks* | | | | | | |
| threadtest | 1.24 | 1,068,864 | 1,324,848 | 80,391,016 | 9,998,831 | 8 |
| shbench | 3.17 | 556,112 | 1,761,200 | 1,650,564,600 | 12,503,613 | 132 |
| Larson | 1.22 | 8,162,600 | 9,928,760 | 1,618,188,592 | 27,881,924 | 58 |
| BEMengine | 1.02 | 599,145,176 | 613,935,296 | 4,146,087,144 | 18,366,795 | 226 |
| Barnes-Hut | 1.18 | 11,959,960 | 14,114,040 | 46,004,408 | 1,172,624 | 39 |

Table 7.3: Hoard fragmentation results and application memory statistics. We report fragmentation statistics for 14-processor runs of the multithreaded programs. All units are in bytes.

and database managers. We chose benchmarks described in other papers and otherwise published: the *Larson* benchmark from Larson and Krishnan [49] and the *shbench* benchmark from MicroQuill, Inc. [55]. We use two multithreaded applications: *BEMengine* [21] and *barnes-hut* [1, 6], and we wrote some microbenchmarks of our own to stress different aspects of memory allocation performance (*threadtest*, *active-false*, *passive-false*). Table 7.2 describes all of the benchmarks. Table 7.6 includes their allocation behavior: fragmentation, maximum memory in use ($U$) and allocated ($A$), total memory requested, number of objects requested, and average object size.

### 7.8.1 Speed

Table 7.4 lists the uniprocessor runtimes for our applications when linked with Hoard and the Solaris allocator. Hoard causes a slight increase in the runtime of these applications (harmonic mean = 4.3%), but this loss is primarily due to its performance on *shbench*. Hoard performs poorly on *shbench* because *shbench* uses a wide range of size classes (spreading out objects across many heap blocks) but allocates very little memory (see Section 7.10.2 for more details). Excluding *shbench*, Hoard performs nearly identically to the Solaris allocator when running on one processor (harmonic mean = 0.5%). The longest-running application, *LRUsim*, runs almost 3% faster with Hoard. Hoard also performs well on *BEMengine* (10.3% faster than with the Solaris allocator), which allocates more memory than any of our other benchmarks (nearly 600MB).

| program | runtime (sec) | | change |
|---|---|---|---|
| | Solaris | Hoard | |
| *single-threaded benchmarks* | | | |
| espresso | 6.806 | 7.887 | +15.9% |
| Ghostscript | 3.610 | 3.993 | +10.6% |
| LRUsim | 1615.413 | 1570.488 | -2.9% |
| *multithreaded benchmarks* | | | |
| threadtest | 16.549 | 15.599 | -6.1% |
| shbench | 12.730 | 18.995 | +49.2% |
| active-false | 18.844 | 18.959 | +0.6% |
| passive-false | 18.898 | 18.955 | +0.3% |
| BEMengine | 678.30 | 614.94 | -10.3% |
| Barnes-Hut | 192.51 | 190.66 | -1.0% |
| *harmonic mean* | | | *+4.3%* |

Table 7.4: Uniprocessor runtimes for single- and multithreaded benchmarks.

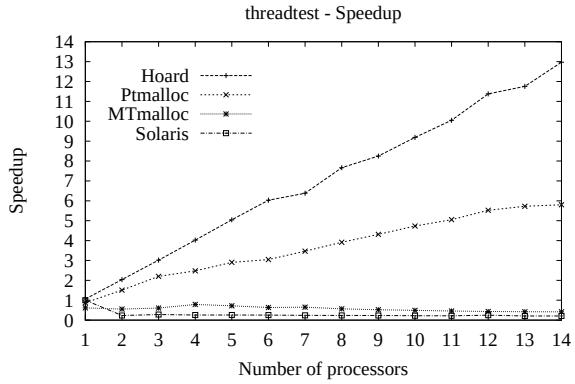
### 7.8.2 Scalability

In this section, we present our experiments to measure scalability. We measure *speedup* with respect to the Solaris allocator. These applications vigorously exercise the allocators as revealed by the large difference between the maximum in use and the total memory requested (see Table 7.6).

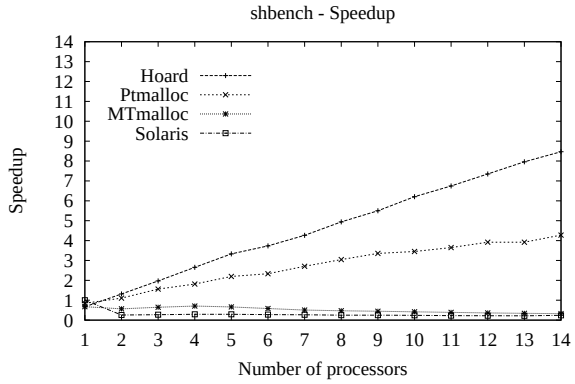
Figure 7.6 shows that Hoard matches or outperforms all of the allocators we tested. The Solaris allocator performs poorly overall because serial single heap allocators do not scale. *MTmalloc* often suffers from a centralized bottleneck. *Ptmalloc* scales well only when memory operations are fairly infrequent (the *Barnes-Hut* benchmark in Figure 7.6(d)); otherwise, its scaling peaks at around 6 processors. We now discuss each benchmark in turn.

In *threadtest*, $t$ threads do nothing but repeatedly allocate and deallocate $100,000/t$ 8-byte objects (the threads do not synchronize or share objects). As seen in Figure 7.6(a), Hoard exhibits linear speedup, while the Solaris and *MTmalloc* allocators exhibit severe slowdown. For 14 processors, the Hoard version runs 278% faster than the *Ptmalloc* version. Unlike *Ptmalloc*, which uses a linked-list of heaps, Hoard does not suffer from a scalability bottleneck caused by a centralized data structure.

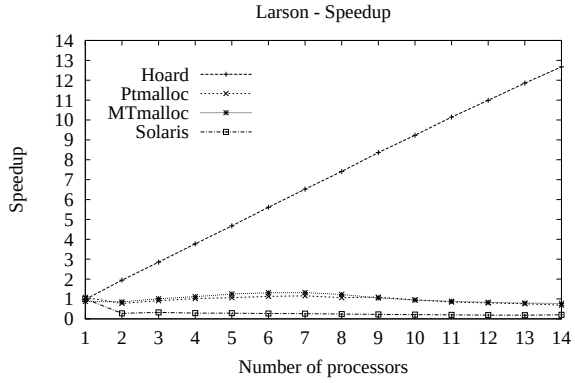
The *shbench* benchmark is available on MicroQuill's website and is shipped with the SmartHeap SMP product [55]. This benchmark is essentially a "stress test" rather than a realistic simulation of appli-



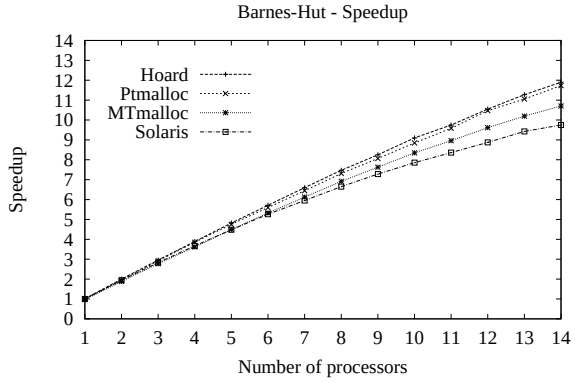

(a) The Threadtest benchmark.



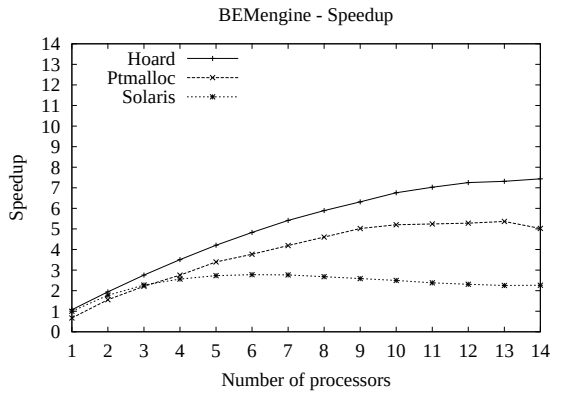

(b) The SmartHeap benchmark (*shbench*).



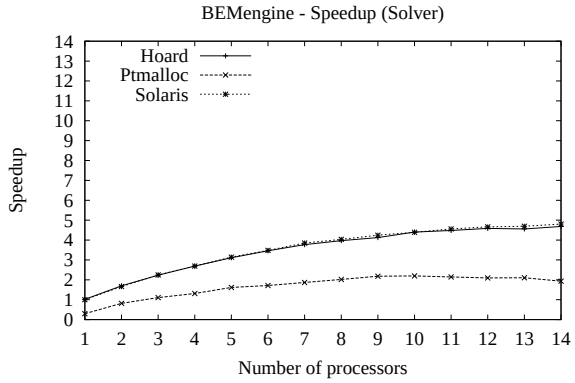

(c) Speedup using the Larson benchmark.




(d) Barnes-Hut speedup.




(e) *BEMengine* speedup. Linking with *MTmalloc* caused an exception to be raised.




(f) *BEMengine* speedup for the system solver only.

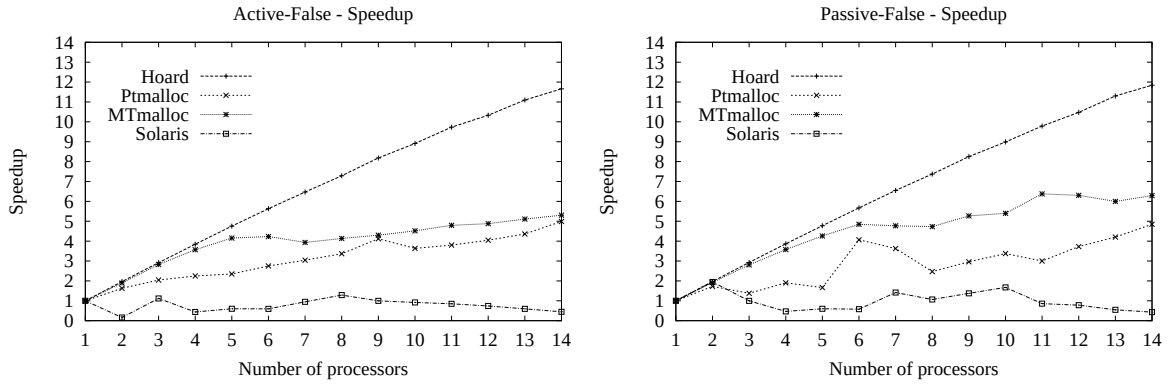
Figure 7.6: Speedup graphs.

cation behavior. Each thread repeatedly allocates and frees a number of randomly-sized blocks in random order, for a total of 50 million allocated blocks. The graphs in Figure 7.6(b) show that Hoard scales quite well, approaching linear speedup as the number of threads increases. The slope of the speedup line is less than ideal because the large number of different size classes hurts Hoard's raw performance. For 14 processors, the Hoard version runs 85% faster than the next best allocator (*Ptmalloc*). Memory usage in *shbench* remains within the empty fraction during the entire run so that Hoard incurs very low synchronization costs, while *Ptmalloc* again runs into its scalability bottleneck.

The intent of the *Larson* benchmark, due to Larson and Krishnan [49], is to simulate a workload for a server. A number of threads are repeatedly spawned to allocate and free 10,000 blocks ranging from 10 to 100 bytes in a random order. Further, a number of blocks are left to be freed by a subsequent thread. Larson and Krishnan observe this behavior (which they call "bleeding") in actual server applications, and their benchmark simulates this effect. The benchmark runs for 30 seconds and then reports the number of memory operations per second. Figure 7.6(c) shows that Hoard scales linearly, attaining nearly ideal speedup. For 14 processors, the Hoard version runs 18 times faster than the next best allocator, the *Ptmalloc* version. After an initial start-up phase, *Larson* remains within its empty fraction for most of the rest of its run (dropping below one-eighth empty only a few times over a 30-second run and over 27 million `mallocs`) and thus Hoard incurs very low synchronization costs. Despite the fact that *Larson* transfers many objects from one thread to another, Hoard performs quite well. All of the other allocators fail to scale at all, running slower on 14 processors than on one processor.

*Barnes-Hut* is a hierarchical $n$-body particle solver included with the Hood user-level multiprocessor threads library [1, 6], run on 32,768 particles for 20 rounds. This application performs a small amount of dynamic memory allocation during the tree-building phase. With 14 processors, all of the multiple-heap allocators provide a 10% performance improvement, increasing the speedup of the application from less than 10 to just above 12 (see Figure 7.6(d)). Hoard performs only slightly better than *Ptmalloc* in this case because this program does not exercise the allocator much. Hoard's performance is probably somewhat better simply because *Barnes-Hut* never drops below its empty fraction during its execution.

The *BEMengine* benchmark uses the solver engine from Coyote Systems' BEMSolver [21], a 2D/3D




(a) Speedup for the *active-false* benchmark, which fails to scale with memory allocators that *actively* induce false sharing.

(b) Speedup for the *passive-false* benchmark, which fails to scale with memory allocators that *passively* or *actively* induce false sharing.

Figure 7.7: Speedup graphs that exhibit the effect of allocator-induced false sharing.

field solver that can solve electrostatic, magnetostatic and thermal systems. We report speedup for the three mostly-parallel parts of this code (equation registration, preconditioner creation, and the solver). Figure 7.6(e) shows that Hoard provides a significant runtime advantage over *Ptmalloc* and the Solaris allocator (*MTmalloc* caused the application to raise a fatal exception)[3]. During the first two phases of the program, the program's memory usage dropped below the empty fraction only 25 times over 50 seconds, leading to low synchronization overhead. This application causes *Ptmalloc* to exhibit pathological behavior that we do not understand, although we suspect that it derives from false sharing. During the execution of the solver phase of the computation, as seen in Figure 7.6(f), contention in the allocator is not an issue, and both Hoard and the Solaris allocator perform equally well.

## 7.9 False sharing

We designed two test programs, *active-false* and *passive-false*, to induce active and passive false sharing to reveal the performance impact on the memory allocators. The active-false benchmark tests whether an allocator avoids actively inducing false sharing. Each thread allocates one small object, writes on it a number of times, and then `free`s it. The rate of memory allocation is low compared to the amount of work done, so

[3]The author of BEMEngine confirms that its algorithms do not scale linearly (Martin Bächtold, personal communication)

| program | falsely-shared objects |
|---|---|
| threadtest | 0 |
| shbench | 0 |
| Larson | 0 |
| BEMengine | 0 |
| Barnes-Hut | 0 |

Table 7.5: Possible falsely-shared objects on 14 processors.

this benchmark only tests contention caused by the cache coherence mechanism (cache ping-ponging) and not allocator contention. While Hoard scales linearly, showing that it avoids actively inducing false sharing, both *Ptmalloc* and *MTmalloc* only scale up to about 4 processors because they actively induce some false sharing. The Solaris allocator does not scale at all because it actively induces false sharing for nearly every cache line.

The *passive-false* benchmark tests whether an allocator avoids both passive and active false sharing by allocating a number of small objects in one thread and giving one to each other thread, which immediately `free`s the object. The benchmark then continues in the same way as the *active-false* benchmark. If the allocator does not coalesce the pieces of the cache line initially distributed to the various threads, it passively induces false sharing. Figure 7.7(b) shows that Hoard scales nearly linearly; the gradual slowdown after 12 processors is due to program-induced bus traffic. Neither *Ptmalloc* nor *MTmalloc* avoid false sharing here, but the cause could be either active or passive false sharing.

In Table 7.5, we present measurements for our multithreaded benchmarks of the number of objects that could have been responsible for allocator-induced false sharing in Hoard (i.e., those objects already in a heap block acquired from the global heap). In every case, when the per-processor heap acquired heap blocks from the global heap, the heap blocks were empty. These results demonstrate that Hoard successfully avoids allocator-induced false sharing.

## 7.10 Fragmentation

We showed in Section 7.4.1 that Hoard has bounded blowup. In this section, we measure Hoard's average case fragmentation. We use a number of single- and multithreaded applications to evaluate Hoard's average-

| Benchmark applications | Fragmentation ($A/U$) | max in use ($U$) | max allocated ($A$) | total memory requested | # objects requested | average object size |
|---|---|---|---|---|---|---|
| *multithreaded benchmarks* | | | | | | |
| threadtest | 1.24 | 1,068,864 | 1,324,848 | 80,391,016 | 9,998,831 | 8 |
| shbench | 3.17 | 556,112 | 1,761,200 | 1,650,564,600 | 12,503,613 | 132 |
| Larson | 1.22 | 8,162,600 | 9,928,760 | 1,618,188,592 | 27,881,924 | 58 |
| BEMengine | 1.02 | 599,145,176 | 613,935,296 | 4,146,087,144 | 18,366,795 | 226 |
| Barnes-Hut | 1.18 | 11,959,960 | 14,114,040 | 46,004,408 | 1,172,624 | 39 |

Table 7.6: Hoard fragmentation results and application memory statistics. We report fragmentation statistics for 14-processor runs of the multithreaded programs. All units are in bytes.

case fragmentation.

Collecting fragmentation information for multithreaded applications is problematic because fragmentation is a global property. Updating the maximum memory in use and the maximum memory allocated would serialize all memory operations and thus seriously perturb allocation behavior. We cannot simply use the maximum memory in use for a serial execution because a parallel execution of a program may lead it to require much more memory than a serial execution.

We solve this problem by collecting traces of memory operations and processing these traces off-line. We modified Hoard so that (when collecting traces) each per-processor heap records every memory operation along with a timestamp (using the SPARC high-resolution timers via `gethrtime()`) into a memory-mapped buffer and writes this trace to disk upon program termination. We then merge the traces in timestamp order to build a complete trace of memory operations and process the resulting trace to compute maximum memory allocated and required. Collecting these traces results in nearly a threefold slowdown in memory operations but does not excessively disturb their parallelism, so we believe that these traces are a faithful representation of the fragmentation induced by Hoard.

### 7.10.1 Single-threaded Applications

In order to measure Hoard's impact on space for uniprocessor applications, we measure fragmentation for the Memory-Intensive benchmark suite (see Section 3.1.1). We follow Wilson and Johnstone [46] and report memory allocated without counting overhead (like per-object headers) to focus on the allocation *policy* rather than the *mechanism*. Hoard's fragmentation for these applications is between 1.05 and 1.2, except for

*espresso*, which consumes 46% more memory than it requires. *Espresso* is an unusual program since it uses a large number of different size classes for a small amount of memory required (less than 300K), and this behavior leads Hoard to waste space within each 8K heap block.

### 7.10.2 Multithreaded Applications

Table 7.6 shows that the fragmentation results for the multithreaded benchmarks are generally quite good, ranging from nearly no fragmentation (1.02) for *BEMengine* to 1.24 for *threadtest*. The anomaly is *shbench*. This benchmark uses a large range of object sizes, randomly chosen from 8 to 100, and many objects remain live for the duration of the program (470K of its maximum 550K objects remain in use at the end of the run cited here). These unfreed objects are randomly scattered across heap blocks, making it impossible to recycle them for different size classes. This extremely random behavior is not likely to be representative of real programs [46] but it does show that Hoard's method of maintaining one size class per heap block can yield poor memory efficiency for certain behaviors, although Hoard still attains good scalable performance for this application (see Figure 7.6(b)).

### 7.10.3 Sensitivity Study

We also examined the effect of changing the empty fraction on runtime and fragmentation for the multithreaded benchmarks. Because heap blocks are returned to the global heap (for reuse by other threads) when the heap crosses the emptiness threshold, the empty fraction affects both synchronization and fragmentation. We varied the empty fraction from $1/8$ to $1/2$ and saw very little change in runtime and fragmentation. We chose this range to exercise the tension between increased (worst-case) fragmentation and synchronization costs. The only benchmark which is substantially affected by these changes in the empty fraction is the *Larson* benchmark, whose fragmentation increases from 1.22 to 1.61 for an empty fraction of $1/2$. Table 7.7 presents the runtime for these programs on 14 processors (we report the number of memory operations per second for the Larson benchmark, which runs for 30 seconds), and Table 7.8 presents the fragmentation results. Hoard's runtime is robust with respect to changes in the empty fraction because programs tend to reach a steady state in memory usage and stay within even as small an empty fraction as $1/8$, as described

| program | runtime (sec) | | |
|---|---|---|---|
| | $f = 1/8$ | $f = 1/4$ | $f = 1/2$ |
| threadtest | 1.27 | 1.28 | 1.19 |
| shbench | 1.45 | 1.50 | 1.44 |
| BEMengine | 86.85 | 87.49 | 88.03 |
| Barnes-Hut | 16.52 | 16.13 | 16.41 |
| | **throughput (memory ops/sec)** | | |
| Larson | 4,407,654 | 4,416,303 | 4,352,163 |

Table 7.7: Runtime on 14 processors using Hoard with different empty fractions.

| program | fragmentation | | |
|---|---|---|---|
| | $f = 1/8$ | $f = 1/4$ | $f = 1/2$ |
| threadtest | 1.22 | 1.24 | 1.22 |
| shbench | 3.17 | 3.17 | 3.16 |
| Larson | 1.22 | 1.22 | 1.61 |
| BEMengine | 1.02 | 1.02 | 1.02 |
| Barnes-Hut | 1.18 | 1.18 | 1.18 |

Table 7.8: Fragmentation on 14 processors using Hoard with different empty fractions.

in Section 7.7.2.

## 7.11 Conclusion

In this chapter, we have introduced the Hoard memory allocator. Hoard improves on previous memory allocators by simultaneously providing four features that are important for scalable application performance: speed, scalability, false sharing avoidance, and low fragmentation. Hoard's novel organization of per-processor and global heaps along with its discipline for moving heap blocks across heaps enables Hoard to achieve these features and is the key contribution of this work. Our analysis shows that Hoard has provably bounded blowup and low expected case synchronization. Our experimental results on eleven programs demonstrate that in practice Hoard has low fragmentation, avoids false sharing, and scales very well. In addition, we show that Hoard's performance and fragmentation are robust with respect to its primary parameter, the empty fraction. Since scalable application performance clearly requires scalable architecture and runtime system support, Hoard thus takes a key step in this direction.

# Chapter 8

# Conclusion

## 8.1 Future Work

The research presented in this thesis points to several areas for future work. First, heap layers are an enabling technology for experimentation. Most design decisions in memory managers are made early and would be difficult to change. Using heap layers, these design decisions can be isolated in individual layers, facilitating experimentation with different policies and mechanisms. We believe that we can use heap layers to solve numerous open questions in memory management.

Heap layers can also be used to develop richer application-specific memory managers. Using profile information, it is possible to discover allocation and access patterns and produce custom memory managers that exploit these. While we show that most custom memory managers do not provide significant performance gains, we believe that exploiting richer profiles and adapting to more complex application behavior can provide improved performance, especially on multiprocessors. Such optimizations include padding out allocations to avoid false sharing and using atomic deques [3] to manage memory between threads in producer-consumer relationships.

We have developed two different memory managers, Hoard and reaps, to address two aspects of memory management. Hoard provides scalable concurrent general-purpose memory management, and reaps provide extra semantics for server applications. We believe that combining these two into one memory

manager would simultaneously address the needs of most high-performance applications.

## 8.2 Contributions

Despite its long history, memory management remains a significant performance and scalability bottleneck for modern high-performance applications. Programmers currently build custom memory managers by hand in order to achieve high performance or semantics they cannot obtain with the system-provided general-purpose allocator. This process is difficult, error-prone, precludes code reuse and results in sub-optimal memory usage. Because of scalability problems in system-provided general-purpose allocators, multithreaded applications often do not scale on multiprocessors. These problems prevent many applications from achieving high performance.

We present heap layers, a software infrastructure that simplifies construction and reuse of high-performance memory managers. We show that heap layers allow programmers to build memory managers that match or exceed the performance of their monolithic hand-tuned counterparts. We show that the use of custom memory managers is generally a mistake, yielding no significant gains in performance. We present reaps, a generalization of regions and heaps that provides high performance while addressing the special needs of server applications on uniprocessors. To address the additional problems posed by multithreaded applications, we develop Hoard, a scalable concurrent memory manager. Our experimental results demonstrate that Hoard achieves its goals of scalability, false-sharing avoidance, and bounded memory consumption.

The key contribution of this thesis is the development of a framework for understanding and constructing high-performance, scalable memory managers. We show that, despite the long history of work on memory management, we can still build much better memory managers.

# Bibliography


[1] Umut Acar, Emery Berger, Robert Blumofe, and Dionysios Papadopoulos. Hood: A threads library for multiprogrammed multiprocessors. http://www.cs.utexas.edu/users/hood, September 1999.

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

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[4] G. Attardi and T. Flagella. A customizable memory management framework. In *Proceedings of the USENIX C++ Conference*, Cambridge, Massachussetts, 1994.

[5] Giuseppe Attardi, Tito Flagella, and Pietro Iglio. A customizable memory management framework for C++. In *Software Practice & Experience*, number 28(11), pages 1143–1183. Wiley, 1998.

[6] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[7] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 187–196, Albuquerque, New Mexico, June 1993.

[8] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through

product-lines and domain-specific languages: A case study. In *Proceedings of the International Conference on Software Reuse*, Vienna, Austria, 2000.

[9] bCandid.com, Inc. http://www.bcandid.com.

[10] William S. Beebee and Martin C. Rinard. An implementation of scoped memory for real-time java. In *EMSOFT*, pages 289–305, 2001.

[11] 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.

[12] 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.

[13] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, November 2002.

[14] B. Bigler, S. Allan, and R. Oldehoeft. Parallel dynamic storage allocation. *International Conference on Parallel Processing*, pages 272–275, 1985.

[15] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[16] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[17] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOP-*

*SLA) / Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

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

[19] Richard Cardone and Calvin Lin. Comparing frameworks and layered refinement. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, May 2001.

[20] 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.

[21] Coyote Systems, Inc. http://www.coyotesystems.com.

[22] Carla Schlatter Ellis and Thomas J. Olson. Algorithms for parallel memory allocation. *International Journal of Parallel Programming*, 17(4):303–345, 1988.

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

[24] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[25] Robert P. Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for java. *Software - Practice and Experience*, 30(3):199–232, 2000.

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

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

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

[29] 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.

[30] 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.

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

[32] A. Gottlieb and J. Wilson. Using the buddy system for concurrent memory allocation. Technical Report System Software Note 6, Courant Institute, 1981.

[33] A. Gottlieb and J. Wilson. Parallelizing the usual buddy algorithm. Technical Report System Software Note 37, Courant Institute, 1982.

[34] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. In *Software Practice & Experience*, number 23(8), pages 851–869. Wiley, August 1993.

[35] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 177–186, New York, NY, June 1993.

[36] 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.

[37] 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.

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

[39] 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.

[40] Arun K. Iyengar. *Dynamic Storage Allocation on a Multiprocessor*. PhD thesis, MIT, 1992. MIT Laboratory for Computer Science Technical Report MIT/LCS/TR–560.

[41] Arun K. Iyengar. Parallel dynamic storage allocation algorithms. In *Fifth IEEE Symposium on Parallel and Distributed Processing*. IEEE Press, 1993.

[42] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 179–188, July 1995.

[43] T. Johnson. A concurrent fast-fits memory manager. Technical Report TR91-009, University of Florida, Department of CIS, 1991.

[44] Theodore Johnson and Tim Davis. Space efficient parallel buddy memory management. Technical Report TR92-008, University of Florida, Department of CIS, 1992.

[45] Mark S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, December 1997.

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

[47] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the Sixth International Conference on Supercomputing*, pages 323–334, Distributed Computing, July 1992.

[48] Murali R. Krishnan. Heap: Pleasures and pains. Microsoft Developer Newsletter, February 1999.

[49] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *International Symposium on Memory Management*, Vancouver, B.C., Canada, 1998.

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

[51] Bil Lewis. `comp.programming.threads` FAQ. http://www.lambdacs.com/newsgroup/FAQ.html.

[52] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 295–305, Berkeley, CA, USA, Winter 1993. USENIX.

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

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

[55] MicroQuill, Inc. http://www.microquill.com.

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

[57] Girija J. Narlikar and Guy E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, January 1999.

[58] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.

[59] Jeffrey Richter. *Advanced Windows: the developer's guide to the Win32 API for Windows NT 3.5 and Windows 95*. Microsoft Press.

[60] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *ACM Computer Journal*, 20(3):242–244, August 1977.

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

[62] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 12–23, October 1998.

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

[64] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '98)*, pages 550–570, Brussels, Belgium, 1998.

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

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

[67] Standard Performance Evaluation Corporation. SPECweb99. http://www.spec.org/osg/web99/.

[68] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, December 1998.

[69] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of the 1992 USENIX Summer Conference*, pages 1–9, 1992.

[70] H. Stone. Parallel memory allocation using the FETCH-AND-ADD instruction. Technical Report RC 9674, IBM T. J. Watson Research Center, November 1982.

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

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

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

[74] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[75] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proceedings of OOPSLA 1996*, pages 359–369, October 1996.

[76] Voon-Yee Vee and Wen-Jing Hsu. A scalable and efficient storage allocator on shared-memory multiprocessors. In *International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 230–235, Fremantle, Western Australia, June 1999.

[77] Ronald Veldema, Thilo Kielmann, and Henri E. Bal. Optimizing java-specific overheads: Java at the speed of c? In *HPCN Europe*, pages 685–692, 2001.

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

[79] Olivier Wall. Private communication. February 2001.

[80] 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.

[81] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

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

# Vita

Emery Berger was born in New York City to George and Sharon Berger, and has two younger brothers, Ryan and Doug. He grew up in Florida and received a B.S. in Computer Science from the University of Miami. He received a Master's degree in Computer Sciences from the University of Texas at Austin in 1991. He and his wife Elayne then taught at the Benjamin Franklin International School in Barcelona, Spain for two years. He returned to the University of Texas to pursue a Ph.D. and spent two summers as a research intern at Microsoft Research in Redmond, Washington. In September, he will be joining the faculty of the Department of Computer Science at the University of Massachusetts, Amherst.

Emery has been married to his lovely wife Elayne Robin Shields since 1989, and has two delightful children, Sophia Alexandra Berger (born in 1997), and Benjamin Charles Berger (born in 1999).

Permanent Address: 1802 Burbank St.

Austin, TX 78757

USA

This dissertation was typeset with LaTeX $2_\varepsilon$[1] by the author.

[1] LaTeX $2_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.