Features of A Multi-Threaded Memory Allocator

by

Ayelet Wasik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2008

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Multi-processor computers are becoming increasingly popular and are important for improving application performance. Providing high-performance memory-management is important for multi-threaded programs. This thesis looks at memory allocation of dynamic-allocation memory in concurrent C and C++ programs. The challenges facing the design of any memory allocator include minimizing fragmentation, and promoting good locality. A multi-threaded memory-allocator is also concerned with minimizing contention, providing mutual exclusion, avoiding false-sharing, and preventing heap-blowup (a form of fragmentation).

Several potential features are identified in existing multi-threaded memory-allocators. These features include per-thread heaps with a global heap, object ownership, object containers, thread-local free-list buffers, remote free-lists, allocation buffers, and lock-free operations. When used in different combinations, these features can solve most of the challenges facing a multi-threaded memory-allocator. Through the use of a test suite composed of both single and multi-threaded benchmark programs, several existing memory allocators and a set of new allocators are compared. It is determined that different features address different multi-threaded issues in the memory allocator with respect to performance, scaling, and fragmentation. Finally, recommendations are made for the design of a general-purpose memory-allocator.

Acknowledgments

I would like to acknowledge Peter Buhr, Ashif Harji, and Richard Bilson for their input and assistance in the work that went into this thesis. I would also like to acknowledge the members of my review committee, Martin Karsten and Ondrej Lhotak, for the assistance they provided. Lastly, acknowledgments go to my family for their proofreading help.

Contents

List of Tables xv

List of Figures xvi

1 Introduction 1

1.1 Memory Structure 1

1.2 Dynamic-Memory Management 2

1.3 Contributions 3

1.4 Outline 3

2 Memory Allocator Background 5

2.1 Components of a Memory Allocator 5

2.2 Single-Threaded Memory Allocators 7

2.2.1 Fragmentation 7

2.2.2 Locality 11

2.3 Multi-Threaded Memory Allocators 13

2.3.1 Mutual Exclusion 13

2.3.2 False Sharing 13

2.3.3 Heap Blowup 15

3 Memory Allocator Design **17**

3.1 Multi-Threaded Memory-Allocator Features . . . 17

3.1.1 Per-Thread Heaps . . . 18

3.1.1.1 Ownership . . . 21

3.1.2 Object Containers . . . 24

3.1.2.1 Containers with Ownership . . . 27

3.1.2.2 Container Size . . . 29

3.1.2.3 Container Free-Lists . . . 32

3.1.3 Thread-local free-list buffer . . . 32

3.1.4 Remote Free-Lists . . . 35

3.1.5 Allocation Buffer . . . 36

3.1.6 Lock-Free Operations . . . 37

3.2 Combining Features . . . 38

3.2.1 Individual Object Headers – No Ownership . . . 40

3.2.1.1 IN . . . 41

3.2.1.2 IN-l . . . 41

3.2.1.3 IN-c . . . 41

3.2.1.4 IN-cl . . . 42

3.2.1.5 IN-r, IN-cr . . . 43

3.2.2 Individual Object Headers – Object Ownership . . . 43

3.2.2.1 IO . . . 43

3.2.2.2 IO-l . . . 44

3.2.2.3 IO-c . . . 44

3.2.2.4 IO-cl . . . 44

3.2.2.5 IO-r, IO-cr . . . 45

3.2.3 Object Containers – No Ownership . . . 45
3.2.3.1 CN . . . 45
3.2.3.2 CN-l . . . 46
3.2.3.3 CN-r . . . 46
3.2.4 Object Containers – Object Ownership . . . 46
3.2.4.1 CO . . . 46
3.2.4.2 CO-l . . . 47
3.2.4.3 CO-r . . . 48
3.3 Summary . . . 48

4 Existing Allocators **49**
4.1 Solaris Malloc . . . 49
4.2 Dlmalloc . . . 50
4.3 Ptmalloc . . . 50
4.4 Hoard Allocator . . . 51
4.5 Streamflow Allocator . . . 52
4.6 Summary . . . 53

5 Test Allocators **54**
5.1 Allocator A: Base Case . . . 54
5.2 Allocator B: Add Thread Heaps . . . 55
5.3 Allocator C: Add Object Containers . . . 56
5.4 Allocator D: Add Object Ownership . . . 57
5.5 Allocator E: Add Restricted Container Movement . . . 58
5.6 Allocator F1: Add Thread–Local Free–List Buffer . . . 59
5.7 Allocator F2: Add Remote Free–Lists . . . 60

5.8 Allocator G: Vary Container Size . . . 60
5.9 Allocator H: Add Lock-Free Operations . . . 61
5.10 Coalescing Allocator . . . 62
5.11 Summary . . . 64

6 Memory Allocator Test Suite **65**
6.1 Single–Threaded Benchmarks . . . 65
6.1.1 P2C . . . 68
6.1.2 GS . . . 68
6.1.3 Espresso/Espresso-2 . . . 68
6.1.4 CFRAC/CFRAC-2 . . . 68
6.1.5 GMake . . . 69
6.1.6 GCC . . . 69
6.1.7 Perl/Perl-2 . . . 69
6.1.8 Gawk/Gawk-2 . . . 69
6.1.9 XPDF/XPDF-2 . . . 70
6.1.10 ROBOOP . . . 70
6.1.11 Lindsay . . . 70
6.2 Multi–Threaded Benchmarks . . . 70
6.2.1 Recycle . . . 71
6.2.2 Consume . . . 71
6.2.3 False–Sharing Micro–benchmarks . . . 71
6.2.4 Larson . . . 72
6.3 Trace Collection . . . 72
6.4 Trace Results . . . 73
6.4.1 Sizes of Requests . . . 74

6.4.2 Lifetimes of Objects . . . 75
6.4.3 Interarrival Times of Allocations and Deallocations . . . 79
6.4.4 Allocation Footprint . . . 79
6.5 Benchmark Selection . . . 80
6.6 Summary . . . 81

7 Memory Allocator Evaluation **82**
7.1 Runtime and Scaling . . . 83
7.1.1 Single-Threaded Benchmarks . . . 84
7.1.2 Recycle . . . 84
7.1.3 Consume . . . 88
7.1.4 False-Sharing Benchmarks . . . 93
7.1.5 Larson . . . 94
7.2 Fragmentation . . . 100
7.2.1 Fragmentation in Single-Threaded Benchmarks . . . 101
7.2.2 Fragmentation in Multi-Threaded Benchmarks . . . 103
7.3 Memory Usage . . . 106
7.3.1 Memory Usage in Single-Threaded Benchmarks . . . 107
7.3.2 Memory Usage in Multi-Threaded Benchmarks . . . 108
7.4 Analysis . . . 110
7.5 Summary . . . 113

8 Conclusions **114**
8.1 Memory–Allocation Challenges . . . 114
8.2 Method of Analysis . . . 115
8.3 Analysis Results . . . 115

8.4 Future Work 117

A Trace Graphs **118**

Bibliography **134**

List of Tables

3.1 Feature Combinations 40

6.1 Allocation Statistics 67
6.2 Runtime Statistics 67
6.3 Size of Requests 74
6.4 Lifetimes 76
6.5 Interarrival Times 78
6.6 Allocation Footprint 80

7.1 Test Setup 83
7.2 Allocator F2 Compared to the Default Allocator 112
7.3 Coalescing Allocator Compared to the Default Allocator 113

List of Figures

1.1 Program Address Space 2

2.1 Memory Allocator Heap 6
2.2 Allocated Object 7
2.3 Internal and External Fragmentation 8
2.4 Fragmentation of Memory 10
2.5 External Fragmentation 10
2.6 Program-Induced False-Sharing 14
2.7 Allocator-Induced Active False-Sharing 14
2.8 Allocator-Induced Passive False-Sharing 15

3.1 Single Heap Allocator 19
3.2 Per-Thread Heaps 19
3.3 Per-Thread Heaps with a Global Heap 21
3.4 Per-Thread Heaps with Ownership 22
3.5 Passive False-Sharing Avoidance 23
3.6 Header Placement 25
3.7 Active False-Sharing using Containers 28
3.8 External Fragmentation Using Object Container Ownership 29

3.9 Example Super-Containers 31
3.10 Free Lists Structures 33
3.11 Thread-Local Free-List Buffer 34
3.12 Remote Free-List 35

5.1 Header and Trailer Structure 63

7.1 Scaling in Recycle on Setup A 85
7.2 Scaling in Recycle on Setup B 87
7.3 Scaling in Recycle on Setup C 89
7.4 Runtime Performance in Consume 91
7.5 Runtime Performance in Consume 92
7.6 Scaling in False-Sharing Benchmark 95
7.7 Scaling in Larson on Setup A 97
7.8 Scaling in Larson on Setup B 98
7.9 Scaling in Larson on Setup C 99
7.10 Fragmentation in Single-Threaded Benchmarks 102
7.11 Fragmentation in Recycle 104
7.12 Fragmentation in Consume 105
7.13 Memory Usage in Single-Threaded Benchmarks 107
7.14 Memory Usage in Multi-Threaded Benchmarks 109

A.1 Bin Size Distribution 120
A.2 Bin Size Distribution 2 121
A.3 Bin Size Over Time 122
A.4 Bin Size Over Time 2 123
A.5 Cumulative Lifetime Distribution 124

A.6 Cumulative Lifetime Distributions 2 125
A.7 Lifetime Over Time 126
A.8 Lifetime Over Time 2 127
A.9 Lifetime Over Time 128
A.10 Interarrival Times Cumulative Distribution 129
A.11 Interarrival Times Cumulative Distribution 2 130
A.12 Allocation Footprint 131
A.13 Allocation Footprint 2 132
A.14 Allocation Footprint 3 133

Chapter 1

Introduction

Multi-processor computers are becoming increasingly popular and are important for improving application performance. However, writing programs that take advantage of multiple processors is not an easy task [Ale01]. For example, shared resources can become a bottleneck for scaling in a multi-threaded program. One typical shared resource is program memory, since it is normally used by all threads in a concurrent program [BMBW00]. Therefore, providing high-performance memory management is important for multi-threaded programs.

1.1 Memory Structure

The virtual-memory address-space for a program is typically divided into distinct zones: static code/data, dynamic allocation, dynamic code/data, and stack, with free memory surrounding the dynamic code/data [Sal]. Figure 1.1 shows a typical layout of these zones.

Static code and data are loaded into memory at load time, and their allocations do not change during runtime. The stack has simple and fixed management in a single-threaded

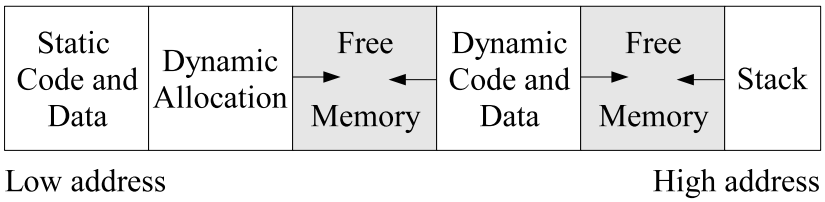


Figure 1.1: Program Address Space

program. In multi-threaded programs, a new stack is created for each new thread. Thread stacks are commonly created in dynamic-allocation memory. Management of dynamic code/data, for example libraries that are loaded at runtime, can be fairly complex especially in a multi-threaded program [HLM06]. However, management of this area is handled by a dynamic loader, and is largely independent of a program, since there is no mechanism to directly affect its behaviour. Therefore, this thesis considers only the management of the dynamic-allocation memory, a very complex area of memory to manage.

1.2 Dynamic-Memory Management

Modern programming languages manage dynamic-allocation memory in different ways. Some languages, such as Java, provide memory management in which data is explicitly allocated, but implicitly deallocated through garbage collection. In general, garbage collection also supports memory compaction, in which dynamic data may be moved during runtime in order to better utilize space. Programming languages such as C and C++, provide the programmer with explicit control over the allocation and deallocation of data. This thesis looks at explicit dynamic-memory management. Garbage collection and compaction are beyond the scope of this thesis.

A memory allocator is responsible for managing dynamic memory. Most programs use a general-purpose memory-allocator, often the one provided by the programming lan-

guage's runtime library. However, high-performance memory allocators for multi-threaded programs are still being designed and improved. C and C++ allow a programmer to replace the memory allocator with an alternative general-purpose memory-allocator. For this reason, several general-purpose allocators have been written for C/C++ with the goal of scaling in a multi-threaded program [SAN06] [BMBW00] [Nak01] [GM]. This thesis looks at the design of high-performance allocators for use by multi-threaded applications written in C/C++.

1.3 Contributions

Several existing memory allocators attempt to achieve good performance in multi-threaded programs. This thesis examines these memory allocators to identify the underlying features they employ to achieve good performance. These features, outlined in Chapter 3, include: per-thread heaps with a global heap, object ownership, object containers, thread-local free-list buffers, remote free-lists, allocation buffers, and lock-free operations. I create several test allocators that differ from each other in terms of these fundamental features. A set of benchmark programs are used to compare the runtime, scalability, and fragmentation of the test allocators in order to identify the effects of each feature. Finally, I select a set of fundamental features that generate a good general-purpose memory-allocator.

1.4 Outline

This thesis is organized as follows. Chapter 2 provides background information on dynamic-memory management. Chapter 3 discusses the design of a multi-threaded memory-allocator. Chapter 4 describes existing allocators and related work. Chapter 5 presents a set of test al-

locators. Chapter 6 describes a test suite for memory allocators using both single-threaded and multi-threaded benchmark programs. Chapter 7 presents results from testing and comparing the different allocators described in Chapters 4 and 5 using the test suite described in Chapter 6. Finally, Chapter 8 provides a summary and some conclusions.

Chapter 2

Memory Allocator Background

When a program dynamically creates a data structure, referred to as an object, it occupies memory in the dynamic-allocation zone. The memory allocator is itself a data structure that handles allocation and deallocation of objects in the dynamic-allocation memory. The dynamic-allocation area grows or shrinks by operating system calls, such as `mmap` or `sbrk`. Dynamic objects are allocated and deallocated by the program through calls such as `malloc` and `free` in C, and `new` and `delete` in C++.

2.1 Components of a Memory Allocator

There are two important parts to a memory allocator: storage data and heap. Storage data reside in dynamic allocation memory, while the heap may reside in dynamic code and data memory. There are three types of storage data: allocated objects, freed objects, and reserved memory. Allocated objects are memory allocated to the program through calls to `malloc` or `new` (other forms exist, but they all funnel through to `malloc`). Freed objects are memory that was allocated to the program, and later deallocated through calls

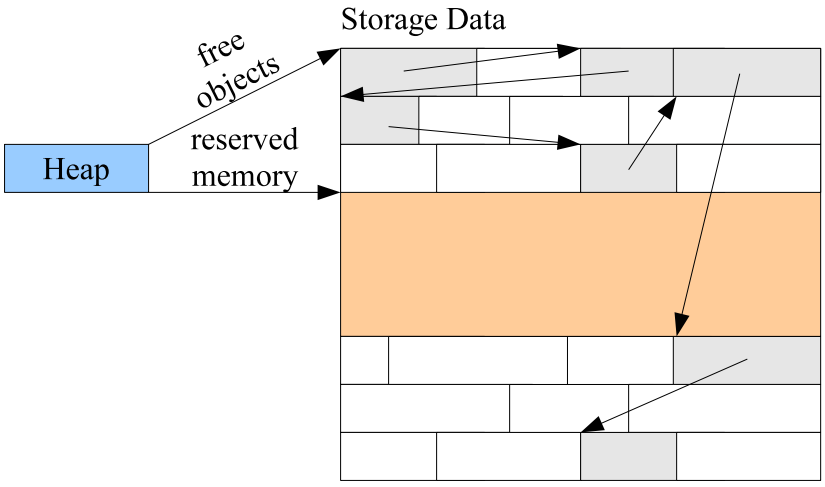


Figure 2.1: Memory Allocator Heap

to `free` or `delete`. Reserved memory is a block of memory that has been obtained from the operating system, through calls such as `mmap` or `sbrk`, but has not yet been allocated to the program. A memory allocator may contain several blocks of reserved memory.

The second important component of the memory allocator is the heap. The heap is a data structure that is located at a known memory address, and manages freed objects and reserved memory. Allocated objects are generally maintained by the program. Figure 2.1 shows an example heap and its associated storage data. The heap points to reserved memory and the freed objects in the heap. Each freed object in the heap, shown in grey, usually points to the next freed object in the heap. The heap data-structure contains all information necessary to manage the storage data of the heap.

Allocated and freed objects are typically surrounded by additional management data through the use of headers and trailers. Object headers and trailers contain information regarding the object, such as the object size, and are located before and after the object in memory. A free object may also hold additional information in the object space, but that information may be lost once the object is allocated to the program. Object trailers are

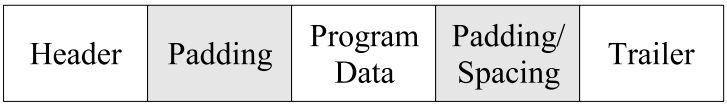


Figure 2.2: Allocated Object

sometimes used for security purposes to signify the end of an object, or to simplify some allocation algorithm implementations. Objects can also be padded either before or after the object, to ensure proper alignment. Some algorithms may require that a larger space be allocated to the program than the program requests, leaving additional spacing after the object. Padding and spacing are reserved memory around an allocated object that cannot be used to satisfy a future allocation request while the current allocation exists. Figure 2.2 shows an allocated object with a header, trailer, and some padding and spacing around the object. A free object may contain additional memory-management data instead of program data.

2.2 Single-Threaded Memory Allocators

A single-threaded memory allocator does not actually run any threads itself, but is used by a single-threaded program. Because the memory allocator code is only executed by the single program thread, issues of synchronization and mutual exclusion are avoided; however, there are two issues in designing a single-threaded memory allocator: fragmentation and locality.

2.2.1 Fragmentation

Fragmentation is wasted space in memory. Wasted space is memory requested from the operating system, but not used by the program. Fragmentation can take one of two forms:

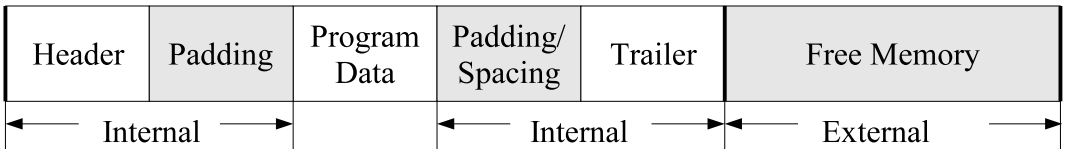


Figure 2.3: Internal and External Fragmentation

internal or external.

Internal fragmentation is memory space that is allocated to the program, but is not intended to be accessed by the program, such as headers, trailers, padding, and spacing around an allocated object. Internal fragmentation is typically memory that is used by the allocator for management purposes or required by the architecture for correctness (e.g., alignment).

There are two definitions for external fragmentation: memory space that is unusable for a given allocation request (because it is too small for example), or all memory space reserved from the operating system but not allocated to the program [WJNB95] [Sie00] [LPB98]. In this thesis, the second definition is used since it encompasses both definitions. Using this definition, external fragmentation includes reserved memory and freed objects with their management data.

Figure 2.3 shows an example section of memory outlining internal and external fragmentation. The header, padding, spacing, and trailer are internal fragmentation used by the allocator to store information, to provide security, or to fulfill implementation requirements. The program data is not fragmentation. Free memory is external fragmentation. The free memory may contain freed objects (including their headers, trailers, and padding/spacing) and reserved memory.

Internal fragmentation can be problematic when the space required to manage an object is a significant proportion of the allocated object. For example, if a header is as large as

the object being managed, then the memory usage for that object is doubled. An allocator should strive to keep management information to a minimum.

External fragmentation can be problematic in two ways: heap blowup and highly fragmented memory. Heap blowup occurs when memory freed by the program is not reused for future allocations, leading to potentially unbounded external fragmentation growth [BMBW00]. Heap blowup can occur due to allocator policies that are too restrictive in reusing freed memory.

Memory can become highly fragmented after multiple allocations and deallocations of objects. Figure 2.4 shows an example of how a small block of memory can become fragmented as objects are allocated and deallocated, where white areas are objects allocated to the program, and grey areas are freed objects. Blocks of free memory become smaller and non-contiguous making them less useful in serving allocation requests. Memory is highly fragmented when the sizes of most free blocks are unusable. For example, 2.5(a) and 2.5(b) have the same quantity of external fragmentation, but 2.5(b) is highly fragmented. If there is a request to allocate a large object, 2.5(a) is more likely to be able to satisfy it with existing free memory, while 2.5(b) would likely have to request more memory from the operating system.

In a single-threaded memory allocator, there are a number of allocation algorithms that can be used to control fragmentation [JW99]. Sequential-fit algorithms maintain one list of free objects that is searched for a block that is large enough to fit a requested object size. Different policies determine which free object is selected, for example the first free object that is large enough, or a free object that is closest to the requested size [JW99].

A segregated or binning allocation algorithm uses a set of bin sizes. The heap maintains a set of lists of freed objects, each of a different bin size. When an object is allocated, the requested size is rounded up to the nearest bin size resulting in spacing around the

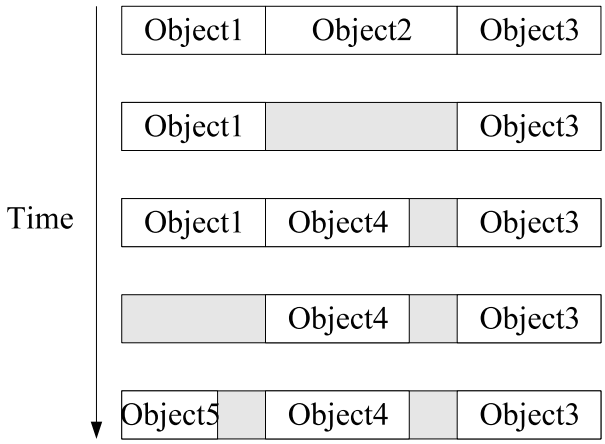
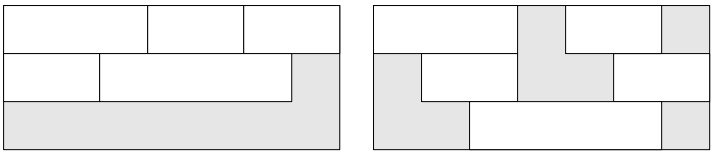


Figure 2.4: Fragmentation of Memory



(a) Contiguous (b) Highly Fragmented

Figure 2.5: External Fragmentation

object. The binning algorithm is very fast at finding free memory of the appropriate size, since the first free object on the free list for that size is used. The fewer bin sizes there are, the fewer lists need to be maintained by the heap; however, the bin sizes are less likely to closely fit the requested object size, leading to more internal fragmentation. The more bin sizes there are, the less likely free objects are to be reused, leading to more external fragmentation and potentially heap blowup.

A variation of the binning algorithm allows objects to be allocated to the requested size, but when an object is freed it is placed on the free list of the next smallest or equal bin size [JW99]. For example, with bin sizes of 8 and 16 bytes, a request for 12 bytes allocates 12 bytes, but when the object is freed, it is placed on the 8 byte bin list. When later

allocation requests are made, the bin free-lists contain objects of different sizes, ranging from one bin size to the next (8-16 in this example), and a sequential-fit algorithm is used to find an object large enough for the requested size.

A third algorithm is the buddy system. The buddy system makes use of splitting and coalescing. When an object is deallocated it is coalesced with the objects immediately before and after it in memory, if they are free. Coalescing the objects turns them into one larger object. When an object is allocated, if there are no free objects of the requested size, a larger free object may be split into two smaller objects to satisfy the allocation request without obtaining more memory from the operating system.

Using the buddy system, a block of dynamic allocation memory is split into two equal chunks, one of those chunks is again split into two equal chunks, and so on until a block just large enough to fit the requested object is created. Similarly, a chunk may be coalesced with its other half, if they are both completely free, to create a large enough area to satisfy an allocation request [JW99].

Splitting and coalescing can be used with other algorithms to avoid highly fragmented memory. Coalescing does not immediately reduce external fragmentation. However, coalesced blocks of memory are more likely to be useful in future allocations, avoiding external fragmentation growth.

2.2.2 Locality

The principle of locality recognizes that programs tend to reference a small set of data, called a working set, for a certain period of time [Den05]. There are two types of locality: temporal and spatial. If an object is accessed, temporal locality suggests that same object will be accessed again within a short time period, while spatial locality implies that a nearby address is also likely to be accessed within a short time period [Den05] [Wil]. Temporal

locality commonly occurs due to loops in a program, while spatial locality commonly appears when accessing arrays of related data [Den05].

Hardware takes advantage of spatial and temporal locality through caching. When an object is accessed, the memory physically located around the object is also cached with the expectation that the current and nearby objects will be referenced within a short period of time. For example, entire virtual memory pages are brought into memory from disk, and entire cache lines are brought into cache. A program exhibiting good locality has better performance due to fewer cache misses and page faults.

Temporal locality is dependent on the program, while spatial locality is determined by the memory allocator [FB05]. An allocator providing optimal spatial locality places objects that are used together close by in memory, such that the working set of the program fits into the fewest possible pages and cache lines. However usage patterns are different for every program. Hence, no general-purpose memory-allocator can provide perfect locality for every program, but an allocator can try to avoid degrading locality.

One way a memory allocator can degrade locality is by increasing the working set. For example, a memory allocator may access several objects before finding a free object to satisfy an allocation request. If there are a large number of objects accessed in very different areas of memory, the allocator may cause several cache or page misses [GZH93]. Another way locality may be degraded is by spatially separating related data. For example, in a binning allocator, objects of different sizes are allocated from different bins that may be located in different pages of memory.

2.3 Multi-Threaded Memory Allocators

When referring to a multi-threaded allocator, it is not the allocator that is multi-threaded, but the program that uses it. The allocator code may be accessed by multiple program threads at any given time. In addition to locality and fragmentation issues, there are issues of mutual exclusion, false sharing, and heap blowup.

2.3.1 Mutual Exclusion

Mutual exclusion provides sequential access to a shared resource. In a memory allocator, the heap is a shared resource to which access must be controlled using mutual exclusion. There are two performance drawbacks to mutual exclusion. The first is the overhead necessary in performing a hardware atomic operation every time the shared resource is accessed. The second drawback arises when multiple threads contend for a shared resource simultaneously, since some threads may be unable to continue until the resource is released. Contention can be reduced through fine-grained locking.

2.3.2 False Sharing

False sharing can lead to cache thrashing. It occurs when two or more objects that are each used by a different thread share a cache line, assuming each thread runs on a different processor with its own cache. Each time one thread modifies its object, the other thread's associated cache is invalidated, even though it is uninterested in the modified object. There are three types of false sharing: program induced, allocator-induced active, and allocator-induced passive.

Program-induced false-sharing occurs when one thread passes one of its objects to another thread as in Figure 2.6. If that object came from a cache line with other objects

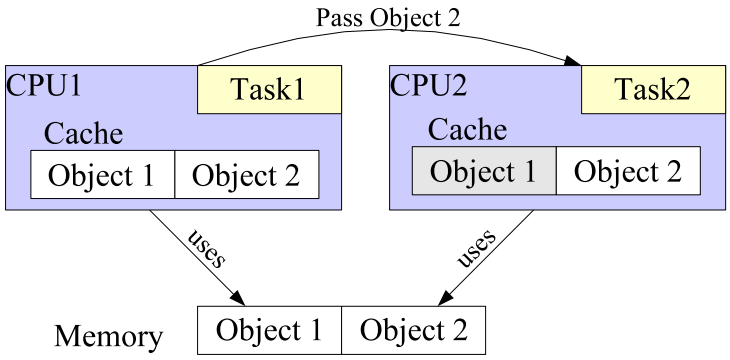


Figure 2.6: Program-Induced False-Sharing

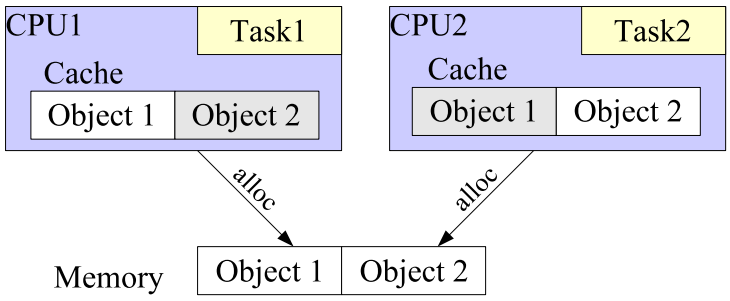


Figure 2.7: Allocator-Induced Active False-Sharing

used by the first thread, then the two threads now share a cache line. When Task1 passes Object2 to Task2, they are in a false-sharing situation. Changes to Object1 invalidate CPU2's cache line, and changes to Object2 invalidate CPU1's cache line.

Allocator-induced active false-sharing occurs when an allocator allocates objects that fall in the same cache line to different threads as shown in Figure 2.7. Each thread allocates an object and loads a cache line size of memory into its associated cache. To keep the cache consistent, any changes to the cache line by one processor invalidate the cache line for all processors with the same memory in their cache.

Passive false-sharing is another form of allocator-induced false-sharing that is caused by program-induced false-sharing. When an object in a program-induced false-sharing

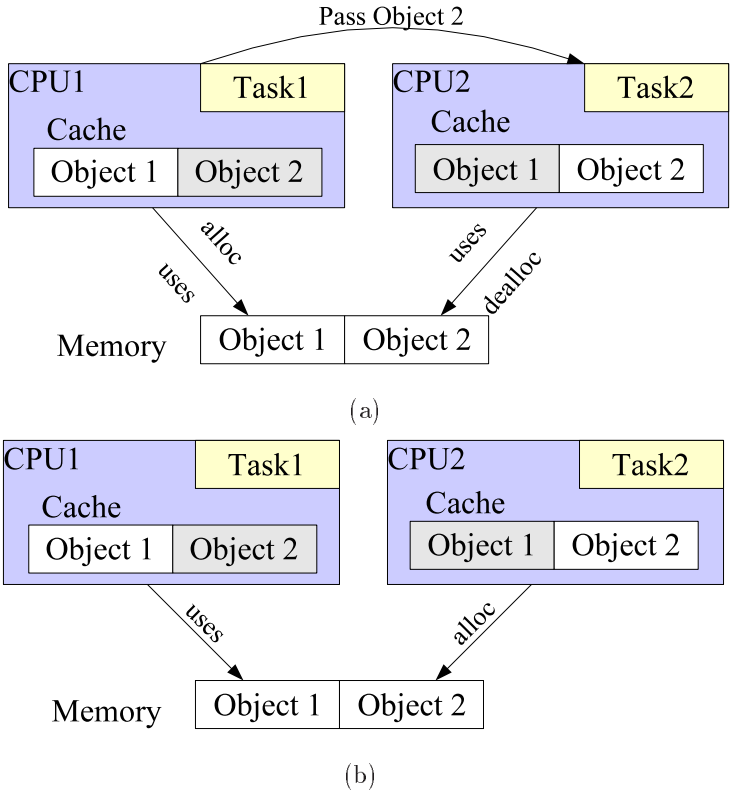


Figure 2.8: Allocator-Induced Passive False-Sharing

situation is deallocated, a future allocation of that object may cause passive false-sharing. In Figure 2.8(a), Task2 deallocates Object2, passed to it by Task1, leaving it free for a future allocation request by Task2. Allocator-induced passive false-sharing occurs when Object2 is allocated to Task2 while Task1 still uses Object1 (as in 2.8(b)).

2.3.3 Heap Blowup

The third issue in memory allocation for a multi-threaded program is an additional form of heap blowup. Heap blowup is the failure to reuse free memory, leading to unbounded external fragmentation [BMBW00]. In a multi-threaded program, heap blowup can occur

when memory freed by one thread is inaccessible to other threads due to the allocation strategy [GPT04].

Chapter 3

Memory Allocator Design

The previous chapter describes a number of challenging issues when designing a memory allocator. This chapter looks at several features found in existing allocators that address these issues. These features are then considered in different combinations to find potential candidate allocators for evaluation.

3.1 Multi-Threaded Memory-Allocator Features

The following features may be present in a memory allocator,

1. Per-thread heaps, but including a global shared-heap to avoid heap blowup
   (a) with or without ownership
2. Object Containers
   (a) with or without ownership
   (b) fixed or different sized

   (c) global or local free-lists

3. Thread-local free-list buffer

4. Remote free-list

5. Allocation buffer

6. Lock-free operations

The first feature, per-thread heaps, looks at different types of heaps. The second feature, object containers, looks at the organization of objects within the storage area. The remaining features can be applied to different parts of the allocator design or implementation.

3.1.1 Per-Thread Heaps

A multi-threaded allocator may use one single heap, or multiple heaps with or without a global shared-heap. A single-heap allocator consists of one heap from which objects are allocated and to which objects are freed. Memory is allocated from the freed objects in the heap or from the operating system. The heap may also occasionally return freed objects to the operating system. Figure 3.1 illustrates a multi-threaded program using a single-heap allocator. The running threads and the single shared heap are shown. The arrows indicate the direction in which memory conceptually moves for each type of operation. This type of allocator is essentially a single-threaded allocator, but with appropriate locking to provide mutual exclusion to this shared resource. Whether using a single lock for all heap operations, or fine-grained locking on different heap operations, the single heap may still be a significant source of contention.

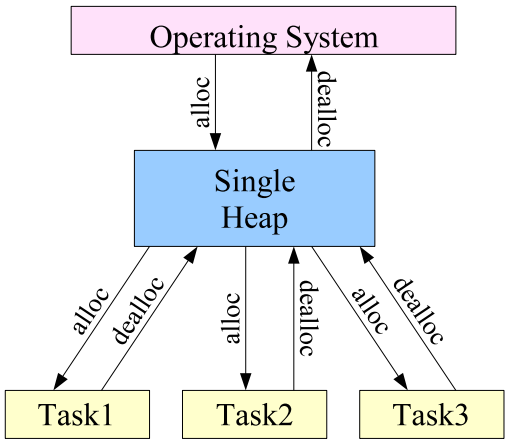


Figure 3.1: Single Heap Allocator

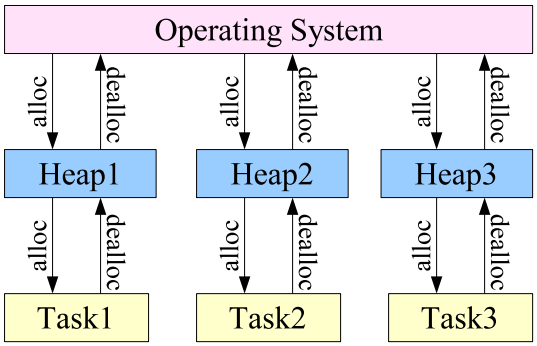


Figure 3.2: Per-Thread Heaps

In order to significantly reduce contention in a multi-threaded program, multiple heaps are used. Having fewer heaps than threads, while reducing contention, does not allow for the removal of locks since more than one thread may access a heap at a time. Since the behaviour of the program cannot be predicted, a worst case scenario is possible where all allocations occur to the same heap. Having more heaps than threads may be redundant if the heaps all behave the same. Later discussion shows cases in which having more heaps than threads can be beneficial. However, as a starting point, the strongest case for multiple heaps is to have a single heap per thread, as in Figure 3.2.

Per-thread heaps provide increased control of the memory being allocated to each thread. Using a one-to-one mapping of threads and heaps, each thread only allocates from its heap, which improves locality since all objects for a thread may be allocated from the same area in memory. For example, in a program where each thread allocates, uses, and deallocates its own objects, a single heap allocator may spread the objects of each thread over a large area of memory, but a per-thread heap allocator can allocate each thread's objects in a smaller area of memory, better utilizing each CPUs cache and causing each thread to access fewer pages.

Per-thread heaps also cause an increase in external fragmentation and may lead to heap blowup. The external fragmentation experienced by a program with a single heap is now multiplied by the number of threads, since each heap must allocate its own area of reserved memory. Additionally, objects freed by one heap cannot be reused by other threads causing heap blowup. In the worst case, a program in which objects are deallocated to one set of thread heaps, but allocated from a different set of thread heaps would mean freed objects are never reused.

A global shared-heap, shown in Figure 3.3, is often used to prevent heap blowup. A global shared-heap is not used directly by any thread, but is used to move free memory among thread heaps. When a thread heap reaches a certain threshold of free objects, it frees some of those objects to the global heap to be reused by another thread heap. Similarly, the global shared-heap may free memory to the operating system when it reaches a certain threshold. Memory can be allocated from the operating system either to the thread heaps or the global heap. However, since any thread may free or allocate objects from the global heap, the global heap is a shared resource that requires locking.

When a thread completes, there are two options of how to handle its thread heap. One option is to free all objects on the thread heap to the global heap and destroy the thread

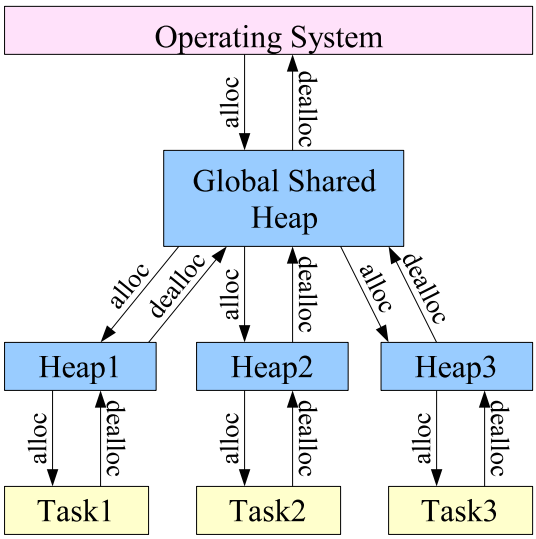


Figure 3.3: Per-Thread Heaps with a Global Heap

heap, while a second option places the thread heap on a list of available heaps and reuses it for a new thread that starts up in the future. Destroying the thread heap immediately may reduce external fragmentation sooner, since all free objects are freed to the global heap and may be reused by other threads. Alternatively, reusing thread heaps may improve performance if the inheriting thread makes similar allocation requests as the thread that previously held the thread heap.

Although contention is reintroduced with the global heap, the cost is minimal since most allocator operations should complete without the use of the global heap. As per-thread heaps are a key feature for a multi-threaded allocator, all further discussion assumes per-thread heaps with a global shared-heap to prevent heap blowup.

3.1.1.1 Ownership

Ownership is an option that is possible with per-thread heaps. Ownership is the notion that an object is owned by the thread that allocates it. Since there is a one-to-one corre-

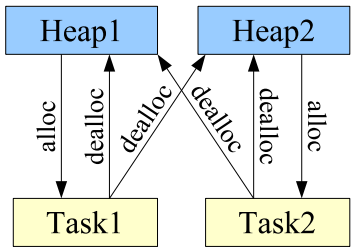


Figure 3.4: Per-Thread Heaps with Ownership

spondence between threads and heaps, an object is simultaneously owned by a thread and its heap.

Without ownership, a task only frees objects to its own heap, as shown in Figure 3.2. This approach means thread heaps are private to their owner thread and do not require any locking. A drawback of per-thread heaps without ownership is that if an object is passed from one thread to another during program execution, passive false-sharing may occur. For example, if task A passes an object to task B, and task B frees the object, then the object is freed to task B's thread heap. As a result, a future allocation request may lead to passive false-sharing, as described in Section 2.3.2.

With ownership, every object must be deallocated to the heap that it was allocated from. This requirement means that heaps are no longer private to a single thread and require locks to provide consistency, since any thread may deallocate an object to its owner heap. Figure 3.4 shows an example of per-thread heaps with ownership (minus the global heap).

The benefit of ownership is the elimination of allocator-induced passive false-sharing by returning an object to its owner thread so that it can never be allocated to another thread. In general, all allocator-induced false-sharing can be eliminated by designating an area of memory to one thread heap, and ensuring that area of memory is always allocated to one thread. For example, assuming that page boundaries coincide with cache line boundaries,

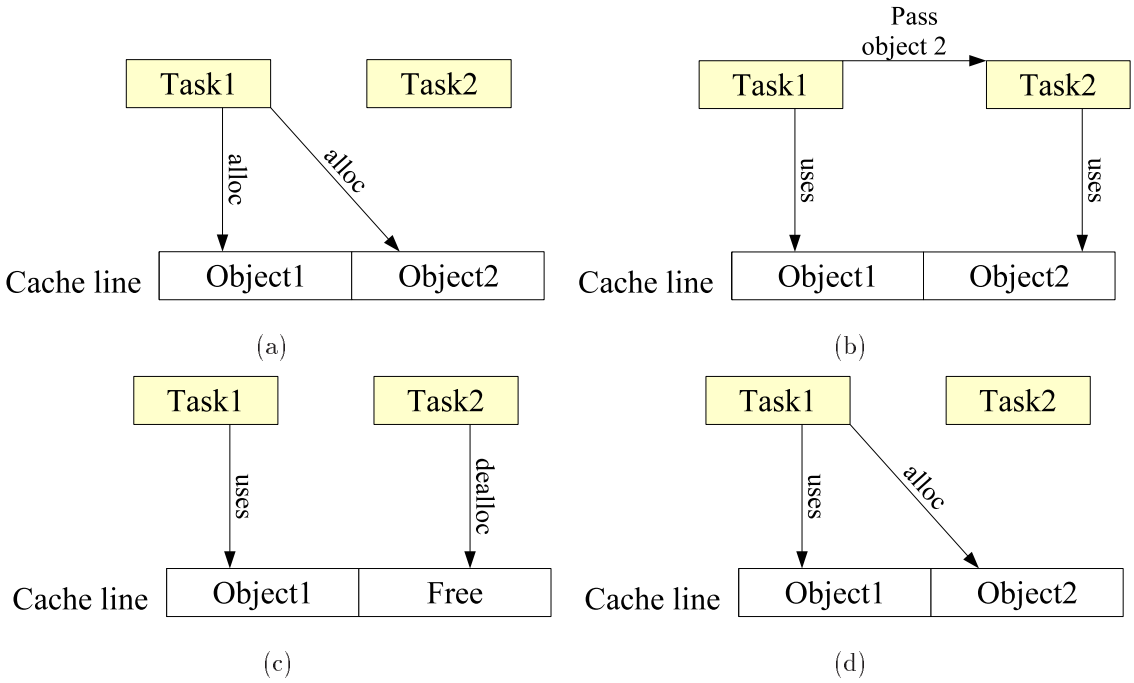


Figure 3.5: Passive False-Sharing Avoidance

designating a page to a thread heap prevents allocator-induced false-sharing since no two threads are allocated memory from the same page. In Figure 3.5, one thread allocates two pieces of memory that fall in the same cache line. False sharing can only occur when one thread passes an object to another, as in part b. However, if that second thread deallocates the memory, ownership requires the object be returned to the original thread heap. Thus, subsequent allocations allocate the object to the original thread preventing any allocator-induced false-sharing.

Object ownership can be enforced as immediate or delayed ownership. Deallocated objects may be returned to the owner thread immediately or after some delay. For example, a thread may store an object it does not own on its free list for a certain number of memory operations. The thread heap may allow these objects to be reallocated to the local thread

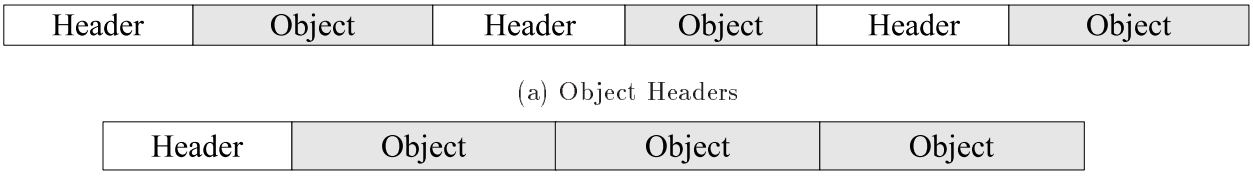
or not. If delayed object ownership is used such that it allows reallocation by the local thread, then some passive false-sharing may occur. For example, in Figure 3.5(c), Object2 may be deallocated to Task2's thread heap initially. If Task2 requests an object before Object2 is returned to its owner, then the allocator may allocate Object2 to Task2 causing passive false-sharing to occur.

Delayed ownership with reallocation can improve performance since the local thread can complete some operations on its own thread heap where it might otherwise be required to go to the global heap. Delayed ownership without reallocation can improve performance by batching together free operations to a remote thread-heap.

3.1.2 Object Containers

A simple allocator places headers/trailers with individual objects, meaning memory adjacent to the object is reserved for object management information, as shown in Figure 3.6(a). However, this approach leads to poor cache usage, since only a portion of the cache line is holding useful information from the program's perspective. Spatial locality is also negatively affected; even though the header and object are together in memory, they are generally not accessed together. The object is accessed by the program when it is allocated, while the header is accessed by the allocator when the object is free. This difference in usage patterns can lead to poor cache locality [FB05]. Additionally, placing headers on individual objects can lead to redundant management information. For example, if a header stores only the object size, then all objects with the same size have identical headers. A more complex approach places the headers in a separate location in memory. The complexity lies in finding the object header given only the object address, since that is normally the only information passed to the deallocation operation.

One approach to separating object headers/trailers from object content is to use object



(a) Object Headers

(b) Object Container

Figure 3.6: Header Placement

containers [FB05]. An object container is a group of adjacent objects in memory, shown in Figure 3.6(b). The header for the container holds information necessary for all objects in the container. A trailer may also be used at the end of the container.

In general, the container header/trailer for any object must be found solely from the address of the object. One way to do this is to start containers on aligned addresses in memory, then truncate the lower bits of the object address to obtain the header address (or round up and subtract the trailer size to obtain the trailer address). For example, if an object at address 0xFC28 EF08 is freed and containers are aligned on 64 KB (0x0001 0000) addresses, then the container header is at 0xFC28 0000.

In general, containers contain homogeneous objects, with fixed information in the header, which is logically distributed across all container objects (e.g., all objects are the same size). Containers with heterogeneous objects implies different headers describing them, which introduces the problem of locating a specific header solely by an address. A couple of solutions can be used to implement containers with heterogeneous objects. However, the problem with allowing objects of different sizes is that the number of objects, and therefore headers, in a single container is unpredictable.

One solution allocates objects at one end of the container, while allocating headers from the other end of the container, until the objects meet the headers and the container is filled. Freed objects cannot be split or coalesced since this would cause the number of

headers to change. The difficulty in this strategy remains finding the header for a specific object. The individual headers in the container would have to be searched until the header for a given object is found.

A second solution combines the use of container headers and individual object headers. Each object header stores the heterogeneous information of the object, such as its size, while the container header stores the homogeneous information, such as the owner thread when using ownership. This approach allows containers to hold different types of objects, but does not separate headers from their objects. The benefit of the container in this case is to reduce some redundant information that is stored in the container header.

In general, the complexity of heterogeneous objects in a container is likely to outweigh the potential benefits. A container header is most efficient when all objects in the container are homogeneous and therefore the same size; only one size is stored in the header, making the header a constant size regardless of the number of objects in the container. This approach greatly reduces internal fragmentation since far fewer headers are required. Using homogeneous object containers, each cache line can hold more objects, since the objects are closer together due to the lack of headers among them.

An additional benefit to object containers is that they can be used to avoid allocator-induced active false-sharing. Similar to the approach described in Section 3.1.1.1, if container boundaries coincide with cache-line boundaries and all objects in a container are allocated to the same thread, then allocator-induced active false-sharing is avoided.

Two drawbacks remain when using containers with homogeneous objects. Although similar objects are close spatially within the same container, different objects are further apart in separate containers. Depending on the program, this may or may not improve locality. If the program uses several objects of the same size in its working set, then locality is improved since these objects may all be in the same container. If a lot of

different sized objects are used, then a lot of containers are in use, which leads to poor paging locality, since each container corresponds to another page that needs to be stored in memory. The second drawback is that external fragmentation may be increased since containers reserve space for objects that may never be allocated by the program. However, external fragmentation can be reduced by using smaller containers.

3.1.2.1 Containers with Ownership

Using containers without ownership, objects are deallocated to the thread heap that frees the object. Thus, different objects in a container may be on different thread-heap free-lists. When a thread heap frees objects to the global heap, individual objects are passed, further separating objects from other objects in their container.

Using object ownership, all objects in a container belong to the same thread heap. Ownership of an object is determined by the owner of its container. In general, ownership avoids passive false-sharing since objects are returned to the thread that allocated the object. Passive false-sharing may still occur, as described in Section 3.1.1.1, if delayed ownership is used. As described in Section 3.1.2, using containers avoids active false-sharing since objects in a container are all allocated to the same thread.

Additionally, when a thread heap reaches its threshold of free objects, it moves some containers to another thread heap via the global heap. When a container changes ownership, the ownership of all objects within it change as well. Moving a container involves moving all objects on the thread heap's free-list in that container to the new owner. This approach reduces contention for the global heap, since each request for objects from the global heap returns a container of several objects rather than individual objects.

Additional restrictions may be applied to the movement of containers. When a container changes ownership, if some of its objects are in use by the program, active false-sharing

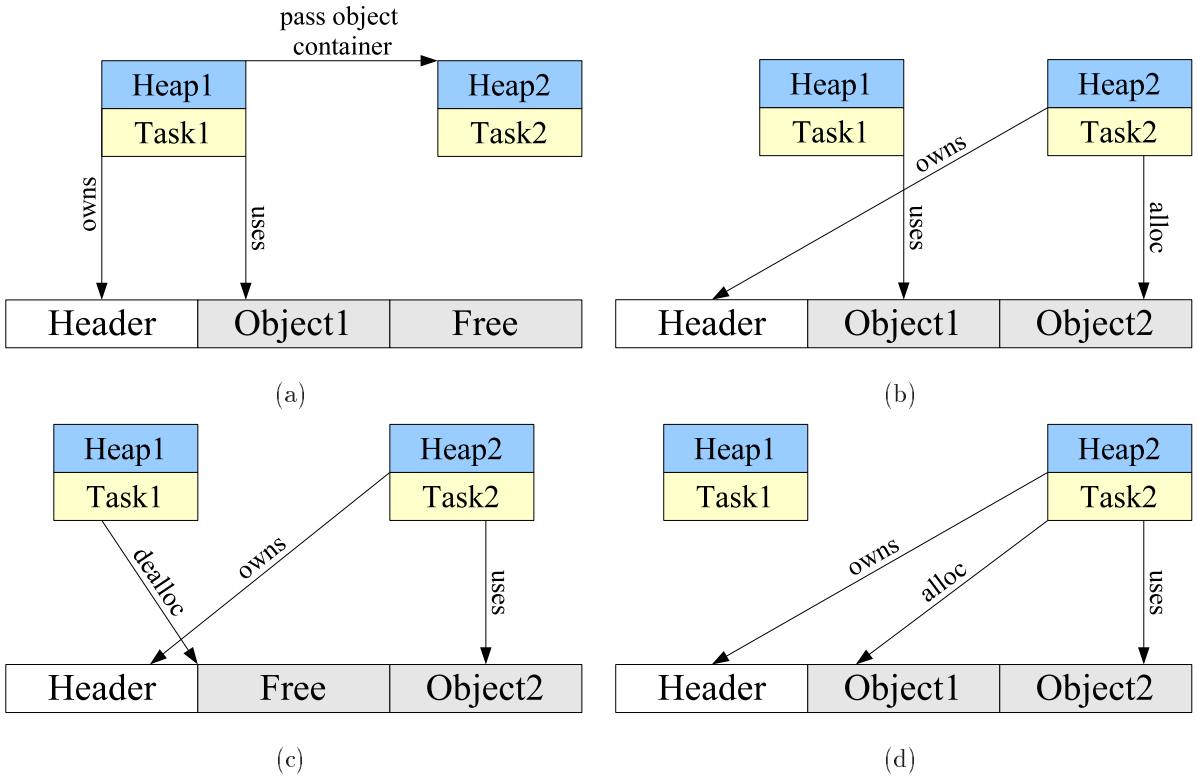
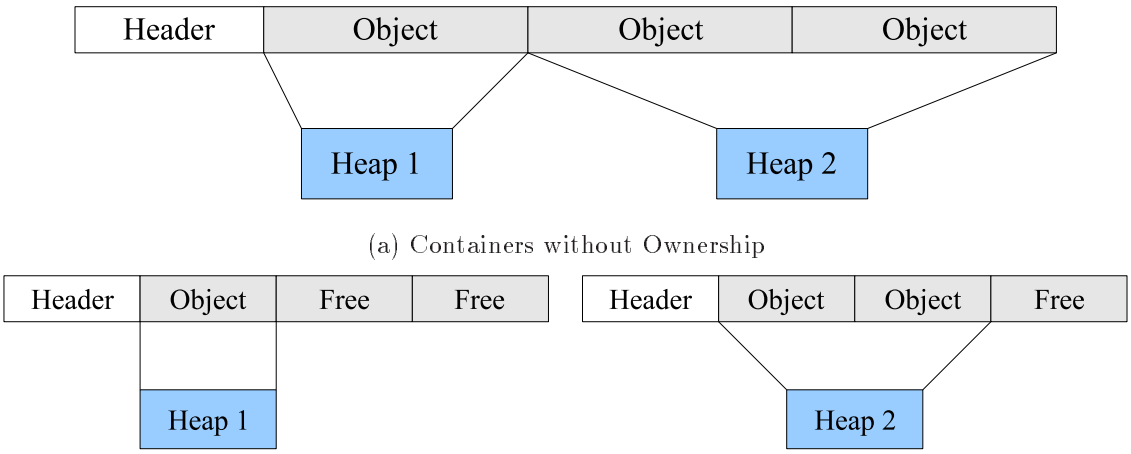


Figure 3.7: Active False-Sharing using Containers

may occur, as demonstrated in Figure 3.7. In 3.7(a), a container is moved from Heap1 to Heap2. When Task2 allocates an object from the container it is in a false-sharing situation, as in 3.7(b). This scenario is an example of active false-sharing since no objects are passed among threads. Note, once the object is freed by Task1 in 3.7(c), no more false sharing can occur until the container changes ownership again. To prevent this form of false sharing, container movement may be restricted to when all objects in the container are free.

A consequence of ownership is that free objects in a container are on the same heap, making it easier to determine if all objects in a container are free. In addition to using the global heap, this information leads to two additional approaches of preventing heap blowup. One approach returns the container to the operating system assuming the container was



(a) Containers without Ownership

(b) Containers with Ownership

Figure 3.8: External Fragmentation Using Object Container Ownership

allocated using a call like `mmap`, which allows memory at an arbitrary address to be returned. A second approach to avoiding heap blowup clears the container so it can be used to allocate objects of a new size.

Using containers with ownership increases external fragmentation since a new container for a requested object size must be allocated separately for each thread requesting it. In the example shown in Figure 3.8, using object ownership allocates 50% more space than required.

3.1.2.2 Container Size

One way to control the external fragmentation caused by allocating a large container for a small number of requested objects is to vary the size of the container. As described earlier, container boundaries need to be aligned on addresses that are a power of two to allow easy location of the header (by truncating the bits). Aligning containers in this manner also determines the size of the container. However, the size of the container has different

implications on the allocator.

The larger the container, the fewer containers are needed, and hence, the fewer headers need to be maintained in memory, improving both internal fragmentation and potentially performance. However, with more objects in a container, there may be more objects that are not allocated, increasing external fragmentation. With smaller containers, not only are there more containers, but a second new problem arises where some objects are larger than the container.

In general, large objects are allocated directly from the operating system and are returned immediately to the operating system to reduce external fragmentation due to infrequent large objects that are unlikely to be reused. If the container size is decreased, for example to 1 KB, then an object that is 1.5 KB is treated as a large object, which is likely to be inappropriate. Thus, it would be ideal to use smaller containers for smaller objects, and larger containers for medium objects, which leads to the issue of locating the container header.

In order to find the container header when using different sized containers, a container superstructure, or super-container is used. The super-container is a container of object containers, as shown in Figure 3.9, that starts on an aligned address. The super-container spans several containers, and contains a header with information for finding each container header. Super-container headers are found using the same method that is used to find container headers when the containers are fixed sizes, by dropping the lower bits of an object address. In the example shown in Figure 3.9, the header of a 64 KB super-container points to the headers of the containers within it. Smaller objects are held within 16 KB containers, while medium objects are held within 64 KB containers. The free space at the end of a super-container can be used to allocate a new container for small objects when another small container is needed.

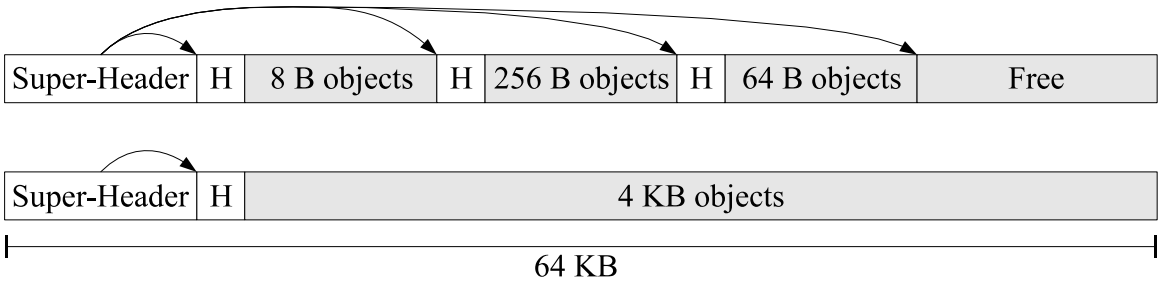


Figure 3.9: Example Super-Containers

The containers within a super-container may be different sizes or all the same size. If the containers in the super-container are different sizes, then the super-container header must perform a search to determine the specific container for an object given its address. If all containers in the super-container are the same size, then a specific container header can be found by an O(1) calculation.

Minimal internal and external fragmentation is achieved by having as few containers as possible, each being as full as possible. It is also possible to achieve additional benefit by using larger containers for popular small sizes, since when fewer containers are used, there are fewer container headers in memory. However, it is impossible for an allocator to determine which sizes are going to be popular in future requests. Keeping statistics on requested sizes may allow the allocator to make a dynamic decision about which sizes are popular. For example, after receiving a number of allocation requests for a particular size, that size is considered a popular request size and larger containers are allocated for that size. However, the decision may be incorrect, leading to a larger container being allocated that remains mostly unused. A programmer may be able to inform the allocator about popular object sizes in order to select an appropriate container size for each object size.

3.1.2.3 Container Free-Lists

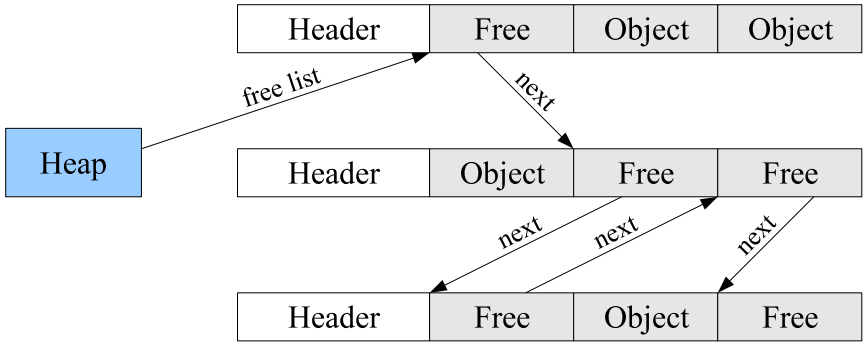
Besides the size of the objects in the container, a container header may hold other useful information that may improve performance. For example, maintaining free lists in a container header (Figure 3.10(b)), rather than in the heap (Figure 3.10(a)), can greatly reduce the complexity of moving all freed objects belonging to a container onto another heap.

Maintaining free lists within container headers assumes all free objects in the container are on the same heap. Thus, it only applies to containers that also enforce ownership. To move a container with free lists on heaps, as in Figure 3.10(a), the heap's free list is first searched to find all objects within the container. Each object is then removed from the free list and linked together to be moved to the new heap. With free lists in containers, as in Figure 3.10(b), the container is removed from the heap's free list and placed on the new heap's free list. Thus, when using free lists within containers, the operation of moving containers is reduced from O(n) to O(1). The cost is adding information to a header, which increases the header size, and therefore internal fragmentation.

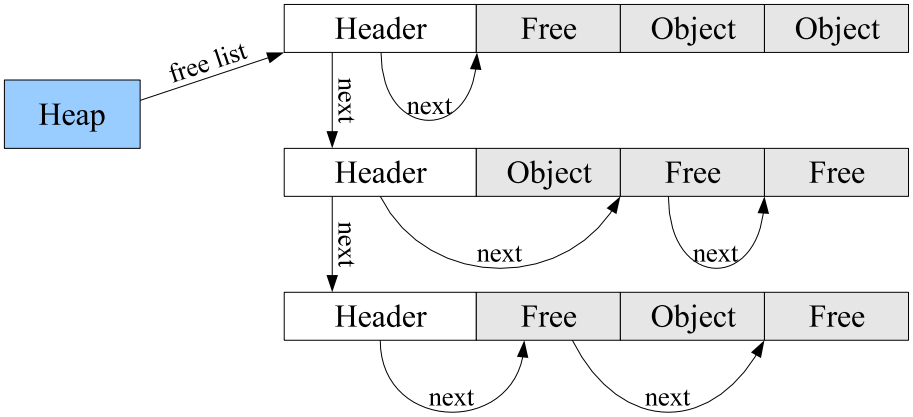
When all objects in the container are the same size, a single free list is sufficient. However, when the objects in the container can be of any size, the header needs to store a free list for each size class when using a binning allocation algorithm, which can be a very significant increase in the container-header size. The alternative is to use a different allocation algorithm with a single free list, such as a sequential-fit allocation-algorithm.

3.1.3 Thread-local free-list buffer

A thread-local free-list buffer contains lists of freed objects. It is a private heap containing only memory that has been freed by its owner thread, as shown in Figure 3.11. It is private in that only the owning thread may access the buffer. The buffer may be used



(a) Free List Among Containers



(b) Free List Within Containers

Figure 3.10: Free Lists Structures

in an allocator with per-thread heaps or a single-shared heap. Placing the buffer in an allocator with only a single-shared heap generates a simple version of private per-thread heaps. However, that type of allocator is not considered in this discussion. The thread-local buffer reduces contention for a shared heap. Allocation and deallocation requests that can be completed from the thread-local buffer avoid locking. However, when the buffer is cleared, it requires obtaining a lock, and depending on the implementation of the thread heap, the operation is either O(1) if it is as simple as adding the list to the end of the

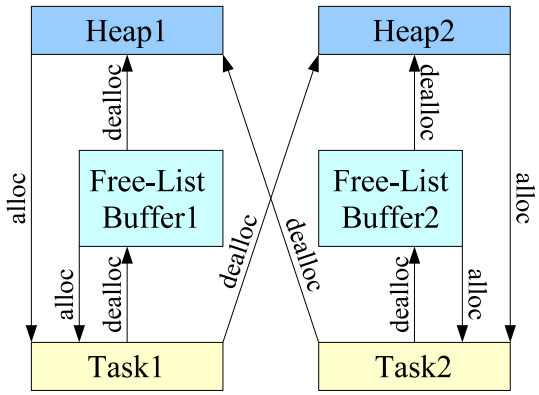


Figure 3.11: Thread-Local Free-List Buffer

thread-heap's free-list, or O(n) if some management needs to be done for each object that is freed.

The objects on the lists may or may not be owned by the local thread-heap depending on the implementation. Figure 3.11 shows an example allocator in which objects owned by other threads are immediately freed to their owner heap, enforcing immediate ownership. The thread-local buffer can also be used to implement delayed ownership. Placing objects that are owned by other threads on the buffer temporarily allows the thread to reuse an object before returning it to its owner.

For a private heap with no ownership, where objects are freed to the thread-heap that deallocates them, the thread-local free-list buffer gains no benefit, since it is essentially the same as the thread heap. However, it may still improve performance if thread-heap operations require more complexity than a simple operation on the buffer. There may also be some performance benefit in storing objects owned by other threads to be freed to their owner heap all at once. The buffer may or may not allow these objects to be reused by the local thread depending on the type of ownership enforced.

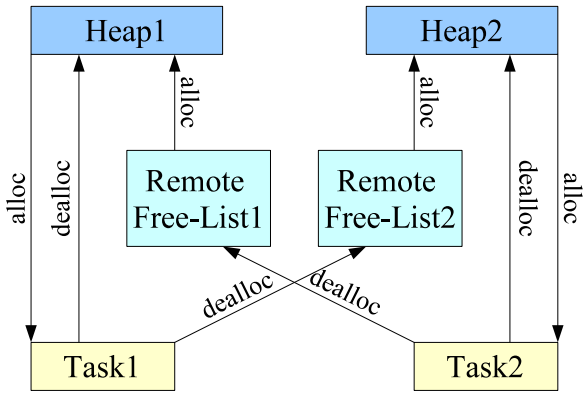


Figure 3.12: Remote Free-List

3.1.4 Remote Free-Lists

A remote free-list is a list of freed objects. Figure 3.12 shows how a remote free-list is used in an allocator. When objects allocated by one thread are deallocated by another, rather than locking the thread heap of the thread owning the object to perform a deallocation, the object is placed on the heap's remote free-list. Objects deallocated by the thread that allocated them can be freed directly to the owner's heap. To avoid heap blowup, the heap with the remote free-list must reuse those free objects before obtaining additional memory.

A remote free-list can reduce contention for a thread heap. Rather than allowing any thread to free to the thread heap, other threads use the remote free-list. Locks are moved from the thread heap to the remote free-list improving the time for local allocations and deallocations. Since the remote free-list is cleared during an allocation when there are no more freed objects in the heap, some allocation operations take longer. Clearing the remote free-list is O(1) if the list can simply be added to the end of the thread-heap's free-list, or O(n) if some maintenance must be performed on each freed object. The time to obtain access for the remote free-list can be limited using lock-free operations (see Section 3.1.6). As long as there is more than one freed object on the list each time the remote free-list is

cleared, performance should be improved.

A remote free-list can also be added to a global heap. The remote free-list on the global heap acts a little differently than on thread heaps, since all frees are remote on the global heap. Thus the remote free-list acts to separate contention for the global heap, since threads allocating from the global heap and threads deallocating to the global heap are not usually contending for the same lock.

3.1.5 Allocation Buffer

An allocation buffer is a chunk of memory that has been allocated from the operating system, but has not yet been allocated to the program. It is basically an area of reserved memory for allocating objects when the free list is empty.

An allocation buffer is used to reduce contention and the number of operating system calls. Rather than reserving memory from the operating system to accommodate a single object, an entire buffer is reserved from which individual objects are allocated later. The buffer may be associated with the global heap, and used when the global heap has no free objects.

An allocation buffer may also be associated with each thread heap, allowing a thread to allocate from the buffer before requesting objects from the global heap, reducing contention for the global heap. To prevent heap blowup, objects should be reused from the global heap before allocating a new allocation buffer. Allocation buffers are useful initially when there are no freed objects in the thread heap and global heap. In the long term, freed objects are used rather than objects from the allocation buffer. Thus, allocation buffers are allocated more frequently to start, but their use eventually diminishes.

Associating an allocation buffer with a thread heap also avoids active false-sharing, since all objects in the allocation buffer are allocated to the same thread. If all objects

sharing a cache line belong to the same allocation buffer, then all objects from a cache line are allocated to the same thread, avoiding active false-sharing. Active false-sharing may still occur when objects from a thread heap are freed to the global heap. Depending on which objects are moved, a future allocation could cause active false-sharing.

Allocation buffers may increase external fragmentation, since some memory in the allocation buffer may never be allocated. A smaller allocation buffer reduces the amount of external fragmentation, but increases the number of calls to the global heap or to the operating system. The allocation buffer also slightly increases internal fragmentation, since a pointer is necessary to locate the next free object in the buffer.

If used with coalescing, the buffer can be a large object that is allocated from the global heap or the operating system and then split into several smaller objects in future allocations.

The unused part of a container, neither allocated or freed, is an allocation buffer. For example, when a container is created, rather than placing all objects within the container on the free list, the objects form an allocation buffer and are allocated from the buffer as allocation requests are made. This lazy method of constructing objects is beneficial in terms of paging and caching. For example, although an entire container, possibly spanning several pages, is allocated from the operating system, only a small part of the container is used in the working set of the allocator, reducing the number of pages and cache lines that are brought into higher levels of cache.

3.1.6 Lock-Free Operations

A lock-free algorithm guarantees that at all times at least one thread is making progress in the system [MHM03]. A wait-free algorithm puts a finite bound on the number of steps any thread takes to complete [Her93]. Lock-free operations can be used in any allocator

as a method to reduce the use of locks. The problem with using a lock is that if the kernel thread associated with the holding user thread becomes blocked, the system as a whole becomes blocked if all other user threads are waiting for that lock [Her93]. However, this situation is unlikely except in an allocator with a lot of contention. Lock-free algorithms may also reduce the number of context switches, since a thread does not yield while waiting for a lock.

The consequence of using lock-free operations is greater complexity and hardware dependency. Lock-free algorithms can be applied most easily to free lists to allow lock-free insertion and removal from the head of a list. Implementing lock-free operations for more complicated data structures may be more complex and depend on hardware support.

3.2 Combining Features

The features discussed in the previous sections can be used in different combinations when designing a multi-threaded memory allocator. An allocator that combines features can solve problems, such as allocator-induced false-sharing, that cannot be solved using any one feature.

Analyzing all possible combinations of allocator features leads to a very large design space. To reduce the analysis, different types of containers and lock-free operations are not specifically discussed. The different types of object containers, varying in size and header information, can be used interchangeably with a basic container and have little influence on the other features. Lock-free operations can be added to any allocator regardless of the other features used.

Per-thread heaps and a global shared-heap, as well as allocation buffers are features used in all allocators discussed. An allocation buffer is implicitly present in a coalescing

allocator, and simply an implementation detail when using object containers. Additionally, both passive and active-false sharing are reduced when combining an allocation buffer with object ownership. Allocators without allocation buffers are possible, and potentially useful, but do not provide additional benefits when considering the combination of features.

The optional features of an allocator that are considered in the discussion are: coalescing, thread-local free-list buffers, and remote free-lists. The coalescing feature is only applied to allocators using individual object headers. Coalescing does not work well with containers, since when objects are split and coalesced the sizes change. As described in Section 3.1.2, containers work best when all objects in the container are the same size.

Thread-local free-list buffers are not considered in combination with remote free-lists. These features are mostly independent of each other. The benefits of using a remote-free list in an allocator are the same whether or not thread-local free-lists are used. Likewise, the benefits of using a thread-local free-list buffer are generally the same whether or not a remote free-list is used, with one exception. The exception is that since a remote free-list removes contention for a local thread-heap, adding a thread-local free-list buffer does not reduce contention for the local thread-heap. The thread-local free-list buffer may still provide other benefits in an allocator with remote free-lists, but they are the same benefits as in an allocator that does not use remote free-lists. Thus, the combination of these features is not discussed since they are mostly independent and no additional insights can be gained.

The design space is broken down based on two main criteria: whether or not headers are container based, and whether or not ownership is enforced. These two criteria have the greatest implications on the performance of an allocator. Using these criteria results in four main types of allocators:

1. Allocators with individual object headers and no ownership

Table 3.1: Feature Combinations

	Coalescing	Base-Case	Thread-Local Free-List Buffer	Remote Free-List
Individual Object Headers No Ownership	No	IN	IN-l	IN-r
	Yes	IN-c	IN-cl	IN-cr
Individual Object Headers Ownership	No	IO	IO-l	IO-r
	Yes	IO-c	IO-cl	IO-cr
Container Headers No Ownership	No	CN	CN-l	CN-r
Container Headers Ownership	No	CO	CO-l	CO-r
All allocators use per-thread heaps, a global shared-heap, and an allocation buffer.				

2. Allocators with individual object headers and enforced ownership

3. Allocators with container headers and no ownership

4. Allocators with container headers and enforced ownership

Using the simplifications and design criteria, Table 3.1 outlines the allocators discussed in this section. Each allocator is given a unique name in the table. As a short form "l" refers to a thread-local free-list buffer, "r" refers to a remote-free list, and "c" refers to coalescing. The first letter indicates whether individual object headers (I) are used or containers (C) and the second letter indicates whether ownership is enforced (N=no ownership, O=ownership).

3.2.1 Individual Object Headers – No Ownership

This section looks at allocators that use individual object headers and do not enforce object ownership.

3.2.1.1 Base Case (IN)

With no ownership, objects are allocated and deallocated to the thread's own thread heap. Thus, thread heaps are only ever touched by one thread and do not require any locking. A lock is only obtained for the global shared-heap when the thread heap has no free objects or too many free objects.

The use of the allocation buffer reduces contention both for the global heap and the operating system, as described in Section 3.1.5. The allocation buffer also reduces active false-sharing, by initially allocating all objects in the buffer to the same thread. However, active false-sharing may still occur when objects are freed to the global heap. As well, passive false-sharing can occur, since objects are freed to the thread-heap that frees them, and may be reallocated to that thread.

3.2.1.2 Thread-Local Free-List Buffer (IN-l)

The thread-local free-list buffer adds no benefit to the IN allocator since there is no contention on the thread heaps.

3.2.1.3 Coalescing (IN-c)

Coalescing is when two free objects next to each other in memory are merged to create a larger free object. There are two options when designing an allocator with coalescing and thread heaps. One option only merges objects on the same heap. A second option allows objects on different heaps to be merged, but requires locking and increases contention on all heaps. Coalescing may avoid highly fragmented memory and may lead to less external fragmentation than an allocator without coalescing, since large objects can be reused for any smaller size request, and smaller objects can be coalesced to satisfy larger requests. However, internal fragmentation is increased since objects must maintain the location of

objects next to them in memory.

Coalescing may reduce active false-sharing. Using the idea of an allocation buffer, when a heap requests memory from the operating system, it requests a large object that is split to the requested size. If this large object is passed to a thread heap, and the thread heap uses this object to split and satisfy allocation requests, then all objects from this large object are allocated to the same thread heap, avoiding active false-sharing. However, when a thread heap frees objects to the global heap, depending on which objects are passed, it may still cause active false-sharing. As in allocator IN, passive false-sharing may still occur when objects are passed among threads in the program.

3.2.1.4 Coalescing and Thread-Local Free-List Buffer (IN-cl)

Using a thread-local free-list buffer in a coalescing allocator can also be used to delay the operation of coalescing objects. Objects placed on the buffer do not change size since they are not coalesced. Hence, if certain object sizes are frequently allocated and deallocated they can be reused from the buffer without going through the processes of being coalesced and split. The buffer acts as a form of cache, caching objects at their requested size until they are no longer useful. External fragmentation may be slightly increased since objects on the thread-local free-list buffer are not coalesced and split. When the buffer is cleared, the objects are coalesced into larger objects that may be more useful in future requests.

If coalescing is used with locks on thread heaps to allow objects on free lists from two separate heaps to be coalesced, then the thread-local free-list buffer also prevents some locking. Operations involving only the local buffer avoid obtaining a lock for the thread heap. Active and passive false-sharing may still occur, as in allocator IN-c.

3.2.1.5 Remote Free-List (IN-r), Coalescing and Remote Free-List (IN-cr)

Adding a remote free-list to thread heaps gains no benefit, since there are no remote-free operations because all objects are freed to the thread heap of the thread that frees them. A remote free-list may be added to the global shared-heap. When thread heaps free objects to the global heap they are placed on the remote free-list. This approach separates contention for the global heap since threads that are passing objects to the global heap are not usually contending with threads that are requesting objects from the global heap. All objects from the remote free-list are moved to the main free-list when a thread requests an object from the global heap and it has no more objects on its main free-list. Adding the remote free-list has no affect on the ways in which false sharing may occur in these allocators.

3.2.2 Individual Object Headers – Object Ownership

Like the previous section, this section assumes individual object headers, but with ownership. Ownership implies that objects must be returned to the heap that allocated them. In order to do so, each object header must store information about the thread that allocates it.

3.2.2.1 Base Case (IO)

Adding object ownership removes all passive false-sharing, since an object is freed to the heap that initially allocated it. Thread heaps must be locked since any thread can access any other thread heap to deallocate an object. Active-false sharing is greatly reduced by using an allocation buffer on each thread heap, but may still occur when objects are freed to the global shared-heap.

3.2.2.2 Thread-Local Free-List Buffer (IO-l)

Adding a thread-local free-list buffer reduces contention for the thread heaps since a thread completes some operations directly through the local buffer. Objects not owned by the current heap may be freed to the local buffer if delayed ownership is used. Delayed ownership allows for potential reuse of the object before the buffer is cleared and the object is returned to its owner, however it also allows passive false-sharing to occur when the object is reused.

3.2.2.3 Coalescing (IO-c)

Using a coalescing allocator with ownership, all allocator-induced false-sharing can be eliminated if large free-objects are allocated such that their boundaries fall on cache-line boundaries. When a free object is split, the ownership of the original free-object is copied to the two new free-objects. Thus, all objects originating from the initial object are owned by the same thread, removing active false-sharing. When freeing objects to the global heap, if only those original large-objects are passed, then all active false-sharing is avoided. The requirement that objects be returned to their owner thread ensures that the original large-objects eventually coalesce to their original state as one object. As in allocator IO, a thread heap must be locked in order to allow any thread to deallocate an object.

3.2.2.4 Coalescing and Thread-Local Free-List Buffer (IO-cl)

The thread-local free-list buffer allows a thread to perform local operations without locking, as in allocator IO-l. Additionally, the buffer can improve performance when objects are reused at their deallocated size by avoiding extra coalescing and splitting, as in allocator IN-cl, at the cost of a slight increase in external fragmentation, since objects are not coalesced while on the buffer.

3.2.2.5 Remote Free-List (IO-r), Coalescing and Remote Free-List (IO-cr)

Adding a remote free-list to the thread heaps reduces contention. Locks can be removed from the thread heaps since they are no longer accessed by other threads. Only the remote free-list needs a lock. A remote free-list may also be added to the global shared-heap to reduce contention, as described in Section 3.2.1.5.

3.2.3 Object Containers – No Ownership

Using object containers without ownership, means objects are allocated and deallocated to the containers in the thread's own thread heap. Using containers can greatly reduce the amount of memory used to store headers, but may also increase external fragmentation depending on the containers used, as described in Section 3.1.2. Cache usage is improved by removing headers from objects, but paging locality may be poor since objects of different sizes must be placed in different containers.

3.2.3.1 Base Case (CN)

As described in Section 3.1.5, active false-sharing is avoided using containers as an allocation buffer. However, once a thread heap reaches its threshold of free objects, it passes some freed objects to the global shared-heap. Depending on which freed objects are transferred, this may induce active false-sharing. Passive false-sharing can also exist since objects can be passed among threads in the program, but may not be returned to the initial thread.

3.2.3.2 Thread-Local Free-List Buffer (CN-l)

Since there are no locks required on thread heaps with no ownership, there is no benefit in using a thread-local free-list buffer.

3.2.3.3 Remote Free-List (CN-r)

Since there are no locks required on thread heaps with no ownership, there is no benefit in using a remote free-list. A remote free-list can be added to the global heap in an attempt to reduce contention, as described in Section 3.2.1.5.

3.2.4 Object Containers – Object Ownership

Using containers with ownership means objects are deallocated to the heap that allocated them. Object ownership information is stored in the container header, applying to all objects in the container. In order to change ownership of an object, the entire container must change ownership. Thus, rather than moving objects between the global heap and thread heaps, entire containers are passed, reducing contention for the global heap.

3.2.4.1 Base Case (CO)

As in allocator CN, using containers avoids active false-sharing by initially allocating all objects in a container to the same thread. When a thread heap reaches its threshold of free objects, it frees a container to the global heap, changing the ownership of the container and all of its objects. This may cause some active false-sharing to occur, as described in Section 3.1.2.1. Passive false-sharing is avoided by freeing objects to the owner of the container.

Additionally, some of the objects in a container transferred to the global heap may still

be in use by the program. Thus, some free operations may free an object to a container that is owned by the global heap, increasing contention for the global heap. However, moving containers between the global shared-heap and a thread heap also reduces contention for the global heap. Rather than making a request to the global heap for every object a thread heap needs, the thread heap makes one request and receives a container with several free objects at once.

Adding restrictions to the movement of containers to require that a container cannot change ownership unless all of its objects are free eliminates all forms of active false-sharing. This restriction also avoids the situations where objects may be freed to a container owned by the global shared-heap, simplifying the global heap and reducing contention for it. This restriction may increase external fragmentation, since free objects in a container cannot change ownership, and hence, are not being allocated.

Maintaining free lists within containers makes the movement of containers a fast operation. A container is taken off the thread heap's list, and moved to the global heap in constant time. If a free-list is not organized by container, then removing all of the container's objects from the thread heap's free list requires O(n) operations.

3.2.4.2 Thread-Local Free-List Buffer (CO-l)

Adding the thread-local free-list buffer can reduce some of the contention for thread heaps. When a thread deallocates an object that belongs to a container its thread heap owns, it can place the object on its private buffer to avoid acquiring a lock. That object can later be allocated by the thread again without obtaining a lock. If the object is owned by another thread, it can be placed on the buffer if delayed ownership is used, potentially causing some passive-false sharing. When the buffer is cleared, locks are obtained to free the objects to the appropriate thread heaps.

3.2.4.3 Remote Free-List (CO-r)

A remote free-list can be added to the global heap to reduce contention, as described in Section 3.2.1.5, and can also be used to remove locks from a thread heap. A remote free-list may be added to each thread-heap, or to each container, moving the lock from the thread heap to the container header. A thread deallocating an object from a container that it does not own, obtains the lock for either the owner thread heap's remote free-list or the container's remote free-list and places the object on the list.

Using remote free-lists on container headers reduces contention for locks, but also increases internal fragmentation since each container header holds a remote free-list. This approach also avoids a situation in which a remote-free operation chases after a moving container. A thread deallocating an object it does not own must determine the thread heap that currently owns the container, but the ownership may change while it is waiting to obtain a lock for the remote free-list. Using remote free-lists on container, even if the ownership of the container changes, the remote free-list used to place the object does not change.

3.3 Summary

This chapter describes several features of a multi-threaded memory-allocator, and the potential interactions of those features. The next two chapters look at existing allocators and a set of test allocators that use these features in different combinations.

Chapter 4

Existing Allocators

The previous chapter discusses the features present in multi-threaded memory-allocators. In order to evaluate how these features perform, both existing allocators and test allocators are examined. There are several existing allocators that tackle the challenges facing a single or multi-threaded memory allocator. This chapter gives an overview of existing allocators that are used to evaluate performance.

4.1 Solaris Malloc

The default allocator on Solaris 8 is used for comparison against other allocators in Chapter 7. Rather than using a sequential-fit allocation algorithm (see Section 2.2.1), it uses a binary search tree to quickly find appropriate free objects. Splitting and coalescing are used along with an allocation buffer from the operating system, providing good control of fragmentation in single-threaded programs. Solaris malloc uses a single lock and a single heap for all memory allocator operations, which slows down multi-threaded programs [Nak01].

4.2 Dlmalloc

This single-threaded, single-heap allocator was created by Doug Lea [Lea]. The allocator uses several techniques to minimizes fragmentation and improve locality. The allocator is thread-safe, meaning that it can be used by a multi-threaded program. However, a single lock is used for the entire allocator, making it very inefficient for use by a multi-threaded program [AN03] [Nak01].

Dlmalloc is a combination of a sequential-fit and binning allocator. Lists of free memory are maintained for each bin size, but objects on the list fall into a range of sizes. The lists are searched for a best-fit, or closest to the requested size, chunk of free memory. Dlmalloc uses coalescing to merge two free objects next to each other in memory into a single larger free object. Large objects are allocated and deallocated directly from the operating system [Lea].

Dlmalloc is not tested in the evaluation presented in Chapter 7 since it does not support multi-threaded programs. Instead, the default Solaris memory allocator is used.

4.3 Ptmalloc

Ptmalloc is included as the default glibc allocator on Linux (with glibc version 2.3.x) [Fer]. The default Linux allocator is used in the evaluation presented in Chapter 7. It is an extension of Dlmalloc with the intention of being used by multi-threaded programs [Glo]. Ptmalloc reduces contention for the memory allocator by having multiple heaps, but it is not exactly per-thread heaps. At each memory operation, a thread first attempts to use the heap it used previously, and if that heap is in use, then it is assigned another heap that is not in use at that time of the request. A new heap is created for an allocation when all other heaps are locked [Fer].

Ptmalloc enforces object ownership, but since there is no one-to-one relationship between threads and heaps, an object is owned by the heap where it was allocated. Each heap is responsible for large chunks of memory, keeping the memory on each heap separate from memory on other heaps. This approach would eliminate active and passive false sharing if each thread always used the same heap, but in Ptmalloc that is not guaranteed.

4.4 Hoard Allocator

Hoard is a multi-threaded allocator built using the heap layers framework [BZM01]. The heap layers framework is meant to help build memory allocators using layers of functionality. The framework provided with version 3.6 is used as the basis for implementing the different test allocators discussed in Chapter 5. The Hoard allocator uses a binning algorithm, which is also used as the basis for the test allocators. Hoard version 3.6 is used in the evaluation presented in Chapter 7 and is available at [Ber]. Several changes have been made to Hoard since the original description provided in [BMBW00]. Version 3.6 is described here.

The Hoard allocator includes several of the features described in Section 3.1. It is a CO-l allocator in Table 3.1 that includes per-thread heaps with a global shared-heap, containers, an allocation buffer, a thread-local free-list buffer, and delayed object ownership. Its object containers, called superblocks, are of a fixed size, with all contained objects being the same size. The superblocks maintain free lists of objects belonging to the superblock [BMBW00].

Delayed object ownership is enforced. All objects are freed to a thread-local free-list buffer, allowing for some passive-false sharing. However, when the buffer is cleared, all objects are freed to the superblock that owns them. Superblock movement is not restricted, allowing superblocks to move to other thread heaps even while they have objects in use

by the program, which does allow for some forms of active false-sharing. However, this is an unlikely occurrence if the heap threshold of free objects is set high enough so that any containers moved are likely to be completely free.

Hoard employs additional optimizations when using certain thread libraries such as pthreads. A function in Hoard is called each time a new thread is created, allowing Hoard to initialize the thread heap, and set a flag indicating the program is multi-threaded. Using this optimization, atomic operations for locking are only used if a program is multi-threaded. This optimization can only be used when support is provided by the thread library.

4.5 Streamflow Allocator

Streamflow [SAN06] is another multi-threaded memory allocator that has been shown to have better or equal performance to Hoard. The version of Streamflow used in the evaluation presented in Chapter 7 is available at [SAN]. Streamflow introduces remote free-lists in order to separate local and remote operations.

Streamflow is a CO-r allocator in Table 3.1 that includes per-thread heaps with a global shared-heap, object ownership, an allocation buffer, remote free-lists, and containers called page blocks. Streamflow uses a different implementation than super-containers to have different container sizes that depend on the size of the objects in the container. Container headers are located in a BIBOP (big bag of pages), which is a table containing one header for every page in the virtual-memory address-space. Thus, all objects in a page share a header and must be the same size. Streamflow also maintains free lists of objects by page blocks.

Streamflow uses remote-free lists to remove heap locks from both malloc and free op-

erations, meaning that most allocation and deallocation operations can complete without acquiring any locks. In addition, Streamflow employs lock-free operations in accessing their remote-free lists.

4.6 Summary

This chapter discusses a group of existing allocators and the multi-threaded features present in each. In order to more fully understand the performance of the multi-threaded features, it is necessary to evaluate additional memory allocators. The next chapter provides a set of test allocators that are used to identify the effects of different multi-threaded feature combinations.

Chapter 5

Test Allocators

The previous chapter discusses features present in existing memory allocators. In order to better comprehend the effects of the different features described in Section 3.1, I implemented a series of test allocators. The first allocator is a basic allocator, and each subsequent allocator adds one or two features at each step to achieve a full-featured allocator.

The allocators are all built using the Hoard heap layers framework ([BZM01]), with some additional heap layers I implemented for specific allocators. The allocators are described, along with some implementation details and the benefits and drawbacks from the previous allocator.

5.1 Allocator A: Base Case

The base-case allocator is a single-heap allocator with one lock around the entire heap. The heap itself is a binning allocator where object requests are rounded up to a bin size. The allocator uses the same set of fixed bin-sizes used by the Hoard allocator. The bin

sizes are closer together for smaller sizes and further apart for larger sizes. A free list is maintained for each bin size. Using a fixed number of bin sizes also implies a fixed number of free lists. The free list for the corresponding bin size is quickly checked for a free object of the correct size. If there are no free objects to reuse, a new object is allocated from an allocation buffer. The 128 KB allocation buffer is allocated using `sbrk`.

A simple header is used with only the object size. The largest bin size is 32 KB less the object-header size. Objects larger than this size are allocated directly using `mmap` so they can be returned to the operating system immediately after deallocation. This approach avoids large amounts of external fragmentation due to infrequent large object requests.

5.2 Allocator B: Add Thread Heaps

The first extension to the base allocator adds per-thread heaps and the base allocator becomes the global shared-heap. This extension makes allocator B equivalent to the IN allocator in Table 3.1. The goal of this allocator is to reduce locking and contention. Operations that can be completed using only the thread heap do not require any locks, leading to reduced contention over the single-heap allocator. A lock is still required when allocating from the global shared-heap.

When an allocation request is made, the free list of the thread heap is checked for a free object of the corresponding bin size. If there are no objects on the list, the global heap is locked while it checks its free list for the corresponding bin size. If the global heap has no free objects of that size, it allocates a new one from its allocation buffer, which could require obtaining a new allocation buffer using `sbrk`.

When an object is deallocated, it is added to the current thread-heap's free list. When a thread-heap's free list for any bin size holds free objects taking up more space than two

times the largest bin size (which is 32 KB less the header size), half the objects are freed to the global heap. Each time a thread heap accumulates more than 64 times the largest bin size on all its free lists, it clears all its free objects to the global heap. Allocations and deallocations of large objects are directly handled through calls to `mmap`, as in allocator A, without using the global heap.

When a new thread starts, a new thread heap is created for it. When a thread runs to completion, its thread heap is placed on a list and reused when a new thread starts up in the future. This reuse implies objects freed by a thread to its thread heap may be reused by new threads created after the initial thread dies. This design does not affect false sharing since the original thread is no longer running. However a thread that inherits a thread heap also inherits the false sharing of the completed thread. Reusing thread heaps may improve performance if the new thread makes similar allocation requests as the original thread, since objects are kept on the thread heap rather than being freed to the global heap.

5.3 Allocator C: Add Object Containers

The next allocator introduces fixed-size object containers with allocation buffers and homogeneous objects. This allocator is equivalent to the CN allocator in Table 3.1. The containers are 64 KB in size with the largest bin size fitting at least two objects in a container (i.e., 32 KB less the size of the header). The container header consists of the object size, the start and end of the container, and an allocation buffer. The allocation buffer points to the next unallocated object in the container's allocation buffer. These object containers reduce internal fragmentation at the cost of external fragmentation, and improve cache usage at the cost of paging locality.

When an allocation request is made, the thread heap checks the free list of the requested bin size for a free object. If there are no free objects, then the thread heap attempts to allocate from the allocation buffer of an appropriate container on its heap. If the allocation buffer is empty, then the global heap checks its free list. If there are no free objects on the global heap for the requested size, then a new container with an allocation buffer is allocated using `sbrk` and returned to the thread heap from which the object is finally allocated. Deallocations occur using the same process as allocator B.

5.4 Allocator D: Add Object Ownership

The ownership of objects is added to remove some allocator-induced false-sharing. This allocator is an implementation of a CO allocator in Table 3.1. In this allocator, objects are deallocated to the thread heap that owns them, not the thread heap of the thread that deallocates them. Object ownership leads to contention for thread heaps, requiring locks to be added, but eliminates passive false-sharing.

A free list is added to container headers so that containers can move easily among thread heaps and the global shared-heap, and reduce contention for the global heap. The slight increase in size of the container header to accommodate the free list slightly increases internal fragmentation.

When an allocation request is made, the thread heap is locked and the free list for the requested bin size is checked. The free list maintains a list of container headers owned by this thread heap that have some free objects, either in their allocation buffer or on their free lists. If there are no free objects on the list, a container is transferred from the global heap, and an object is allocated from that container.

When an object is deallocated, the object is placed at the head of the free list of

its container header, and the container is moved to the front of the container list. This placement allows the next allocation request to receive the last object of that size to be freed. If all objects in the container are free, the container is moved to a separate list of completely-free containers. Completely-free containers are only used for an allocation request when there are no more objects on the main container list. When any free list or all the free lists on a thread heap cumulatively reach a threshold of free objects, as many containers as required are transferred to the global heap, beginning with the containers that are completely free.

The global heap holds lists of containers for each object bin size. When it reaches a threshold of 128 free containers, up to half of the containers are converted into allocation buffers that can be used for a new object size. Only completely-free containers can be converted in this manner. Whenever the global heap has no containers of a requested bin size on its free list, it either reuses one of these containers or obtains a new one using `sbrk`.

Since the global heap can be the owner of some objects, it must handle requests to free objects on its heap, adding another source of contention for the global heap. However, since a thread heap receives a container with several free objects from the global heap, rather than one individual free object, it makes fewer requests to the global heap for free objects.

5.5 Allocator E: Add Restricted Container Movement

The goal of this allocator is to remove all forms of allocator-induced false-sharing by combining containers, object ownership and container movement restrictions. Although allocator E is different from allocator D, it still falls into the category of a CO allocator in Table 3.1. The restriction is that a container can only be moved when the container is completely

free. Because thread heaps are reused rather than destroyed when a thread heap completes, this restriction means the global heap can no longer be the owner of any objects in use by the program, and does not need to handle a free operation for an individual object.

When a thread heap reaches its free-list threshold, only containers on the completely free list are moved to the global heap. Contention for the global heap is further reduced by the fact that each time a thread heap receives a container from the global heap, it has all objects free, further reducing the number of requests a thread heap makes to the global heap. External fragmentation is slightly increased, however, since free objects on a partially free container cannot be used by other threads.

5.6 Allocator F1: Add Thread–Local Free–List Buffer

In order to reduce contention for thread heaps, introduced by allocator D, a thread-local free-list buffer is added. This allocator is equivalent to the CO-l allocator in Table 3.1. Immediate ownership is enforced in order to avoid all allocator-induced false-sharing. Hence, the thread-local free-list buffer only holds objects that are owned by the local thread.

The thread-local free-list buffer contains a set of free lists; one for each bin size. When any free list on the buffer reaches a threshold of two times the container size in bytes, the buffer is cleared to the thread heap. The operation is O(n) since each free object must be placed on its own container header's free list. However, operations that can be completed using only the buffer can be completed quickly with a simple addition or removal from a linked list.

5.7 Allocator F2: Add Remote Free–Lists

This allocator adds remote free-lists that are separated by bin size to thread heaps, making it equivalent to the CO-r allocator in Table 3.1. The situation described in Section 3.2.4.3 where remote-free operations may chase after a moving container is not a concern because container movement is restricted. A container can only change ownership if all objects are free and not on any remote free-list. Placing remote free-lists on thread-heaps rather than container headers, avoids an increase in internal fragmentation. A remote free-list is also added to the global heap to reduce contention as described in Section 3.1.4.

Locks that are added to thread heaps in allocator D are moved instead to the remote free-list. Since there are no locks on thread heaps, a thread-local free-list buffer is unnecessary for avoiding locks on local operations. Therefore, allocator F2 builds on allocator E.

When a thread heap has no objects of the requested bin size on its free list, it clears all objects on the remote free-list. One object from the remote free-list is allocated to the program, while the remaining objects are each placed on the free list of their container headers. The operation is O(n) since each object is freed to its own container header; thus, allocation time is variable.

5.8 Allocator G: Vary Container Size

Building on allocator F2, allocator G adds the variation of the container size in order to reduce external fragmentation. A 64 KB super-container is used to hold containers of 1 KB, 4 KB, 16 KB, or 64 KB in size. All containers within a super-container are the same size (although each container may have a different object size that it holds) to simplify finding the container header. The super-container header simply holds the size of the containers

within it. Since the super-container header is so small, it is easiest to simply add this piece of information to each container header. Thus, the first container in the super-container is used as the super-container header as well. This approach leads to all container headers being aligned within the super-container by the size of the containers.

To find the container header for an object, the lower two bytes are dropped to find the super-container header (aligned on 64 KB addresses). Then the size of the containers within the super-container is used to find the container header. For example, if the containers in the super-container are 4 KB in size, then the lower 12 bits of the object address are dropped to find its container header.

The container size is determined based on two factors. First, the smallest container that fits at least two objects of the requested size is used. Second, the number of requests for a container of each bin size is recorded, and if this counter has reached a certain number, then the container size is increased. The slight increase in header size and the increase in number of containers increases internal fragmentation.

5.9 Allocator H: Add Lock-Free Operations

Lock-free operations, discussed in Section 3.1.6, may improve performance. Adding lock-free operations to the F2 allocator removes locks from remote free-list insertions and clears on both thread heaps and the global heap, removing all locks from the allocator. The drawback to adding lock-free operations is the increase in code complexity and hardware dependency.

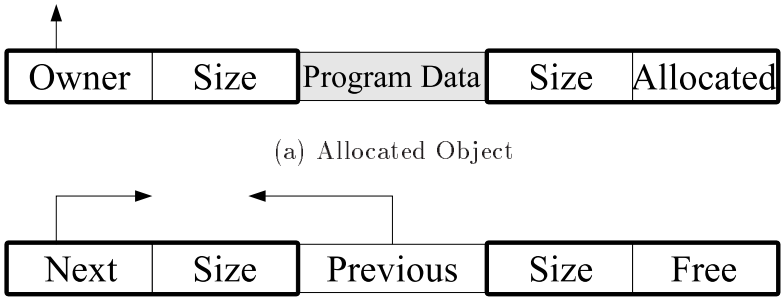
5.10 Coalescing Allocator

In addition to the multiple non-coalescing test-allocators, I also implemented a single coalescing allocator, IO-clr (a combination of IO-cl and IO-cr in Table 3.1). This allocator avoids all forms of false-sharing, and reduces contention through the use of per-thread heaps and a remote-free list. The thread-local free-list buffer is also used in order to cache freed objects and reduce some unnecessary coalescing and splitting. The same bin sizes are used as the other test allocators, with the largest bin size being 64 KB less the header size. However, a variation of binning is used that allocates objects to the exact requested size. Each free list contains objects that range in size from next smallest bin size to the current bin size.

This allocator uses both headers and trailers each containing the object size, as shown in Figure 5.1. The object size allows the header to locate the trailer and the trailer to locate the header, which is necessary when coalescing with objects before and after the object in memory. The trailer also holds an additional flag to indicate whether the object is allocated or free. In addition to the object size, when the object is allocated, the header points to the owner of the object, as shown in Figure 5.1(a). When the object is free, as in Figure 5.1(b), the header points to the next object on a free list, and the deallocated storage points to the previous object on a free list.

When an allocation request is made, the thread-local free-list buffer is checked first. The request size is rounded down to the nearest bin size, and if there are no objects on that list large enough for the requested size, then the next largest bin's free list is checked. If there are no objects on that free list, then the thread heap checks its free-lists.

The thread heap maintains doubly linked lists of free objects. The request size is rounded down to the nearest bin size and if any object on that free list is large enough, it is removed from the free list and allocated to the program. If there are no free objects of



(a) Allocated Object

(b) Free Object

Figure 5.1: Header and Trailer Structure

that bin size, then the first object found on any larger bin size is split. One piece becomes an object of the requested size, while the remaining piece is placed on the free list for the appropriate bin size. If there are no free objects large enough in the thread heap, the remote free-list is cleared. All objects on the remote free-list are freed to the thread heap, and if any are large enough for the requested object size, it is allocated to the program. If no free object is large enough, then the thread heap requests an object from the global heap. The global heap maintains one list of free objects, all 64 KB in size. If the global heap has no free objects, it allocates one using `sbrk` with 64 KB alignment.

When an object is deallocated, if it is owned by the current thread, it is placed on the thread-local free-list buffer. Otherwise, it is placed on the remote-free list of the owner thread. When the thread-local free-list buffer reaches a threshold of 64 KB of free objects, all objects are freed to the thread heap. When an object is freed to the thread heap, it checks the objects immediately before and after it, and coalesces them if they are on a thread-heap free-list, are owned by the same thread, and belong to the same 64 KB alignment. The free objects being coalesced are removed from their current free list, coalesced into one free object, and placed on a free list with a bin size based on the coalesced size. When objects are coalesced into a 64 KB object, the object is freed to the

global heap.

5.11 Summary

This chapter describes a set of test allocators that each implement a subset of the multi-threaded features described in Chapter 3. The next chapter considers several single and multi-threaded benchmarks that are used to evaluate the existing and test allocators.

Chapter 6

Memory Allocator Test Suite

The previous two chapters describe a set of existing allocators and test allocators used in evaluating multi-threaded memory-allocator features. This chapter describes a test suite of single and multi-threaded benchmarks used to compare the memory allocators.

6.1 Single–Threaded Benchmarks

There are several single-threaded benchmarks used for comparing memory allocators [BMBW00] [BZM02] [BZM01] [DDZ93]. This section looks at several of these benchmarks and their allocation characteristics. Finally, a smaller set of benchmarks is selected for the evaluation of memory allocators.

In order to characterize the benchmarks, I modified the Hoard allocator to include an additional layer for collecting statistics. These statistics are independent of the allocator used. The number of memory allocations and deallocations is recorded and summarized in Table 6.1. For example, the first row is for benchmark P2C indicating a total of 199,263 allocation requests, a total of 188,058 deallocation requests, a total of 387,321 memory

requests (a sum of both allocations and deallocations), and 5.62% of allocated objects are not deallocated during the program's lifetime.

I made a second modification to Hoard to collect timing information, by adding a layer to mark the time of the first memory operation to the last memory operation, as well as the total time spent in all memory operations. Although these statistics are dependent on the memory allocator, they provide a general idea about the behaviour of the program. Table 6.2 shows the statistics collected from running the benchmark programs with this allocator. For example, the first row is for benchmark P2C indicating it runs for 780 ms from its first allocation to its last allocation or deallocation (the last time the allocator has control), it makes 255,322 allocations and 496,287 calls to the memory allocator (in both allocations and deallocations) per second (based on its 780,436 us runtime and the number of operations listed in Table 6.1), it spends a total of 109 ms in the memory allocator, which is 14% of its total runtime (as measured from the first to last memory operation).

This information gives a general idea of how memory intensive these benchmarks are. Espresso, CFRAC, GCC, Perl, Gawk and ROBOOP make a large number of memory requests, while Espresso-2, GMake, Perl-2, Gawk-2, and XPDF-2 make a small number of memory requests. All programs except GMake and Gawk-2 deallocate nearly all of their objects. Espresso, CFRAC, GCC, Perl, Gawk, ROBOOP, and Lindsay are long running programs, while Espresso-2, GMake, Perl-2, and Gawk-2 are short running programs. GS, Espresso, Espresso-2, CFRAC, CFRAC-2, Perl, Gawk, and ROBOOP make a large number of allocation/deallocation requests per second and spend a significant portion of their runtime in memory operations, while GCC, Perl-2, Gawk-2, and Lindsay make a small number of allocation/deallocation requests per second and spend an insignificant portion of their runtime in memory operations. The following sections give a brief description of the benchmark programs.

Table 6.1: Allocation Statistics

Benchmark	# Allocs	# Deallocs	Memory Ops	% Unfreed
P2C	199,263	188,058	387,321	5.62
GS	108,546	102,388	210,934	5.67
Espresso	1,675,492	1,675,490	3,350,982	0.00
Espresso-2	24,759	24,757	49,516	0.01
CFRAC	10,890,124	10,890,122	21,780,246	0.00
CFRAC-2	227,092	227,090	454,182	0.00
GMake	4,641	2,662	7,303	42.64
GCC	651,919	645,359	1,297,278	1.01
Perl	591,984	590,778	1,182,762	0.20
Perl-2	16,343	15,865	32,208	2.92
Gawk	874,306	873,809	1,748,115	0.06
Gawk-2	3,760	2,953	6,713	21.46
XPDF	227,073	224,471	451,544	1.15
XPDF-2	61,501	58,975	120,476	4.11
ROBOOP	9,268,177	9,268,175	18,536,352	0.00
Lindsay	108,790	108,788	217,578	0.00

Table 6.2: Runtime Statistics

Benchmark	Total Runtime (us)	Avg Allocs/sec	Avg Memory Ops/sec	Time in Memory Ops (us)	Time in Memory Ops (%)
P2C	780,436	255,323	496,288	109,709	14.1
GS	358,263	302,978	588,768	70,209	19.6
Espresso	3,700,698	452,750	905,500	1,020,538	27.6
Espresso-2	49,898	496,196	992,352	14,432	28.9
CFRAC	20,567,315	529,487	1,058,974	5,443,342	26.5
CFRAC-2	311,489	729,053	1,458,099	118,520	38.1
GMake	16,383	283,282	445,768	2,177	13.3
GCC	15,546,826	41,933	83,443	608,727	3.9
Perl	1,882,696	314,434	628,228	329,622	17.5
Perl-2	102,332	159,706	314,741	10,102	9.9
Gawk	2,602,546	335,943	671,694	529,742	20.4
Gawk-2	42,619	88,224	157,513	2,020	4.7
ROBOOP	14,563,440	636,400	1,272,800	5,295,497	36.4
Lindsay	4,682,409	23,234	46,467	69,283	1.5

6.1.1 P2C

P2C is a Pascal to C translator. The version provided in Hoard heap layers version 3-4-0 is compiled and run with an input provided by the download (mf.p).

6.1.2 GS

GS is a postscript interpreter and viewer. GS version 2.1 is run with a 422 KB input file, provided by the Zorn download, and the display turned off, so that it only interprets the postscript file [DDZ93].

6.1.3 Espresso/Espresso-2

Espresso is an optimizer for programmable logic arrays. Espresso version 2.3 (released 01/31/88) is run with the two inputs that are included in the Hoard heap layers download: largest.espresso as the first input and Z5XP1.espresso as the second input. While the runtime and number of objects increase as the size of the input file increases, the average calls per second and percent of time spent in memory operations does not change.

6.1.4 CFRAC/CFRAC-2

CFRAC is an implementation of the continued fraction algorithm for factoring large numbers. The version provided by the Hoard heap layers download is compiled and run with two different input numbers: 35 digits longs (41 757 646 344 123 832 613 190 542 166 099 121) and 22 digits long (1 000 000 001 930 000 000 057, which is a product of two primes). The larger the number, the longer the program runs, but with both inputs, a significant portion of the program is spent in memory operations.

6.1.5 GMake

GMake version 3.80 is run with the input provided in the Zorn download [DDZ93]. A large portion of objects are never deallocated, indicating that most memory operations are allocation requests.

6.1.6 GCC

GCC version 3.4.2 is run with options to run only the compile step and with second level optimizations to compile combine.c - the largest file in the GCC source code.

6.1.7 Perl/Perl-2

Perl is a scripting language. It handles all memory management for the script that it runs. Perl version 5.005_03 for sun4-solaris is run with the two inputs provided in the Zorn download [DDZ93]. The first script is called adj and formats text based on some inputs for line length and indentation. The second script, called hosts, transforms a host file from one format to another. The runtime and number of memory requests varies greatly depending on the script being run.

6.1.8 Gawk/Gawk-2

Gawk is a scripting language like Perl. Gawk version 2.11 is run with the two inputs provided in the Zorn download [DDZ93]. The first script is a Gawk version of the adj script used for Perl. The second script, called prog, calculates allocation costs for a memory simulation. Like Perl, the memory call behaviour varies depending on the script being run.

6.1.9 XPDF/XPDF-2

XPDF version 3.01 is run twice to open a file that has 13 pages and is about 730 KB. XPDF is a graphical program, and thus required interaction to run. In the first run, the first 4 pages are flipped through and then the viewer is closed. In the second run, the viewer is closed as soon as the window opens. The runtime and time spent in memory operations are irrelevant for this benchmark since it is highly dependent on input from the user, and hence do not appear in Table 6.2.

6.1.10 ROBOOP

ROBOOP is a robotics simulation toolkit. Version 1.09 is run with the bench executable provided with the ROBOOP toolkit to benchmark different operations. This benchmark runs slightly longer than most of the single-threaded benchmarks, makes a large number of memory requests, and spends a significant portion of its runtime in memory operations.

6.1.11 Lindsay

Lindsay is a hypercube simulation. The version provided with the Hoard download is compiled and run with the input provided. This benchmark is one of the least memory intensive benchmarks making a relatively small number of memory requests, and spending a small portion of its time in memory operations.

6.2 Multi–Threaded Benchmarks

Several micro-benchmarks have been created for comparing multi-threaded memory allocators [BMBW00] [SAN06]. This section describes them in detail. Each benchmark has a

specific memory allocation pattern and number of memory operations.

6.2.1 Recycle

The Recycle benchmark stresses the ability of the allocator to handle different threads allocating and deallocating independently. There is no interaction among threads. The number of threads is an input parameter. Each thread allocates 1 000, 8-byte objects then deallocates them in the order they were allocated. The total number of objects allocated in the program is 10^7 and is distributed among its threads. Hence, the work performed by each thread decreases as the number of threads increases.

6.2.2 Consume

Consume is a micro-benchmark that simulates a producer-consumer scenario. Its purpose is to test for heap blowup in a situation where only one thread allocates objects, and other threads only deallocate objects. The number of consumer threads is an input parameter. One producer creates 6 000, 8-byte objects for each consumer thread. Once a set of 6 000 objects is created, it is given to a consumer to deallocate. This process is repeated 5 000 times. No work is done on the objects, so as consumer threads are added the producer becomes the bottleneck.

6.2.3 False–Sharing Micro–benchmarks

Two micro-benchmarks, Passive-False and Active-False, are used to test for passive and active false-sharing. Both benchmarks are provided with the Hoard download. The number of worker threads is an input parameter in each benchmark. In Passive-False, the main thread creates a number of worker threads, passing to each an 8 byte object it allocated.

Each worker thread deallocates the object, allocates a new 8 byte object, writes to it 10 000 times, and then repeats the process 100 000 times divided by the number of threads. Active-False is the same as Passive-False, except that no initial object is created by the main thread. The amount of work is constant and distributed over the number of threads. Therefore, ideally the runtime should scale with the number of threads.

6.2.4 Larson

Larson is a micro-benchmark provided with the Hoard download that simulates the memory allocation behaviour of a server [LK99]. The benchmark is run for 30 seconds creating objects of random sizes between 10 and 100 bytes. The number of active threads remains constant through the life of the program, but is configured as an input parameter. Each thread is passed an array of 10 000 objects. It then randomly selects an object to destroy and replaces it with a new one, thus maintaining a working set of 10 000 objects. Each thread repeats the deallocation/allocation process 100 000 times. Finally, before the thread terminates it passes its array of 10 000 objects to a new child thread to continue the process. The number of generations varies depending on the speed of the threads. The throughput is calculated as the number of allocations that occur per second.

6.3 Trace Collection

The multi-threaded micro-benchmarks are simple programs, and their allocation behaviour is well understood. To further analyze the allocation patterns of the single-threaded benchmarks, traces are collected. While statistics give general overall characteristics of the program, traces can be used to collect information about the patterns of allocations and deallocations throughout the lifetime of the program.

To collect the traces, I added a log heap-layer to the Hoard memory allocator written using heap layers (version 3.4.0). For each malloc and free, a log record is generated including the size of the malloc and the address of the malloc or free. At the end of a program, the records are all written out to a file, which reduces the probe effect on the program. Each entry in the log also contains a time stamp when the operation occurred.

To add a time stamp to the records, several methods for collecting time were considered, including: getitimer, gettimeofday, gethrvtime/gethrtime, and cpu performance counters (all performance counters are hardware/software dependent). Each method is considered for use on a Sparc machine running Solaris. Two requirements were used to select a timing method: the timer resolution and its ability to measure virtual time. In these programs it is necessary to have microsecond resolution. A virtual timer that does not count time during a kernel-thread time slice (when the program is not running) is also necessary to obtain an accurate measurement. Of the listed options, the cpu performance counters provide a virtual timer at the required resolution.

6.4 Trace Results

Analysis of the collected traces provide an overall distribution and variation over the lifetime of the program for each of the following pieces of information:

- sizes of requests
- lifetime of objects
- interarrival times of allocations and deallocations
- allocation footprint

Table 6.3: Size of Requests

Benchmark	Average object size (bytes)	Total allocated (bytes)	Largest object (bytes)	Smallest bin size(bytes)	Most common bin size (bytes) - frequency (%)
P2C	24.3	4,851,116	8,200	8	24 - 63.0
GS	173.3	18,806,773	20,016	16	296 - 38.4
Espresso	64.0	107,184,579	55,072	8	32 - 52.1
Espresso-2	43.1	1,068,053	8,200	8	8 - 41.1
CFRAC	17.7	192,941,761	8,200	16	16 - 60.5
CFRAC-2	14.7	3,338,366	8,200	8	16 - 71.1
GMake	48.5	224,949	8,200	8	8 - 39.4
GCC	880.5	574,031,347	932,052	8	40 - 30.0
Perl	19.8	11,740,395	8,203	8	8 - 64.2
Perl-2	25.7	420,114	8,257	8	24 - 38.1
Gawk	55.9	48,836,034	8,200	8	8 - 28.0
Gawk-2	28.4	106,848	8,200	8	8 - 81.5
XPDF	229.0	51,991,393	2,955,168	8	8 - 56.7
XPDF-2	327.1	20,118,022	2,955,168	8	24 - 35.0
ROBOOP	34.7	321,322,872	8,200	8	24 - 54.6
Lindsay	67.8	7,373,660	1,490,944	8	56 - 93.0

6.4.1 Sizes of Requests

Some general statistics on allocation sizes are shown in Table 6.3. The bin sizes used are those used in the Hoard allocator. Half of the benchmarks have a majority of objects falling under one bin size. For all programs, the most common bin size is quite small.

Figures A.1 and A.2 show the distribution of objects among bin sizes having at least one percent of objects. Most programs allocate objects in only a few of the smaller bin sizes. Almost all of the benchmark programs have 75% of their objects falling in one to three bin sizes. All bin sizes that account for at least one percent of the objects in the program fall in a small range from the smallest bin size to a bin size less than half a kilobyte. The single exception is GCC, which has several larger objects.

The allocation sizes over time are shown in Figures A.3 and A.4. To reduce the number of data points on the graph, several nearby points are condensed into one point, with the different colours indicating the number of objects condensed into one point.

The allocation behaviour falls into two categories. The first category consists of programs that are very uniform in the allocation behaviour for the entire program. The second category consists of programs that have distinct segments with each segment having different allocation patterns. These segments of different behaviour reappear in all graphs showing allocation characteristics over the runtime of the program. GS, CFRAC, Perl (both inputs), Gawk (both inputs), and Lindsay fall into the first category. P2C, Espresso (both inputs, although it is more clear for input 1), GMake, GCC, XPDF, and ROBOOP fall into the second category.

In P2C, the allocation behaviour changes for two short periods during the program, while the rest of the time the allocation sizes are quite uniform. Espresso has several different program segments that behave differently in terms of memory allocation sizes. GMake and GCC are difficult to categorize because GMake has too few points, and GCC has too many. However, GMake allocates certain object sizes in the first half of the program that are different from those in the second half. GCC has a short startup period in which the allocation behaviour differs from the rest of the program. XPDF has different allocation patterns for startup, page loading, and shutdown. In the first input, the pages of the document are flipped, causing a repeat of that segment of the program. ROBOOP has three different segments of different allocation behaviour.

6.4.2 Lifetimes of Objects

The lifetime of an object is calculated as the difference between the timestamp taken just after the malloc operation and the timestamp taken just before the free operation. Some

Table 6.4: Lifetimes

Benchmark	Avg Lifetime (freed) (us)	Avg Lifetime (all) (us)	Shortest Lifetime(us)	Longest Lifetime (freed) (us)
P2C	4,206	30,858	0.2	771,008
GS	67	12,657	0.5	352,524
Espresso	625	630	0.2	3,699,626
Espresso-2	178	183	0.2	48,926
CFRAC	8,339	8,341	0.4	20,566,470
CFRAC-2	772	774	0.4	310,698
GMake	263	4,288	0.9	6,873
GCC	2,622	105,298	0.2	15,524,398
Perl	2,905	6,700	0.5	1,881,408
Perl-2	1,986	4,852	0.6	101,201
Gawk	47	1,511	0.7	2,599,207
Gawk-2	453	8,835	1.1	38,128
XPDF	66,535	104,850	0.5	3,520,422
XPDF-2	53,597	82,725	0.5	826,259
ROBOOP	102	105	0.6	14,562,460
Lindsay	8,804	8,890	3.0	4,682,065

statistics on average, shortest, and longest lifetime are shown in Table 6.4. An object that is never deallocated has a lifetime from the time it is allocated to the time of the last memory operation of the program (which is the last measurable point for the logging layer). The average lifetime of objects is given both for objects that are deallocated (freed) and for all objects (including ones not freed). The shortest lifetime is less than one microsecond for most programs. The longest lifetime of an object that is deallocated before the completion of the program is also shown. In many cases, this is an object that is allocated near the start of the program, and deallocated just before the end of the program.

Figures A.5 and A.6 show a cumulative distribution of lifetimes of objects. Most objects live for a very short time. The cumulative distribution of lifetimes indicates that for almost all programs, at least half of the objects live for less than 100 us. Plotting object

size relative to the object's lifetime indicates that most objects are small and have a short lifetime. Lindsay is the only program to exhibit a very different behaviour in terms of lifetime distribution. Almost all objects in Lindsay live close to 10 000 us. No other program exhibits such a large and steep spike in lifetime distribution and at a relatively large lifetime.

For the most part, the lifetime distribution graphs tend to follow one of two patterns. The first pattern is an S-curve in the cumulative distribution. This pattern indicates a large portion of objects having similar lifetimes. The programs that follow this pattern are P2C, GS, Espresso (both inputs), GMake, GCC, XPDF (both inputs) and ROBOOP. GS, GMake and XPDF are slightly different. Their graphs indicate that a number of objects also have longer lifetimes. This behaviour is likely the result of a large number of objects allocated near the start of the program and freed near the end. The second pattern is a set of steps in the cumulative distribution graphs. This behaviour indicates several popular object lifetimes. The programs that follow this pattern include CFRAC (both inputs), Perl (both inputs), Gawk (both inputs) and Lindsay (although it really only has two steps). Gawk and Perl both have very similar functions, that leads to similar behaviour in lifetime distributions.

Figures A.7 and A.8 plot the lifetime of objects relative to the time in the program they are allocated. These graphs show the changes in program behaviour that also appear in Figures A.3 and A.4 and are noted in Section 6.4.1. Since most objects have very short lifetimes, it is difficult to see detailed patterns in Figures A.7 and A.8. Figure A.9 shows some of the details in Figures A.7 and A.8 by showing only short-lived objects over a short period of time for some of the benchmark programs. These figures show that there are at least two common patterns of repeated behaviour in terms of the lifetimes of objects.

One common pattern, shown in Figures A.9(a) to A.9(e), are dots forming straight

Table 6.5: Interarrival Times

Benchmark	Average Time (us)			Shortest Time (us)			Longest Time (us)		
	All	Alloc	Dealloc	All	Alloc	Dealloc	All	Alloc	Dealloc
P2C	2.0	3.9	4.1	0.2	0.4	0.4	3,078	3,420	4,178
GS	1.7	3.3	3.5	0.3	0.4	0.5	586	586	622
Espresso	1.1	2.2	2.2	0.2	0.3	0.4	4,019	4,022	4,486
Espresso-2	1.0	2.0	2.0	0.2	0.3	0.4	806	839	844
CFRAC	0.9	1.9	1.9	0.2	0.3	0.4	408	5,237	413
CFRAC-2	0.7	1.4	1.4	0.2	0.4	0.4	43	724	51
GMake	2.1	3.3	5.7	0.3	0.3	0.4	137	137	273
GCC	12.0	23.8	24.1	0.2	0.3	0.3	106873	106924	107578
Perl	1.6	3.2	3.2	0.2	0.4	0.4	93	149	232
Perl-2	3.2	6.2	6.4	0.2	0.4	0.5	79	91	261
Gawk	1.5	3.0	3.0	0.2	0.4	0.4	65	82	384
Gawk-2	6.2	11.1	14.0	0.3	0.5	0.5	98	112	1,166
XPDF	7.8	15.5	15.7	0.2	0.4	0.4	141926	141931	141945
XPDF-2	6.9	13.4	14.0	0.3	0.3	0.4	139194	139199	139215
ROBOOP	0.8	1.6	1.6	0.3	0.4	0.4	471	471	113
Lindsay	21.5	43.0	42.9	1.3	1.3	1.6	7466	7466	2697

lines. When the dots in the line are close together, this indicates objects that are allocated close together and are also freed close together. When these lines fall at 45 degrees, this indicates that objects are being allocated over time and then deallocated all at once. When the line is vertical, the objects are allocated at the same time and then deallocated slowly over time in reverse order. When the line is horizontal, the objects are allocated together and deallocated together in the same order.

A second common pattern is a group of repeated lifetimes, which is likely caused by repeated behaviour in the program such as what might occur in a loop. This pattern is demonstrated in Figures A.9(d) to A.9(h).

6.4.3 Interarrival Times of Allocations and Deallocations

The interarrival time is calculated as the amount of time since the previous request. The interarrival times of all memory requests, allocations, and deallocations are calculated from the trace logs. The average, shortest, and longest times are shown in Table 6.5. Figures A.10 and A.11 show a cumulative distribution of the three types of interarrival times. The interarrival times are all very short, with about 90% of calls being less than 10 us apart in all benchmarks except GCC and Lindsay. There appears to be a relationship between the interarrival times and object lifetimes, as the cumulative distribution graphs of both are very similar in shape for these benchmarks programs.

6.4.4 Allocation Footprint

The allocation footprint is calculated by increasing the memory size by the requested size, each time an object is allocated, and decreasing the memory size by the object size, each time an object is deallocated. The memory size over the runtime of each program is shown in Figures A.12 to A.14. The number of allocated objects is also shown as a separate line. This line indicates whether large increases in the footprint are caused by a large number of objects being allocated close together, or one large object being allocated. Table 6.6 shows the average and maximum values of these two lines.

There are two types of patterns in the allocation footprint. One pattern is a gradual increase, the second is a fast increase followed by a plateau and a final drop in the allocation footprint and number of objects. Programs that do not deallocate all objects do not return to a zero allocation-area size at the end of the program. The segments of different program behaviour noted in Section 6.4.1, also appear in the allocation footprint graphs.

Table 6.6: Allocation Footprint

	Size (bytes)		Number of Objects	
Benchmark	Average	Maximum	Average	Maximum
P2C	275,871	406,897	7,874	12,645
GS	351,001	484,049	3,823	6,197
Espresso	169,710	280,115	285	4,389
Espresso-2	23,688	42,873	92	691
CFRAC	75,784	150,036	4,417	8,810
CFRAC-2	8,890	18,395	565	1,231
GMake	44,511	62,366	1,165	1,987
GCC	2,609,506	4,830,775	4,415	8,019
Perl	114,888	123,694	2,107	2,137
Perl-2	72,167	82,015	778	804
Gawk	38,232	38,905	508	551
Gawk-2	66,555	70,956	779	828
XPDF	5,845,536	6,701,879	6,762	9,609
XPDF-2	4,839,299	6,552,307	6,152	8,324
ROBOOP	14,083	15,960	67	117
Lindsay	1,910,420	1,915,712	207	296

6.5 Benchmark Selection

Since single-threaded benchmarks do not highlight the efficiency of a multi-threaded memory-allocator, only a small set of single-threaded benchmarks are necessary. The programs selected are P2C, espresso (input 1), GCC, gawk (input 1), and ROBOOP. The analysis of the selected benchmarks indicates that they have a wide range of allocation characteristics, which justifies their use in characterizing the performance of a memory allocator. The discarded benchmarks have very similar allocation behaviours to the selected benchmarks or have some other disadvantage. For example, XPDF uses a graphical interface and depends on user interaction, and some benchmarks have a relatively small number of memory allocations.

6.6 Summary

This chapter describes the test suite used to evaluate memory allocator performance. The next chapter uses this test suite to compare existing allocators and test allocators.

Chapter 7

Memory Allocator Evaluation

The previous chapter describes a set of single and multi-threaded benchmark programs used to evaluate multi-threaded memory allocators. This chapter analyzes the results of running the benchmarks with existing allocators and test allocators.

The main goal of a multi-threaded allocator is to allow a well behaved multi-threaded program to scale in performance as threads are added. If a program is written to scale with the number of threads, the memory allocator should not be the bottleneck preventing it from scaling. Scaling can be tested using micro-benchmarks that are written to scale and stress the memory allocator in different situations. A related measurement is the overall runtime of the programs. Running single and multi-threaded benchmark programs using different memory allocators shows the overall effect of the allocator in each context.

Besides the runtime performance of a memory allocator, the amount of memory that it requires may also be important. Typically, there is a trade-off between runtime performance and memory usage. Different situations place different priorities on these goals.

Internal and external fragmentation indicates the amount of additional memory required by a memory allocator. However, these measures are difficult to obtain without

Table 7.1: Test Setup

Setup	OS	Number of CPUs	CPU detail	Memory
A	Solaris 8	8	900 MHz Sparc	16 GB
B	Linux	8	2.5 GHz Dual Core AMD Opteron	16 GB
C	Linux	64	1.3 GHz Itanium 2 IA-64	192 GB

modifying the source code of the memory allocator. An indirect way to observe the effect of the memory allocator on memory usage is to observe the virtual memory and resident-set size used by the program while it is running. These measures indicate the amount of memory that is reserved from the operating system and the overall effect of the allocator on the system in which the program is running.

Table 7.1 describes the three test setups in which the benchmark programs are run. Setup A and B are used by other users, and hence the benchmarks cannot make full use of all CPUs. This interference causes a flattening of performance curves at their ends. The setup C machine had 8 CPUs isolated for the purpose of these tests. Due to the different architectures, Streamflow only supports setup C. On Setup A, the default allocator is Sun's malloc (a single-heap allocator), whereas on Setup B and C, the default allocator is Glibc (Ptmalloc).

This section looks at these measurements for the existing allocators described in Chapter 4 as well as the test allocators described in Chapter 5 using the benchmark programs described in Chapter 6.

7.1 Runtime and Scaling

The single-threaded benchmarks are tested for a comparison between single-threaded and multi-threaded allocators. The multi-threaded benchmarks show how different memory allocators influence the performance as threads are added to the program. Each micro-

benchmark is designed to stress a different issue in multi-threaded programs. This section looks at each benchmark and how the different allocators perform.

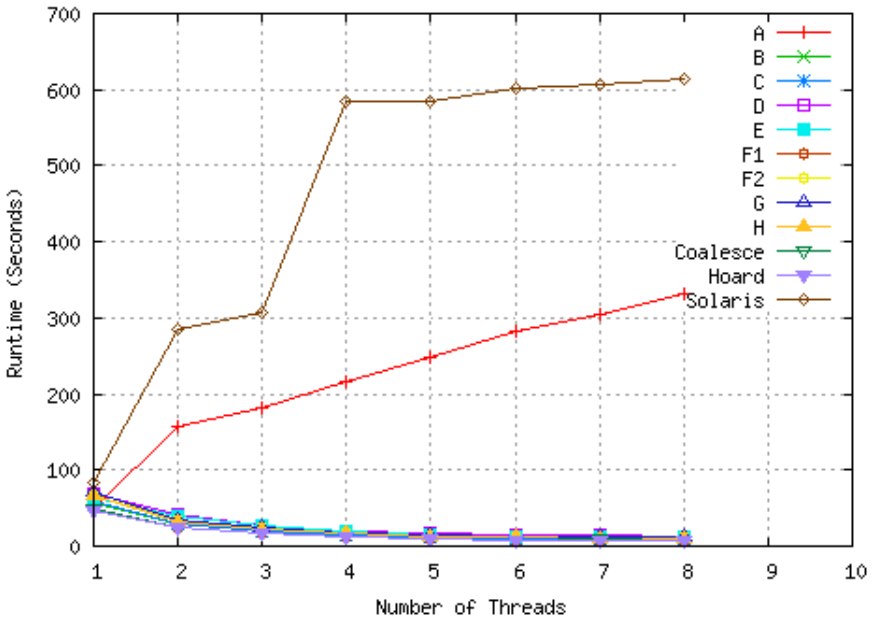
7.1.1 Single-Threaded Benchmarks

In order to replace the default allocator for all programs, the new allocator should perform at least as well as the current default allocator in both single and multi-threaded programs. All the single-threaded benchmarks have very short runtimes of just a few seconds. Since the runtime measurement only measures to a precision of milliseconds, only large differences can be identified. However, the runtime varies very little among allocators. They all perform equally well to the default Solaris or Linux allocator.

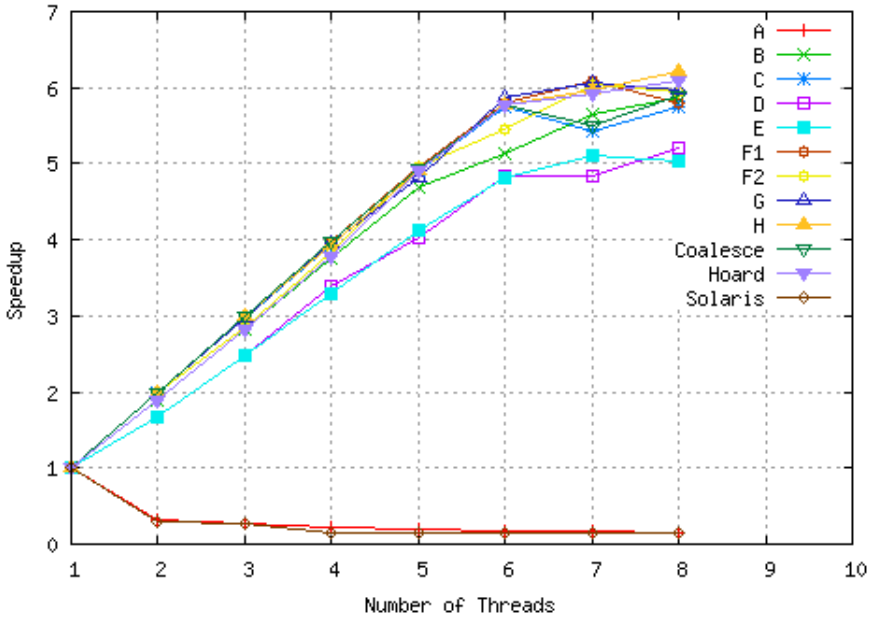
7.1.2 Recycle

The Recycle benchmark stresses the ability of the allocator to handle different threads working independently. In this situation, per-thread heaps and reduced locking in local operations result in the best performance, since all operations are local to each thread. The benchmark is run on all three test setups with the results in runtime and speedup shown in Figures 7.1 to 7.3. The speedup is calculated as the runtime with one thread divided by the runtime with `n` threads.

In setup A, Figure 7.1(b) shows that a single-heap allocator limits the scaling of Recycle. Both the Solaris default allocator and allocator A degrade the performance of the program as threads are added. The increased contention for the single heap prevents any parallel execution in the program. Allocator B removes a great deal of the contention observed in the Solaris and A allocators by adding per-thread heaps. However, allocator B does not prevent active false-sharing, which leads to slightly less than perfect scaling. Allocator C adds containers, which removes most active false-sharing, leading to perfect scaling up to



(a) Runtime

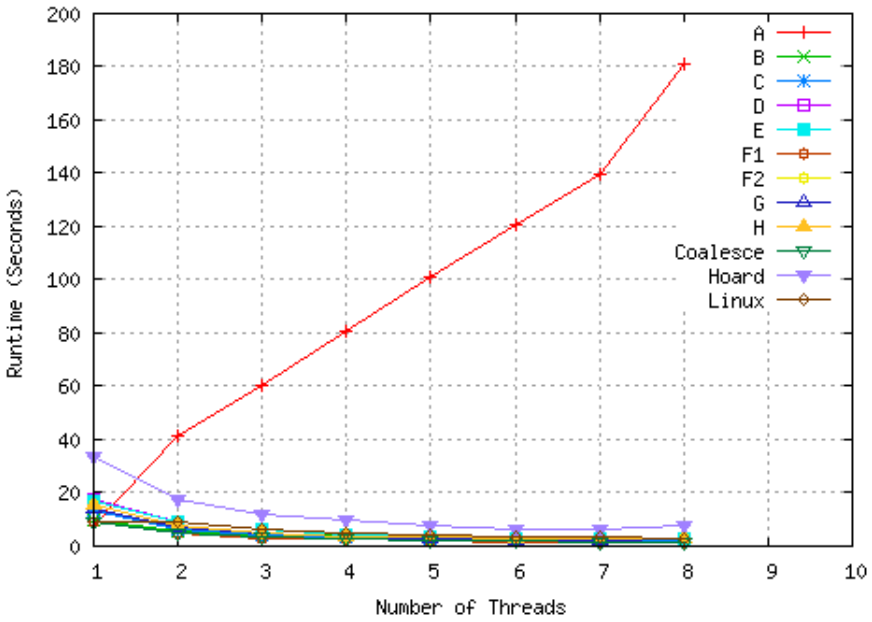


(b) Speedup

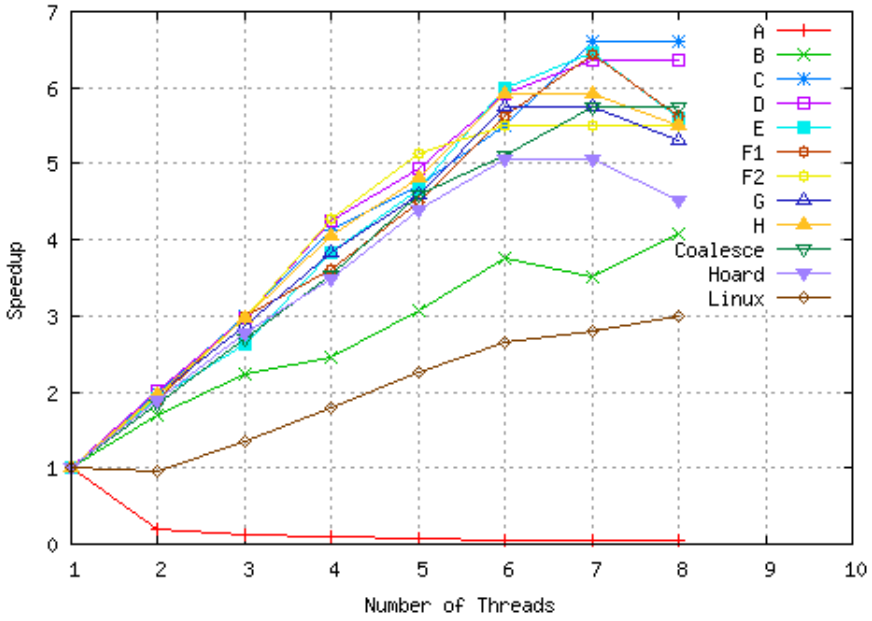
Figure 7.1: Scaling in Recycle on Setup A

six processors. Allocators D and E introduce locks, which increases their runtime by a constant amount, except in the case of one thread. An optimization in heap locking avoids atomic operations if the program is single-threaded. For this reason, the single-threaded case performs slightly better and lowers the calculated scalability for the multi-threaded cases in allocators D and E. Allocators F1 and F2 use thread-local free-lists and remote free-lists to remove the need to obtain locks for memory operations in this program, leading to perfect scaling up to six processors. Allocator G introduces super-containers, which can reduce performance due to additional complexity, but can also improve performance by reducing the number of containers. As a result, allocator G has similar performance to F2 on this setup. The lock-free operations in allocator H show no effect on performance. The coalescing allocator, which has all the benefits of allocators F1 and F2 except that it does not use container headers, scales perfectly up to six processors. Hoard, being very similar to F1, also scales perfectly up to six processors.

On setup B, shown in Figure 7.2, locks have less of an influence on performance, so allocators D and E do fairly well, possibly due to faster atomic instructions on the newer architecture. Active false-sharing, seen in allocator B, has a significant effect. The default Linux allocator, Ptmalloc, scales only slightly. Ptmalloc reduces contention by providing multiple heaps. Although it is expected that each thread should request all of its objects from one heap, by examining the addresses of the objects allocated, I discovered that threads are constantly switching heaps throughout the entire run of the program. Thus, although there is little contention for thread heaps, its scaling ability is limited by active false-sharing. Although Hoard has very good scaling, it has a slower runtime. After further investigation, I found the cause to be a difference in calculation in the thread-local free-list buffer. Hoard rounds up request sizes to at least the size of two pointers. In this benchmark, all allocations are for 8 bytes, which on a 64-bit machine is smaller than two



(a) Runtime



(b) Speedup

Figure 7.2: Scaling in Recycle on Setup B

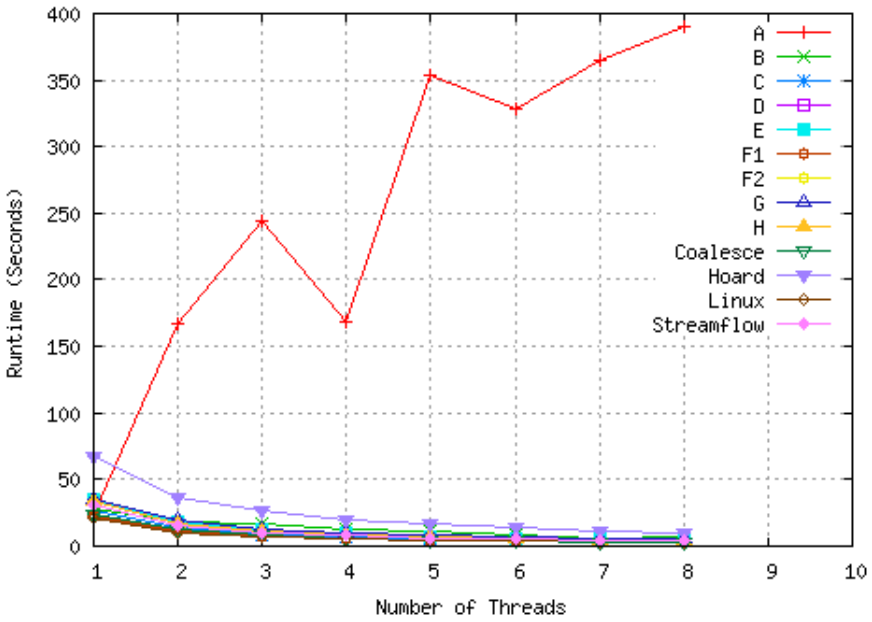
pointers. However, the rounding occurs after the check for objects on the thread-local free-list buffer, causing objects to be freed to a different bin list than they are requested from. Hence, in setups B and C, the buffer is not used, causing a slight reduction in runtime performance, but having no influence on scaling.

Setup C, shown in Figure 7.3 has similar results to setup B. Allocators A and B scale quite poorly, where active false-sharing occurs. The Linux allocator does fairly well on this machine. The performance of Recycle using Ptmalloc is highly dependent on the heap selected for each allocation. By examining the addresses of the objects allocated, I discovered that threads switch heaps a few times at the start of the program, but then quickly stabilize so that each thread uses one heap. However, when there are more threads running, the stabilization takes much longer. Thus, the scaling begins to level off and drops with six, seven, and eight threads. The new allocator to this test setup, Streamflow, scales fairly well and its performance is similar to that of other common architecture allocators.

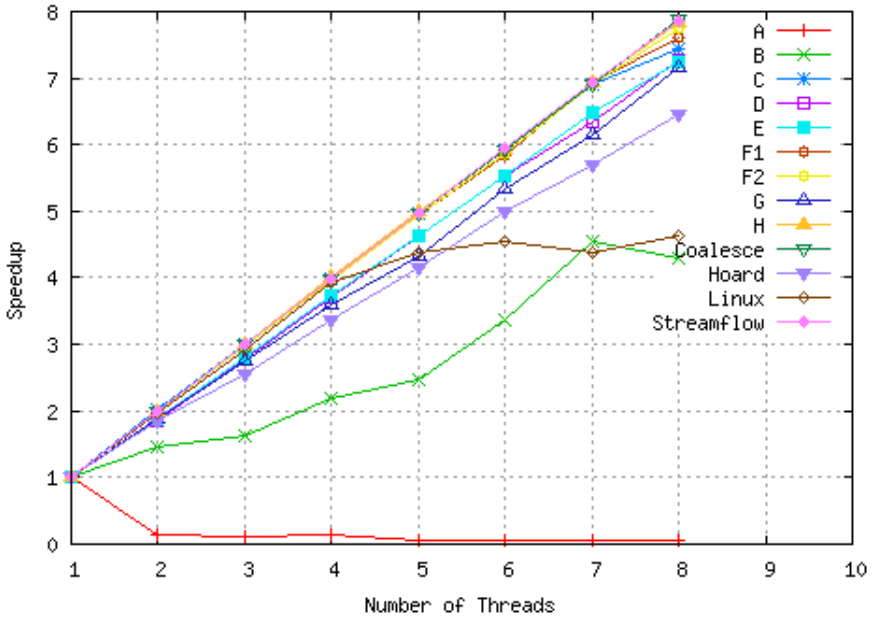
7.1.3 Consume

The Consume benchmark is not expected to scale as the number of threads increases since the amount of work also increases proportionally. As well, the producer is expected to become the bottleneck as more consumer threads are added. The purpose of this benchmark is to test for heap blowup in a situation where only one thread allocates objects, and other threads only deallocate objects. Two features expected to help the performance in this benchmark are an allocation buffer for the producer thread to allocate from, and a remote free-list for consumer threads to free to. The impact of these features is limited by the synchronization in this program.

In this benchmark, each consumer has an array, and in each iteration the producer thread fills each array with objects. When an array is filled, the consumer begins to



(a) Runtime



(b) Speedup

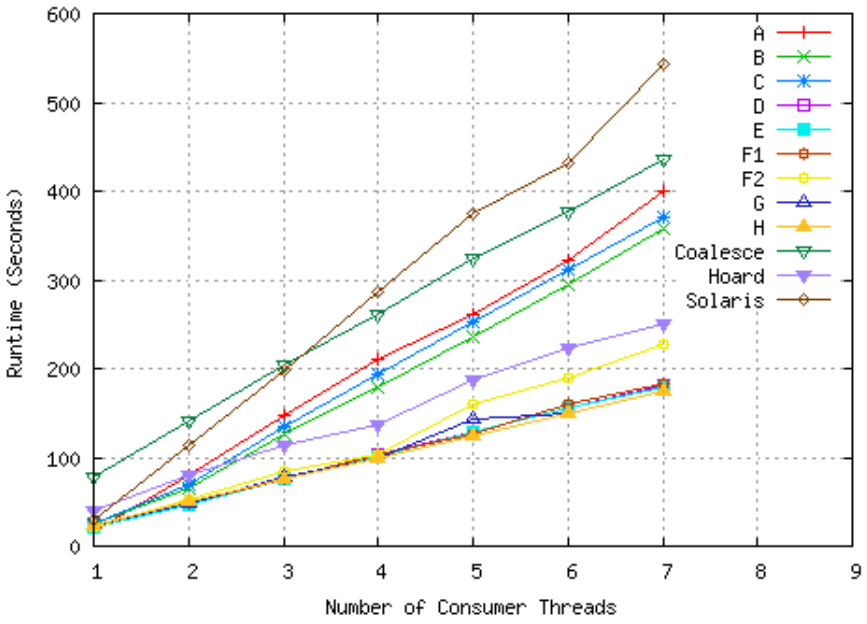
Figure 7.3: Scaling in Recycle on Setup C

deallocate the objects in its array. Once the producer has filled all consumer arrays, it waits for them to all be consumed before moving to the next iteration. If filling the array takes at least as long as consuming the array, then it is expected that only one consumer thread runs at a time. If filling the array takes less times than consuming the array, then multiple consumers may be running at once.

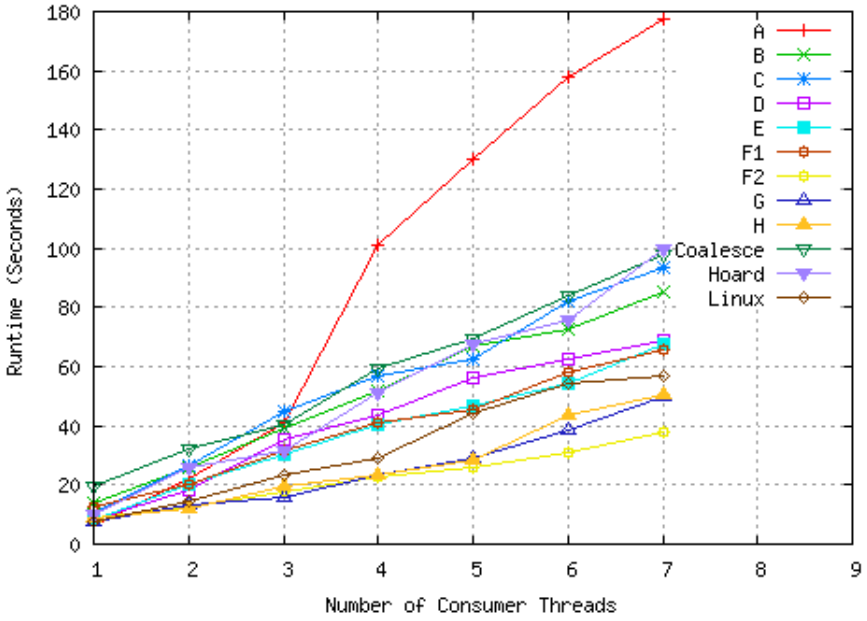
Figures 7.4 and 7.5 show the runtime of running the benchmark with 6 000 objects in each array and 5 000 iterations with one producer thread and one to seven consumer threads. Each data point represents an average of five test runs. The shorter the runtime, the better the performance. It was observed that the runtime with each allocator varies significantly for this benchmark on all test setups. The instability in performance is a result of contention for a shared resource. In all the tested allocators, the consumer threads contend for a shared resource: the single heap, the global shared heap, or the producer heap. Because the performance tests provide unstable results, it is impossible to make any detailed conclusions. Nevertheless, it is possible to make some general statements about the results.

In general, in allocators A, B and C, the producer thread must obtain each object it allocates individually, leading to generally poor performance. In allocator A, each object is obtained from `sbrk`. In allocator B, each object is obtained from the global heap. Although allocator C does use an allocation buffer, it prefers to reuse objects from the global heap before allocating a new allocation buffer. Hence, once there are enough objects in the global heap, the producer thread requests objects from the global heap rather than allocate a new allocation buffer. Thus, the producer thread must obtain each individual object from the global heap.

Allocators D to H all have similar and good linear performance. In these allocators, the producer thread obtains a container full of free objects from the global heap. Once

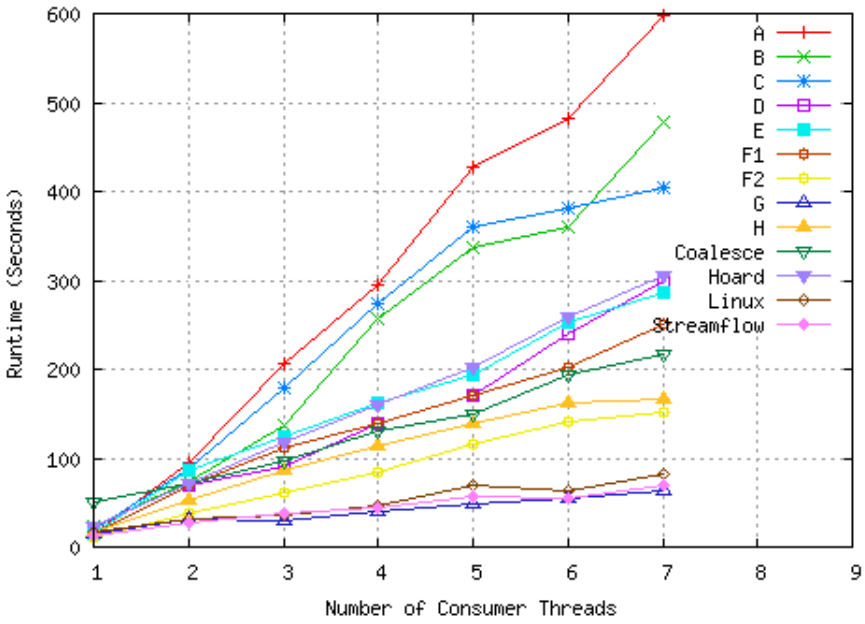


(a) Setup A



(b) Setup B

Figure 7.4: Runtime Performance in Consume



(a) Setup C

Figure 7.5: Runtime Performance in Consume

the consumer threads begin to deallocate objects, they are deallocated directly to the producer thread. Although the remote free-list in allocators F2, G, H, and Streamflow is expected to improve performance by allowing the producer to run without contending for locks, it is not always the case. The remote free-list removes locks for most producer operations. However, by allowing the producer to run faster, more consumer threads run at one time, causing more contention among consumer threads. In some situations, the consumer threads become a bottleneck, leaving the producer waiting for the consumer threads to finish consuming their arrays.

The performance of the coalescing allocator is slightly reduced because all deallocations are remote frees. In this implementation of the allocator, when the remote free-list is cleared, the objects are freed directly to the thread heap. This decision means the objects

are coalesced immediately with nearby objects, and not cached at their deallocated size, as when they are locally freed to the thread-local free-list buffer. Since all objects in Consume are the same size, performance is negatively affected by this implementation decision.

Although Hoard is similar to allocator F1, it generally performs slightly worse. Hoard allows any object to be freed to the thread-local free-list buffer, not just those owned by the local thread. Thus, a consumer thread using allocator F1 returns each object immediately to the producer thread, while a consumer thread using Hoard places the object on its thread-local free-list buffer. Eventually, the free-list buffer reaches its threshold of free objects and the buffer is cleared. When this happens, each object is individually freed to the producer's thread heap, acquiring a lock for each object. Thus, the thread-local free-list buffer only adds additional complexity in this program.

The default Solaris allocator has fairly poor performance, since all threads contend for the same heap. The default Linux allocator performs fairly well, since the producer thread can use another thread heap if its previous one is being used by a consumer thread to deallocate objects. Having the producer switch heaps allows the producer to avoid contention, and distributes the deallocations of consumer threads among several heaps, avoiding some contention.

7.1.4 False-Sharing Benchmarks

The false-sharing benchmarks test an allocator's ability to handle active and passive false-sharing. Only one form of active false-sharing is tested, since the number of free objects never reaches a significant size and objects are never freed to the global heap. The runtime performance of these benchmarks using different allocators on the different test setups are shown in Figure 7.6. In all three setups, allocators that avoid false-sharing have very stable runtime performance and scaling, while those that do not have random spikes of

slow runtimes due to hardware cache loading.

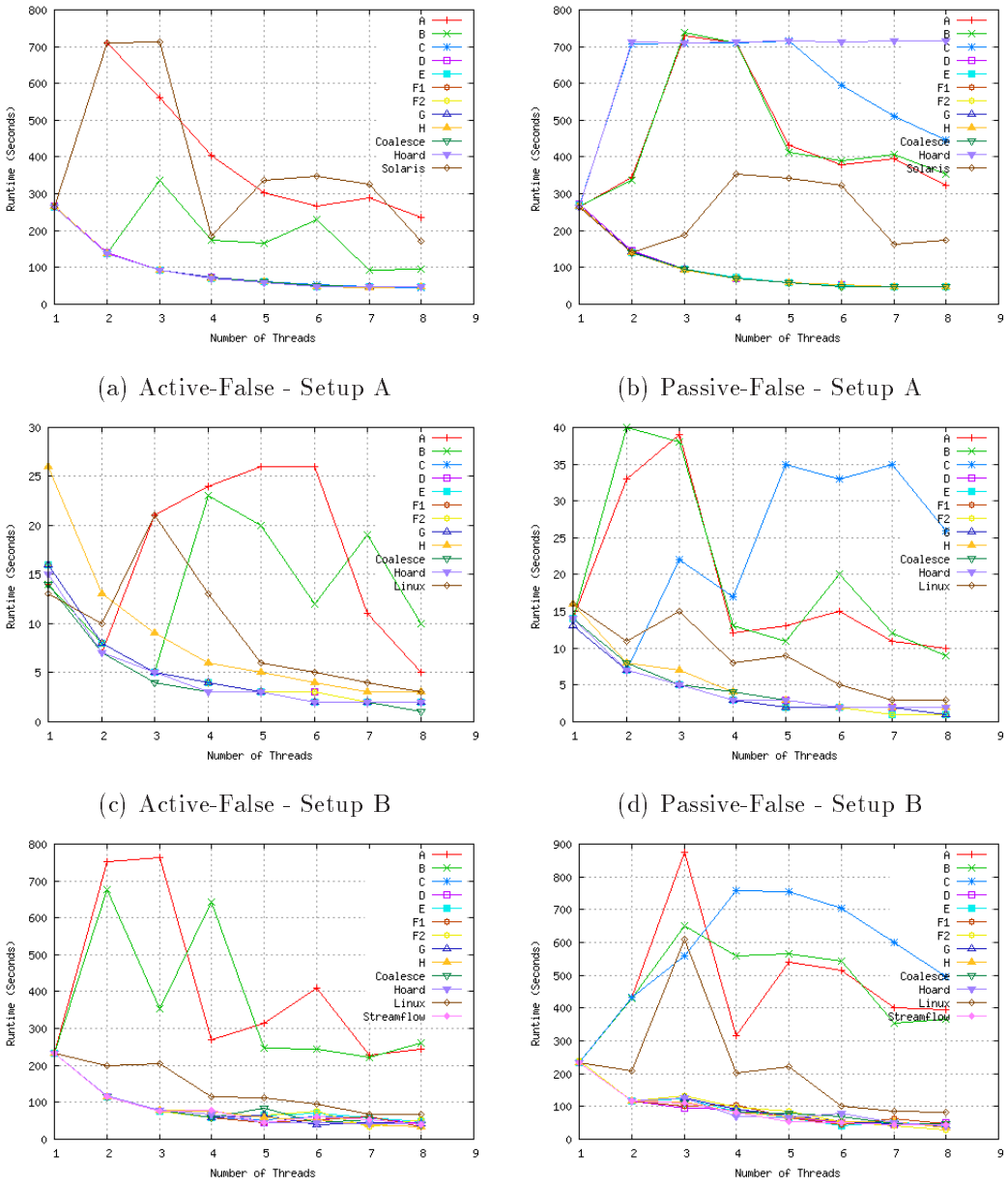
The first column shows that allocators using containers avoid all cases of active false-sharing tested by the active-false benchmark. The Linux allocator does exhibit some performance loss due to active false-sharing; however, it is not as extreme as allocators A, B, and the Solaris allocator. The effect in the Linux allocator is dependent upon the selection of a heap to satisfy the allocation request. If a thread allocates from the same heap most of the time, then it experiences little active false-sharing. Both Hoard and Streamflow prevent active false-sharing and scale well.

The second column shows that allocators A, B, and C exhibit poor scaling on all test setups because they allow passive false-sharing to occur. Solaris also scales fairly poorly, while the Linux allocator scales slightly better. The Linux allocator would prevent passive false-sharing if each thread always allocated from the same thread heap, but the results show that this is not the case. Allocators D to H, and Streamflow all prevent passive false-sharing, and all scale well.

The only anomaly is Hoard, which has poor scaling in passive-false on setup A, but good scaling in passive-false on setups B and C. Hoard's delayed ownership should allow passive false-sharing to occur through the thread-local free-list buffer. However, due to the rounding of object sizes on Hoard in 64-bit machines, described in Section 7.1.2, objects in the buffer are not used, eliminating passive false-sharing.

7.1.5 Larson

Larson calculates allocations per second, which should scale with the number of worker threads. An array is created for each working thread, and filled with objects of random sizes between 10 and 100 bytes. Each worker thread randomly selects an object to deallocate from its array and replaces it with a new allocation. Each thread repeats this process



(a) Active-False - Setup A

(b) Passive-False - Setup A

(c) Active-False - Setup B

(d) Passive-False - Setup B

(e) Active-False - Setup C

(f) Passive-False - Setup C

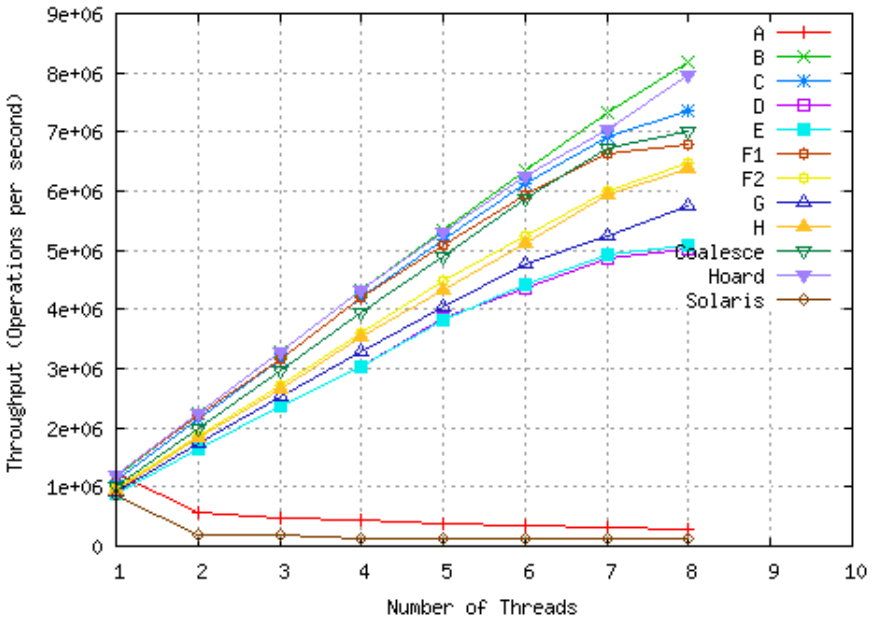
Figure 7.6: Scaling in False-Sharing Benchmark

several times, then creates a new thread to continue working on its array and dies. The benchmark is run with arrays of 10 000 objects and each thread replacing 100 000 objects.

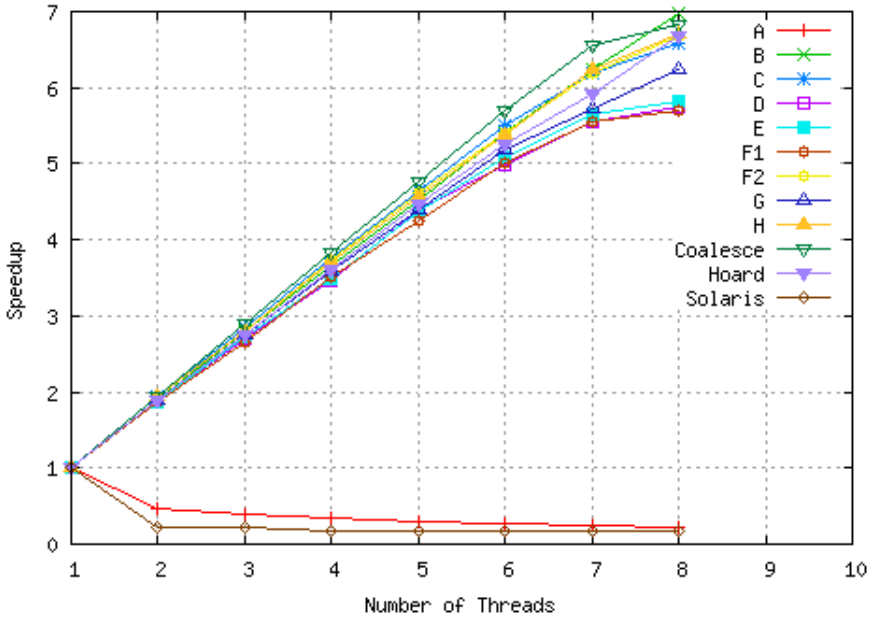
There is little contention for thread heaps in Larson, since deallocations are mostly for objects owned by the current thread, and the remaining few are for objects owned by a completed thread. Figures 7.7 to 7.9 show the throughput in allocations per second and the speedup for each test setup. The speedup is calculated as the throughput with `n` threads divided by the throughput with one thread. All the allocators scale quite well except for allocator A, and the default allocators on Solaris and Linux.

Most of the allocators scale well for the following reasons. Each allocated object is written and read just twice, so false sharing has a minimal impact. Approximately ten percent of the deallocations are for objects allocated by a thread that is no longer running (The first 10 000 deallocations replace all objects in the array, and the remaining 90 000 deallocations are of objects allocated by the current thread). Since thread heaps and ownership are inherited in the test allocators, objects that were allocated by a completed thread become owned by a new active thread. Thus, the remote free-list only helps a little in the scaling of throughput. Allocators F2, G, H, the coalescing allocator, and Streamflow all use a remote free-list and all have slightly better scaling.

Since most operations in this benchmark are allocations and local deallocations, the removal of locks from thread heaps improves performance. Allocators B and C have no locks on thread heaps because no ownership is enforced, and tend to have very good relative performance on all test setups. These allocators have the simplest implementation. Since allocations tend to stay within the local thread, once the thread has obtained enough objects to satisfy its allocations it simply reuses them without any additional contention or complexity. Allocator F1 does not require a lock for operations on the thread-local free-list buffer, which covers most operations in this benchmark. Allocators F2, G, and H all

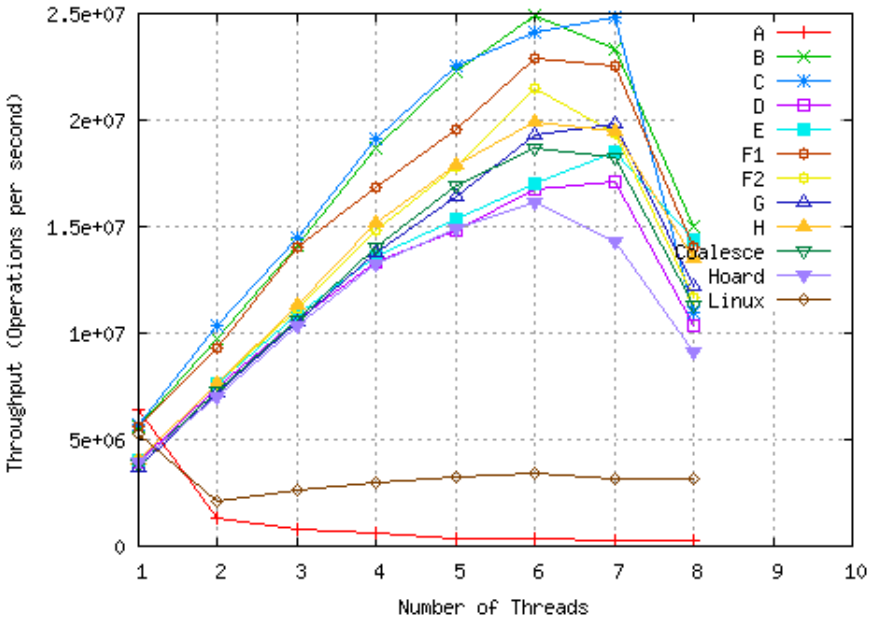


(a) Setup A

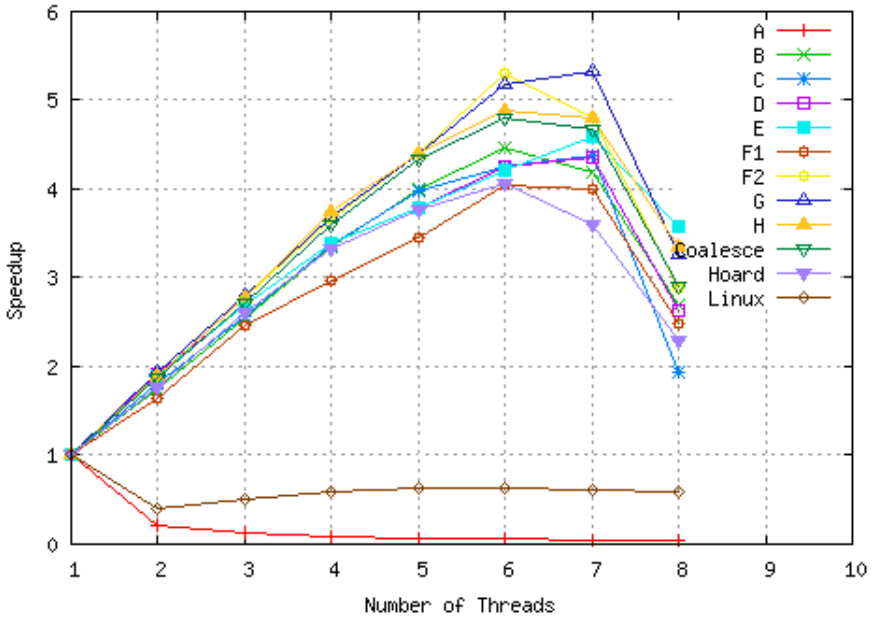


(b) Setup A - Speedup

Figure 7.7: Scaling in Larson on Setup A

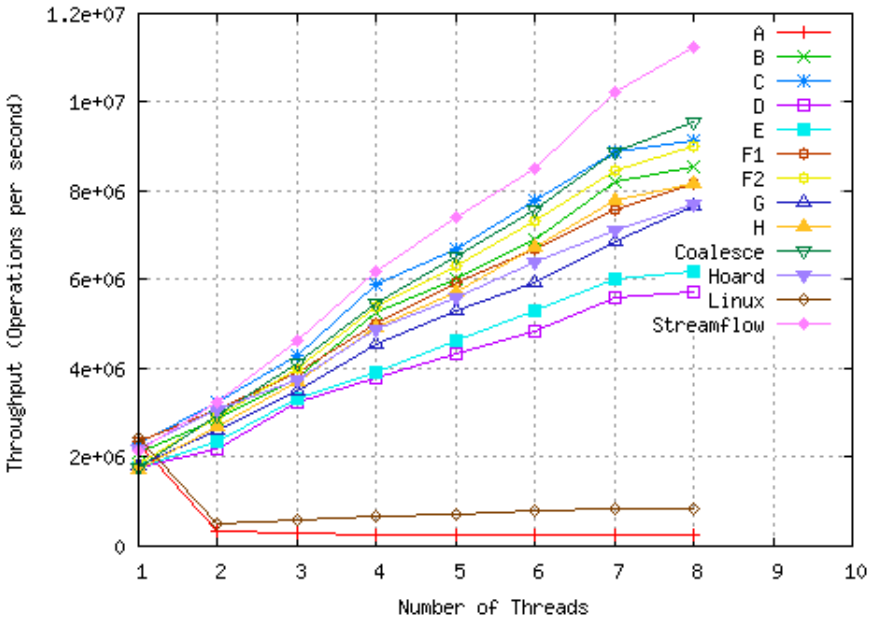


(a) Setup B

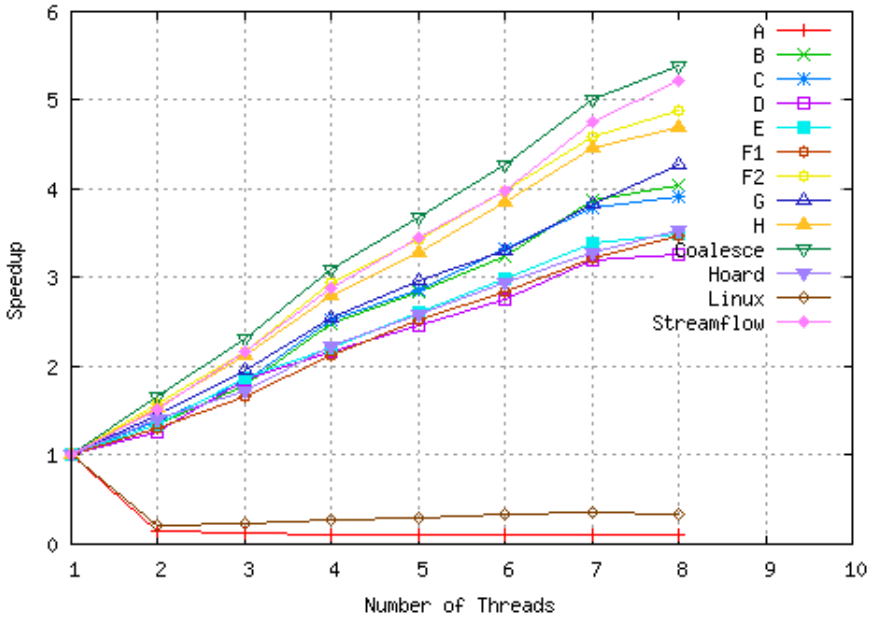


(b) Setup B - Speedup

Figure 7.8: Scaling in Larson on Setup B



(a) Setup C



(b) Setup C - Speedup

Figure 7.9: Scaling in Larson on Setup C

avoid locking for local operations with the use of a remote free-list. Thus, only allocators A, D and E lock local operations, giving them slightly worse performance on all test setups.

Allocators A and Solaris exhibit poor scaling simply because all allocations and deallocations are performed on the same heap. The Linux allocator does not scale very well, even though multiple heaps are used. Each time an allocation request is made, a thread attempts to use its previously used heap; however, when a new thread is created it attempts to acquire a lock for each heap until it finds one available. Since Larson creates many new threads, one for every 100 000 objects created, the process of establishing a heap for a thread happens frequently. As was noted in the Recycle benchmark, it can take several allocations before a thread stabilizes to using a single heap. In this benchmark, there is no opportunity for this stabilization to occur, reducing performance and limiting throughput.

7.2 Fragmentation

The internal and external fragmentation experienced by a program depends on the allocator, and can only be accurately measured from within the allocator. Hence, the test allocators are modified to include a logging layer where they record allocation and deallocation requests along with the changes in internal and external fragmentation.

The internal fragmentation is measured as the headers, padding and spacing around allocated objects, while external fragmentation is all other memory reserved from the operating system that is not allocated to the program. To determine the internal and external fragmentation, three measures are recorded in the logs. These measures are: the allocation request size, the amount of memory used by the allocator to satisfy each request, and the amount of memory reserved from the operating system through calls to `sbrk` and `mmap`. A running total is calculated to determine the total memory, internal

fragmentation, and external fragmentation at any time in the program by increasing the total at each allocation and decreasing the total at each deallocation.

Johnstone and Wilson discuss four different ways to calculate the fragmentation of a program [JW99]. The first method is to average the fragmentation across all points in time. The second method is to use the fragmentation at the point in time when the program has the largest amount of bytes in use. The third method is to count the fragmentation at the point when the most memory is being held from the operating system. The fourth method is to measure the difference between the high watermark (the most amount of memory reserved from the operating system) and the most amount of memory used by the program [JW99]. Each method of measurement has its drawbacks, but the fourth measure is used because it avoids extreme measures of fragmentation (i.e. a best or worst case), which may be misleading.

Fragmentation measurements for the single-threaded and some multi-threaded benchmarks are collected and discussed in the next sections. Of the multi-threaded benchmarks, only Recycle and Consume are analyzed. Fragmentation in Larson is not measured since the number of allocations varies depending on the performance of the program, making it difficult to compare fragmentation results for different memory allocators. The active-false and passive-false benchmarks are also left out of this measurement since their purpose is to test performance, and their memory usage patterns are very simple and uninteresting.

7.2.1 Fragmentation in Single-Threaded Benchmarks

Since the allocators are very similar, the main differences that effect fragmentation are the type of containers and coalescing. Per-thread heaps do not influence fragmentation, since the single-threaded benchmarks only require a single heap. Differences in fragmentation caused by ownership and container movement restrictions are not noticeable in

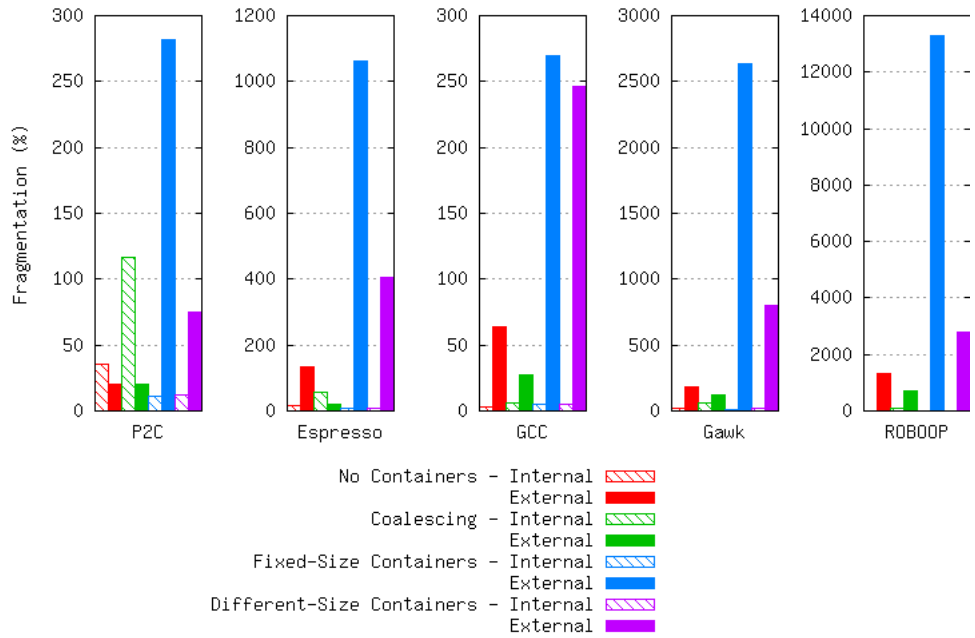


Figure 7.10: Fragmentation in Single-Threaded Benchmarks

single-threaded programs. Thus, the allocators can be separated into four categories: no containers, coalescing, fixed-size containers, and different-size containers. The No Containers category includes allocators A and B, the Coalescing category is the coalescing allocator, the Fixed-Size Containers category includes allocators C through F2 and H, and the Different-Size Containers category is allocator G.

Figure 7.10 shows the internal and external fragmentation calculated using the fourth method of measurement for each of the single-threaded benchmarks for one allocator in each of the four categories. There is negligible variation in fragmentation among the different allocators in each category.

As expected, coalescing has more internal fragmentation from than an allocator with no containers, since more management information is required around each object. However, coalescing has less external fragmentation than an allocator with no containers, since

freed objects are more likely to be reused. In P2C, Gawk, and ROBOOP most request sizes are small and common leading to minimal external fragmentation in an allocator with no containers. Thus, coalescing does little to reduce external fragmentation in these benchmarks. Espresso and GCC have more large and unique request sizes, leading to a significant decrease in external fragmentation when adding coalescing to an allocator with no containers.

Adding containers has the effect of decreasing internal fragmentation, while increasing external fragmentation by a significant amount. Using various sized containers does decrease external fragmentation, while also increasing internal fragmentation a little, since there are more containers.

7.2.2 Fragmentation in Multi-Threaded Benchmarks

Both Recycle and Consume are run with a smaller overall number of allocations in order to obtain manageable logs, while still performing the same function. Recycle is reduced to allocate a total of 100 000 objects, and Consume is run with an array of 600 objects and for only 500 iterations. The maximum program allocation size in both Recycle and Consume is quite small. They both allocate and deallocate a large number of small objects, without ever having a large number of allocated objects at one time. The differences in fragmentation caused by ownership and container movement restrictions are not noticeable. Thus, the same categories used in Section 7.2.1 are also used to compare fragmentation in Recycle and Consume.

Figure 7.11 shows the fragmentation of the four categories in Recycle when run with one, two, and four threads. As the number of threads increases, the number of objects allocated at once increases. Since each thread holds the same number of allocated objects regardless of the number of threads, the number of allocated objects is multiplied by the

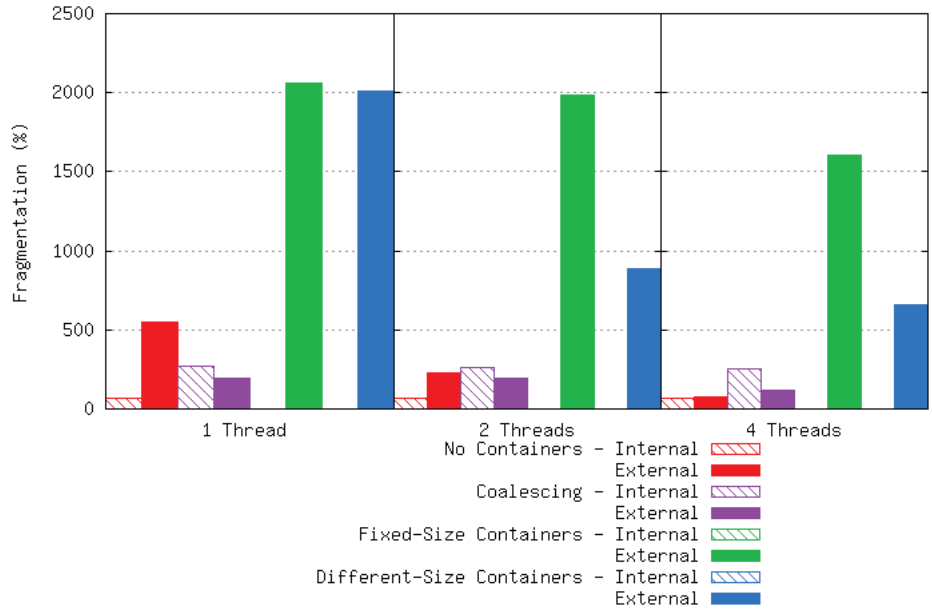


Figure 7.11: Fragmentation in Recycle

number of threads. As more objects are allocated, the reserved memory on the heap shrinks, until at some point a more reserved memory is allocated. External fragmentation decreases as the reserved space shrinks, and increases when it grows. Thus, with more allocated objects, external fragmentation decreases unless another chunk of reserved space is allocated. Internal fragmentation is related to the number of objects. As the number of objects increases, the internal fragmentation increases. However, since nearly all objects are the same size in this benchmark, the internal fragmentation relative to the number of allocated objects does not change as the number of objects change.

The number of allocated objects is slightly different each time the benchmark is run due to non-deterministic timing characteristics, but the difference is very small. As expected, adding coalescing to an allocator with no containers increases internal fragmentation, but tends to decrease external fragmentation. The expected benefit of coalescing is very small

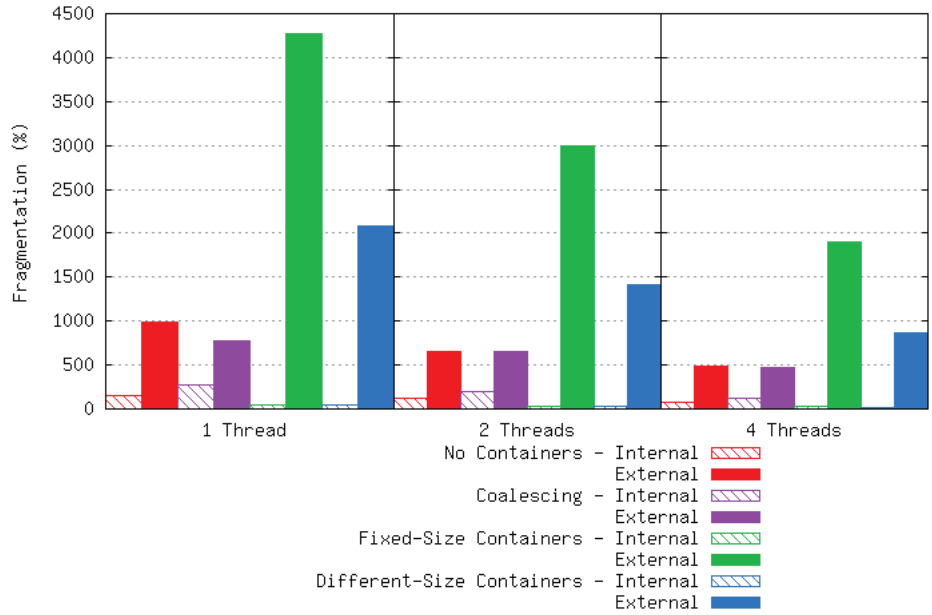


Figure 7.12: Fragmentation in Consume

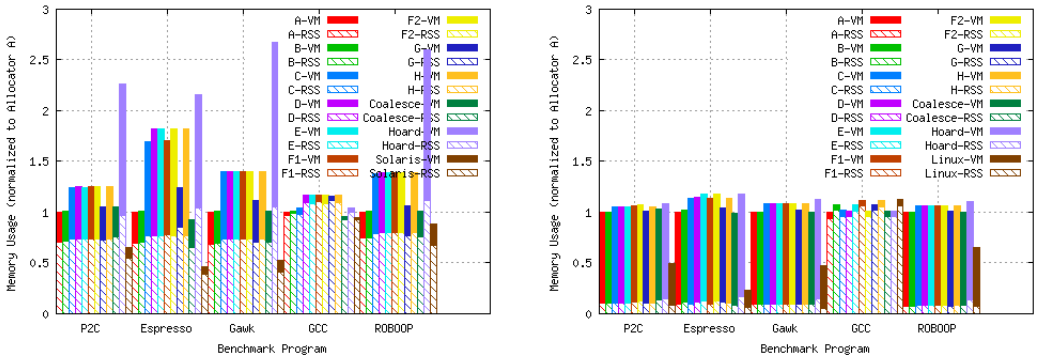
in this benchmark, since almost all objects are the same size. In fact, in some cases, it appears the external fragmentation is increased slightly, but this is simply due to the differences in the number of allocated objects at one time.

Object containers decrease internal fragmentation, but at the cost of increasing external fragmentation. The increase in number of allocated objects only slightly decreases external fragmentation in the case of fixed-size containers, but makes a much larger difference in the case of different-size containers. In the case of one thread, the reserved space is increased by the minimal amount, leaving no potential for improvement with different-size containers. With more threads, the fixed-size containers require that more containers be allocated for each new thread, causing more reserved space to be allocated. With different-size containers, each thread receives a small container that takes up a small portion of the reserved space, avoiding a request for more reserved memory.

Figure 7.12 shows the fragmentation of the four categories in Consume when run with one, two, and four consumer threads. As the number of consumer threads increases, the number of objects allocated increases. Thus, the internal fragmentation stays close to the same, while the external fragmentation drops significantly. Adding coalescing to an allocator with no containers is not expected to provide any savings since nearly all objects in this benchmark are the same size, but coalescing does decrease external fragmentation slightly. Object containers decrease internal fragmentation significantly, but also increase external fragmentation significantly. Using different-sized containers helps to reduce the increase in external fragmentation.

7.3 Memory Usage

Fragmentation causes the running program to consume more operating system resources. Two measures that indicate the memory resource usage of the program are virtual memory size and resident set size. The virtual memory size is the number of pages reserved by the program, while the resident set size is the number of bytes that have been brought into main memory. Virtual memory gives an indication of the third method of measuring fragmentation, which is the fragmentation at the point when the most memory is being held from the operating system. Although the resident set size is measured, it is highly dependent on the program and it is difficult to make any conclusions based on this measure. The `top` command provides these two measures. Hence, the memory usage can be obtained for any memory allocator without modifying any source code. This information is collected for both single and multi-threaded benchmarks by querying the `top` command at one second intervals when running a benchmark program and storing the highest recorded measure.



(a) Setup A (b) Setup B

Figure 7.13: Memory Usage in Single-Threaded Benchmarks

7.3.1 Memory Usage in Single-Threaded Benchmarks

Figure 7.13 shows the virtual memory and resident set size normalized to that in allocator A. Setup C is nearly identical to setup B, and is therefore not shown. The virtual memory usage shows that containers increase memory usage. For most benchmarks, using different-sized containers reduces this increase.

In setup A, the Hoard allocator, in most cases, increases both the virtual memory and resident set size, which is due to the method of calling `mmap` in Hoard running on Solaris. On Solaris, Hoard `mmap`s a large number of containers at once, and places each of them on a list. This operation increases both the virtual memory and the resident set size due to initialization.

The default Solaris and Linux memory allocators generally uses less virtual memory and have a smaller resident set size, which is due to the method of loading in allocators. The test allocators are all created as dynamically loadable libraries. Dynamically loading this additional library increases the memory usage by a small amount (less than 1 MB). However, since the overall memory usage is quite small, the relative difference caused by

loading the additional library appears to have a large effect.

The one benchmark that is noticeably different from the rest is GCC. GCC has a very similar virtual memory usage and resident set size for all allocators. This behaviour is because GCC allocates several very large objects. Thus, a larger portion of the virtual memory is always in use by the program.

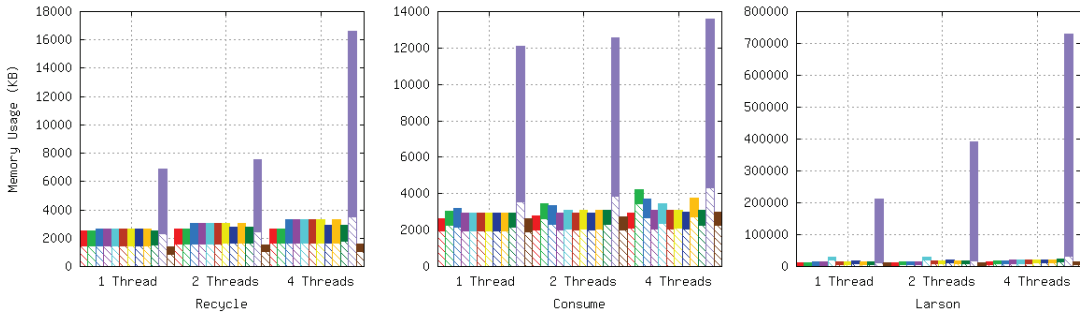
7.3.2 Memory Usage in Multi-Threaded Benchmarks

Each benchmark is shown with one, two, and four threads run on the different test setups. Figure 7.14 shows the results from running Recycle, Consume, and Larson. The false-sharing benchmarks are left out of these graphs since they are intended to test performance, and do not have interesting memory usage characteristics.

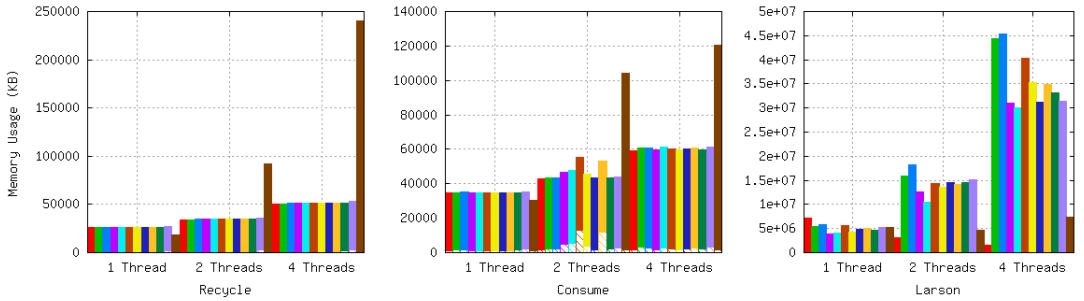
As explained in Section 7.3.1, Hoard has a significantly larger virtual memory and resident set size for all benchmarks on Solaris. The difference in memory usage among all other allocators is quite small in setup A. The default Solaris allocator uses less memory, as explained in Section 7.3.1, because it is not dynamically loading additional libraries.

On setup B, the Linux allocator increases memory usage significantly as the number of threads are increased in the Recycle and Consume benchmarks, due to a large initial allocation buffer for each heap. Specifically, each additional heap starts with approximately 64 MB of memory, leading to large increases in virtual memory when using multiple threads. In Larson, thread heaps grow beyond their initial size, leading to variations in virtual memory that depend on the objects allocated.

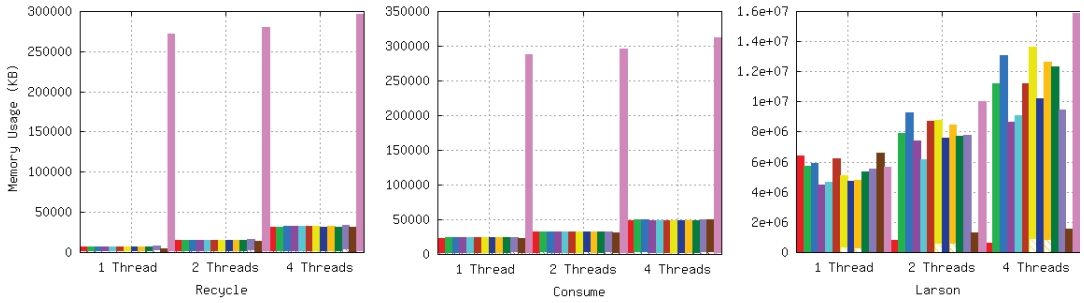
On setup C, the Linux allocator starts the thread heaps at a smaller size, leading to similar memory usage to other allocators. The Streamflow allocator uses a significant amount of memory in all benchmarks, which may be caused by a large BIBOP table (see Section 4.5) on a 64 bit processor. The difference is less significant in Larson where memory



(a) Setup A



(b) Setup B



(c) Setup C

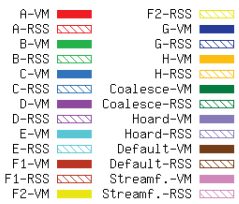


Figure 7.14: Memory Usage in Multi-Threaded Benchmarks

usage is quite high for all allocators.

7.4 Analysis

Running the benchmark programs with existing and test allocators leads to several conclusions regarding runtime performance and memory usage. Allocator A is a very basic memory allocator providing fairly low fragmentation, but poor performance and limited scaling in all the multi-threaded benchmarks. Allocator B, with its per-thread heaps, increases memory usage slightly on some multi-threaded benchmarks. Although allocator B improves performance in programs like Recycle where threads work independently, it does not avoid active and passive false-sharing.

Allocator C introduces containers, improving performance in some multi-threaded programs by avoiding most active false-sharing. However, containers cause a large increase in external fragmentation (by an average of approximately 400%), with only a relatively minor reduction in internal fragmentation. This larger external fragmentation causes up to a 75% increase in virtual memory usage depending on the program.

Allocators D and E introduce ownership and container-movement restrictions, respectively. These allocators improve performance in some multi-threaded programs by avoiding passive false-sharing in addition to active false-sharing. A consequence of ownership is locks on thread heaps, which reduce performance and are a source of contention in some benchmarks. Fragmentation and memory usage remain very similar to allocator C.

Allocator F1 introduces a thread-local free-list buffer to reduce contention for thread heaps. The buffer also provides additional performance benefits by placing objects on simple free-lists and avoiding the container-based free-lists. The buffer is most effective in programs like Larson where there is a mix of local and remote deallocations. Allocator

F2 uses remote free-lists to reduce contention on thread heaps. The remote free-list is most effective in programs like Larson, where there are a significant number of remote deallocations. Allocators F1 and F2 have very similar fragmentation and memory usage to allocators C, D, and E.

Allocator G uses super-containers to allow different-sized containers. External fragmentation is reduced by at least 50% over fixed-sized containers in most programs. However, runtime performance suffers slightly in some benchmark programs due to the additional complexity. External fragmentation remains significantly higher than allocators without containers, although this translates to a very small increase in virtual memory usage of at most 10% in the programs tested. Allocator H adds lock-free operations to the remote free-lists in allocator F2. The lock-free operations provide insignificant performance benefits in the benchmark programs tested.

The coalescing allocator avoids both active and passive false-sharing by enforcing ownership and controlling object movement to the global heap. The thread-local free-list buffer improves performance in several of the benchmark programs by caching objects at their allocated size. This effect is made clear by the Consume benchmark in which the remote free-list bypasses the thread-local free-list buffer when clearing the remote free-list having a negative impact on performance. Regardless, the coalescing allocator performs relatively well in all benchmark programs, while providing low fragmentation and memory usage. External fragmentation is reduced from the basic allocator in all programs, while internal fragmentation increases by a small amount in some programs.

If a single allocator needs to be selected for all programs, allocator F2 provides a good compromise between speed, memory usage, and code complexity for both sequential and a mix of concurrent programs. Table 7.2 shows a summary of how the F2 allocator compares to the default allocator on each test setup with respect to memory usage and runtime

Table 7.2: Allocator F2 Compared to the Default Allocator

	Runtime	Throughput	Memory Usage	
Test Setup	Recycle	Larson	Single-Threaded	Consume
A	85% faster	2627% increase	125% increase	10% increase
B	7% faster	338% increase	142% increase	31% reduction
C	2% faster	655% increase	82% increase	1% increase

performance. The first column indicates the average percent reduction in runtime from the default allocator when Recycle is run with one to eight threads. The second column shows the average percent increase in calculated throughput over the default allocator when running Larson with one to eight threads. The third column shows the average percent reduction in memory usage from the default allocator from the tested single-threaded benchmarks. The final column shows the average percent reduction in memory usage from the default allocator when Consume is run with one, two, and four threads. A faster runtime and increase in throughput indicates that F2 has better runtime performance than the default allocator. A reduction in memory usage indicates that F2 has better memory usage than the default memory allocator.

As an alternative single allocator, the coalescing allocator also performs relatively well with smaller memory usage. The coalescing allocator is likely to benefit from clearing the remote free-list to the thread-local free-list buffer, rather than the thread heap. Table 7.3 shows how the coalescing allocator compares to the default allocator on each test setup with respect to memory usage and runtime performance. The information follows the format of Table 7.2.

Table 7.3: Coalescing Allocator Compared to the Default Allocator

	Runtime	Throughput	Memory Usage	
Test Setup	Recycle	Larson	Single-Threaded	Consume
A	90% faster	2897% increase	53% increase	10% increase
B	42% faster	310% increase	119% increase	31% reduction
C	22% faster	682% increase	53% increase	1% increase

7.5 Summary

This chapter provides results from comparing different existing allocators and test allocators. The next chapter summarizes the findings of this thesis and provides some conclusions.

Chapter 8

Conclusions

8.1 Memory–Allocation Challenges

All memory allocators are concerned with providing fast performance while supporting good locality, limiting fragmentation and preventing heap blowup. Additionally, multi-threaded memory-allocators must provide mutual exclusion while reducing contention, avoiding false sharing, and preventing additional forms of potential heap blowup. Several features are presented as means for addressing the concerns of a multi-threaded memory allocator. These features include per-thread heaps with a global heap, object ownership, object containers, allocation buffers, thread-local free-lists, remote free-lists and lock-free operations. Evaluating the performance of these features can assist in designing a memory allocator to achieve certain goals.

8.2 Method of Analysis

Several existing allocators are presented along with a description of the multi-threaded features they employ. In addition to these existing allocators, a set of test allocators are implemented, each employing a different set of features. Through the use of a test suite composed of single and multi-threaded benchmark programs, these allocators are analyzed in order to evaluate the effect of different features.

It is determined that each feature has different effects on the challenges that a memory allocator can address. Depending on the behaviour of the program, different features can be applied to the memory allocator in order to achieve different goals of runtime performance, scaling, and fragmentation. Depending on the needs of a particular program, the best performance can be achieved through the use of a specific set of features.

8.3 Analysis Results

Any multi-threaded program that is memory-allocation intensive benefits from the use of multiple heaps to reduce contention. In general, the global heap is essential to maintaining a balance of free objects among heaps.

Programs in which threads independently allocate and deallocate objects, like Recycle, gain significant benefits from the use of per-thread heaps to reduce contention. Additionally, active false-sharing avoidance through the use of an allocation buffer greatly improves performance in these applications. Programs that allocate and deallocate objects within the same thread do not require the overhead of enforced ownership by the allocator.

Programs in which objects are frequently shared among threads and allocated and deallocated by different threads gain significant benefits from ownership to avoid passive false-sharing. Using a remote free-list to avoid locks on thread heaps when enforcing

ownership greatly improves the performance of the allocator.

Programs that use several objects of the same size in their working set benefit from the use of containers. In such programs, containers reduce internal fragmentation and improve cache usage. Programs that use different sized objects in their working set may prefer to use an allocator with coalescing, where the objects may be placed closer together in memory. Programs using a large range of object sizes in different working sets can benefit from using an allocator with different-sized containers. Some external fragmentation can be avoided by using different-sized containers at the cost of a slight reduction in performance. Allocators using containers and ownership can also improve performance by placing free lists of objects on containers.

The thread-local free-list buffer can improve the performance of some allocators by allowing some objects to be cached and easily accessed by the local thread, which is especially important in the coalescing allocator. Although remote free-lists have a greater impact on performance scaling than the thread-local free-list buffer, using the two features in combination can provide the greatest performance benefit, as observed in the coalescing allocator. Using lock-free operations may improve performance in certain situations, on certain machines, but the tests presented did not find any significant differences when using lock-free operations.

Thus, I recommend the following features for a general-purpose memory-allocator: per-thread heaps with a global shared-heap, object ownership, object containers (with allocation buffers, container based free-lists, and restricted container movement), thread-local free-list buffers, and remote free-lists. This memory allocator demonstrated very good performance in several multi-threaded programs, improving performance by a factor of 100 in some benchmarks. The cost is an increase in memory usage that is typically less than 200% in the tested benchmarks. On systems with limited memory, I recommend a

coalescing allocator with the following features: per-thread heaps with a global shared-heap, object ownership, thread-local free-list buffers, and remote free-lists. This alternate allocator provides similar performance benefits with slightly reduced memory usage.

8.4 Future Work

The presented test suite is composed of real single-threaded applications, but only micro-benchmark multi-threaded programs. In searching for real multi-threaded programs, none were found to be allocation-intensive. Some multi-threaded programs provide their own special form of memory allocation within the program. As future work, converting some of these programs to work with general-purpose memory-allocators, or a further search to find additional memory-intensive multi-threaded programs may provide interesting analysis of allocation behaviour in full-featured multi-threaded applications. Such analysis would also provide insight into the effects of containers on locality and paging in real multi-threaded programs.

Appendix A

Trace Graphs

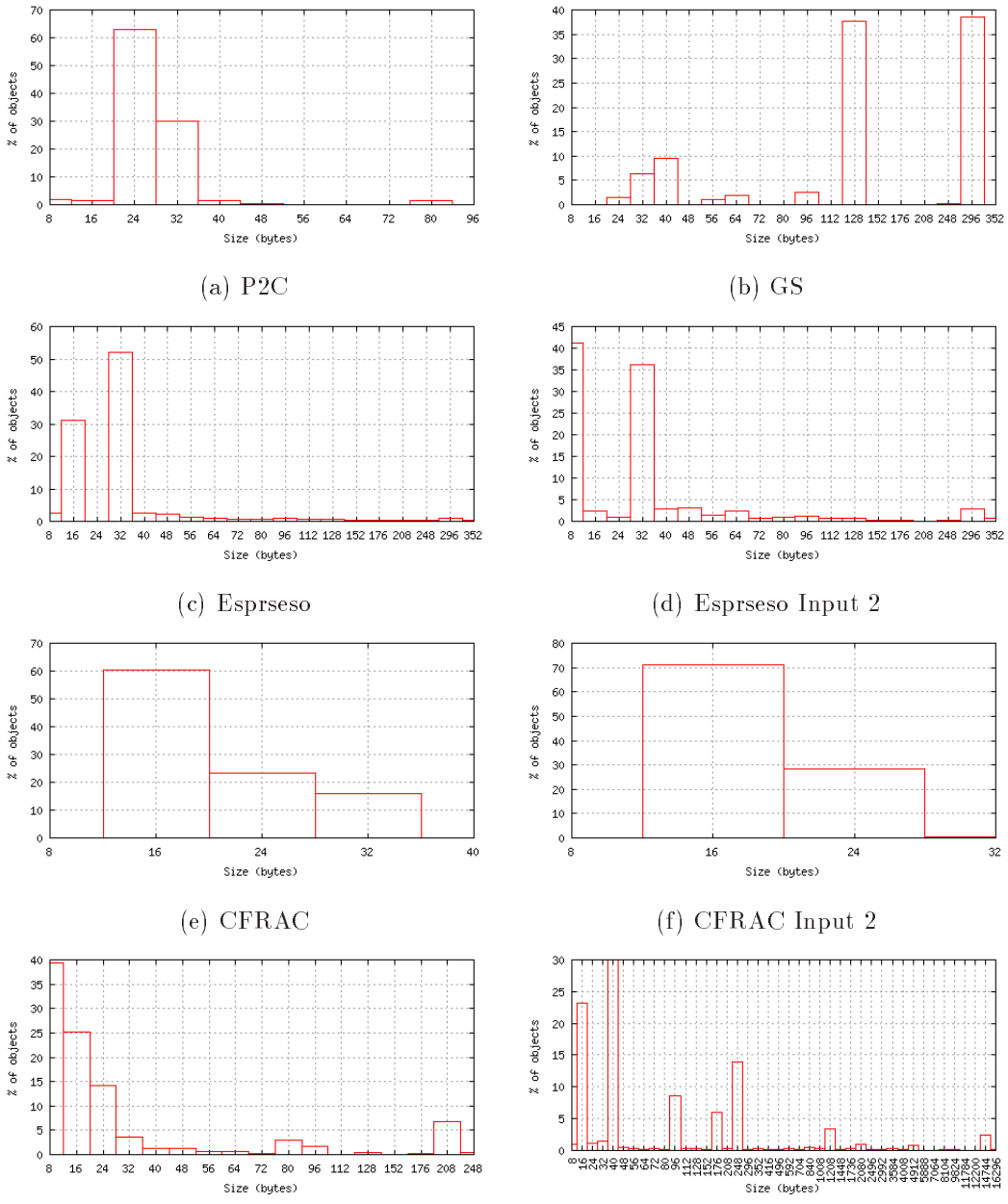
Figures A.1 and A.2 show a distribution of objects among bin sizes. Each graph shows the portion of objects allocated by the program that fall into each bin size. Figures A.3 to A.4 show the bin size of objects relative to the time in the program when they are allocated. To reduce the number of data points on the graph, several nearby points are condensed into one point, with the different colours indicating the number of objects condensed into one point.

Figures A.5 and A.6 show a cumulative distribution of the lifetime of objects in each program. Each point in the graph indicates the portion of objects that have a lifetime equal to or shorter than that time. Figures A.7 and A.8 show the lifetime of objects over the runtime of the program. Again, to reduce the number of data points on the graph, several nearby points are condensed into one point, with the different colours indicating the number of objects condensed into one point. Figure A.9 shows the lifetime of objects relative to the time they are allocated in the program for only short living objects over a short period of time, for a select set of programs.

Figures A.10 and A.11 show the cumulative distribution of interarrival times of all

memory operations, just malloc requests, and just free requests for each program. Each point in the graph indicates the portion of requests that arrive with equal to or less than the indicated time from the previous request.

Figures A.12 to A.14 show the allocation footprint over the runtime of the program. Each point in the graph indicates the amount of dynamic memory in use by the program at that point in time of the program.



(a) P2C

(b) GS

(c) Esprseso

(d) Esprseso Input 2

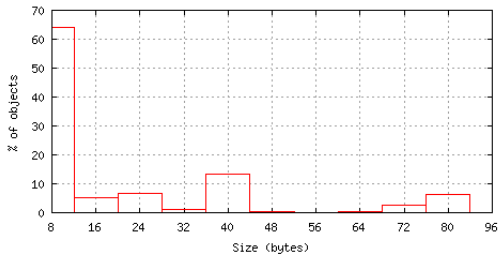
(e) CFRAC

(f) CFRAC Input 2

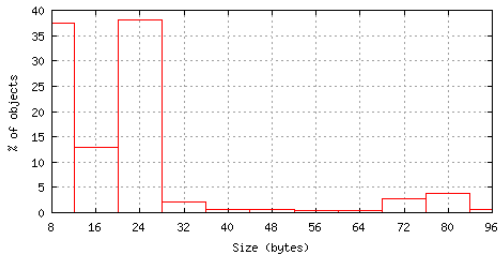
(g) GMake

(h) GCC

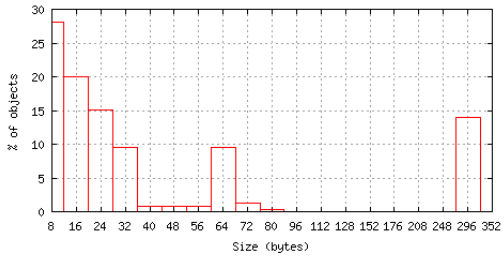
Figure A.1: Bin Size Distribution



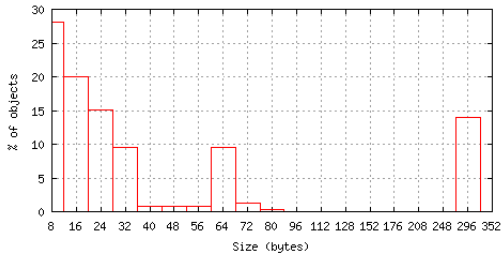
(a) Perl



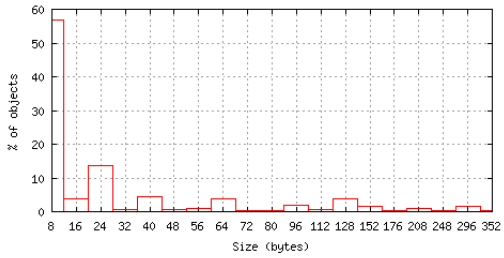
(b) Perl Input 2



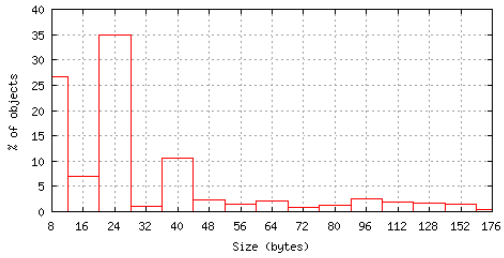
(c) Gawk



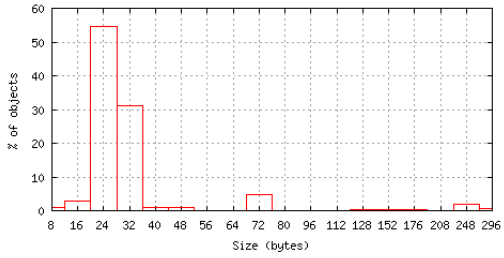
(d) Gawk Input 2



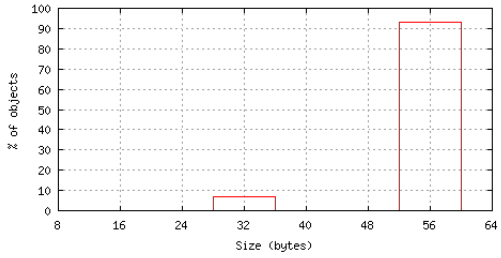
(e) XPDF



(f) XPDF Input 2

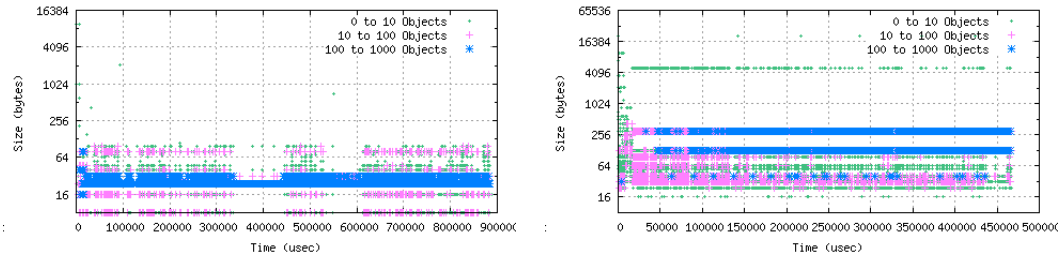


(g) ROBOOP



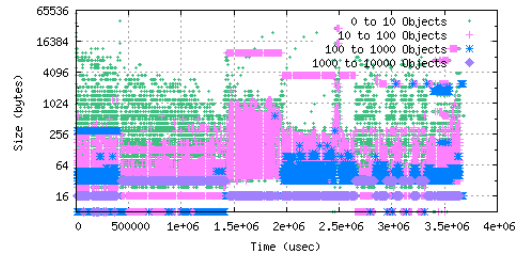
(h) Lindsay

Figure A.2: Bin Size Distribution 2

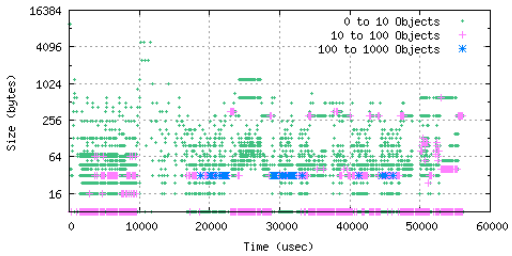


(a) P2C

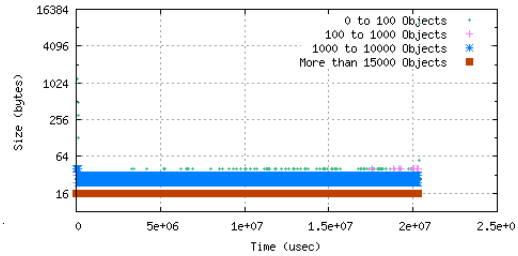
(b) GS



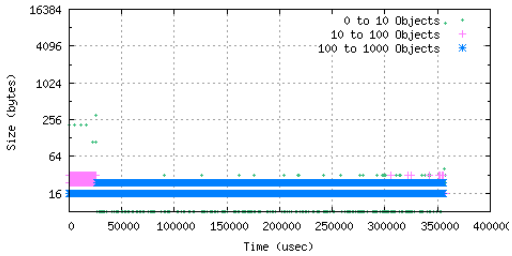
(c) Espresso



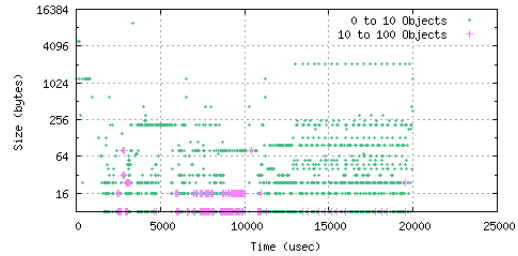
(d) Espresso Input 2



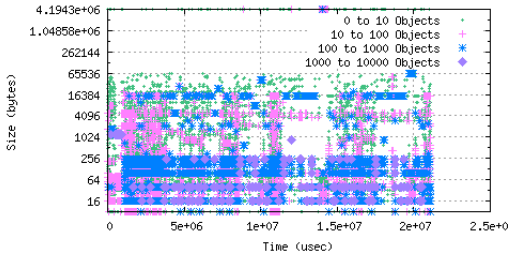
(e) CFRAC



(f) CFRAC Input 2

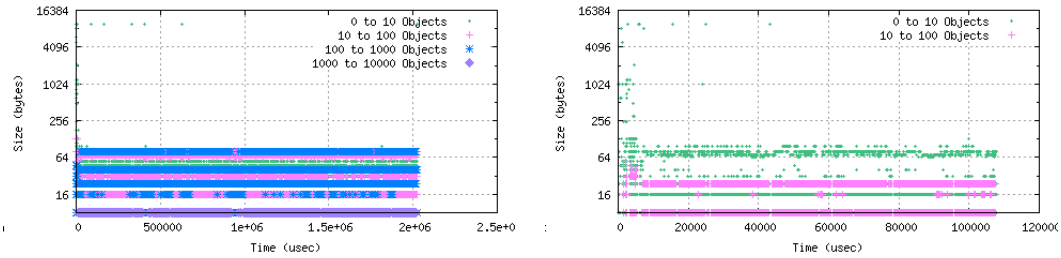


(g) GMake



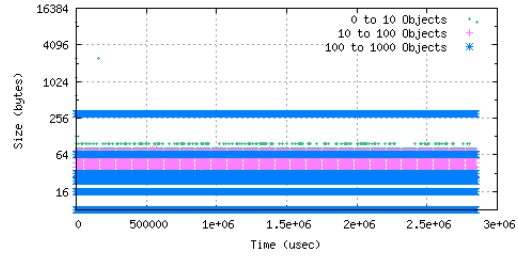
(h) GCC

Figure A.3: Bin Size Over Time

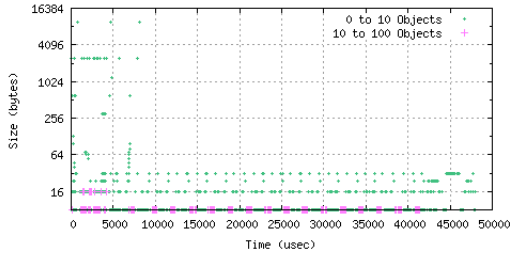


(a) Perl

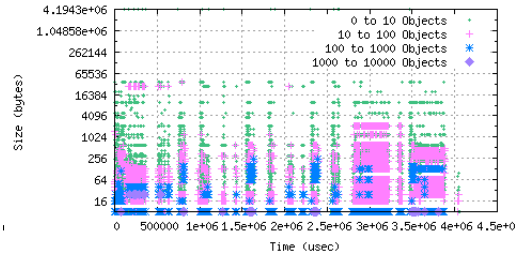
(b) Perl Input 2



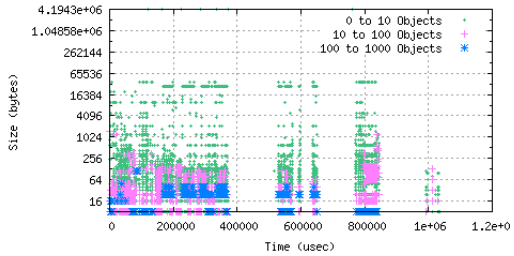
(c) Gawk



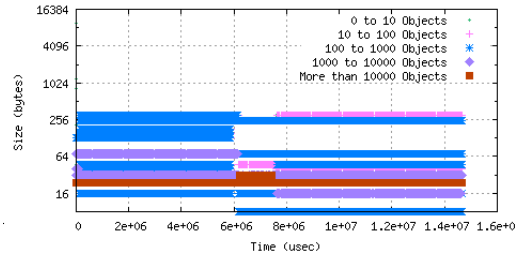
(d) Gawk Input 2



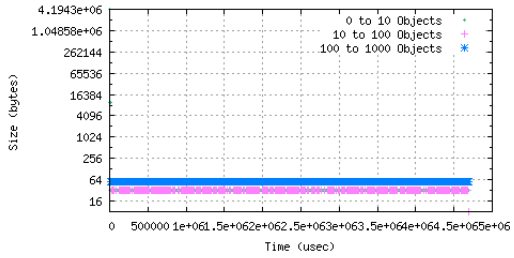
(e) XPDF



(f) XPDF Input 2

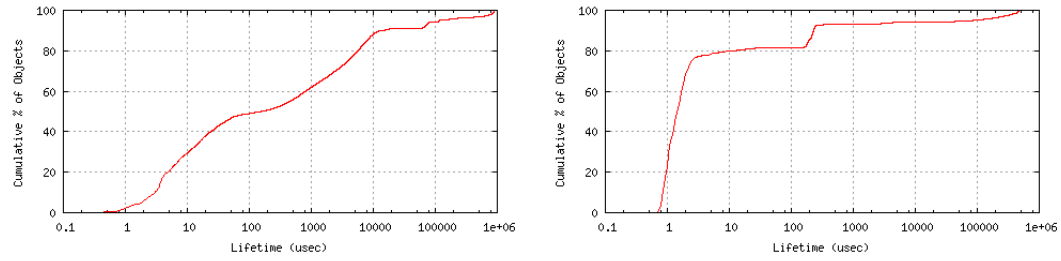


(g) ROBOOP



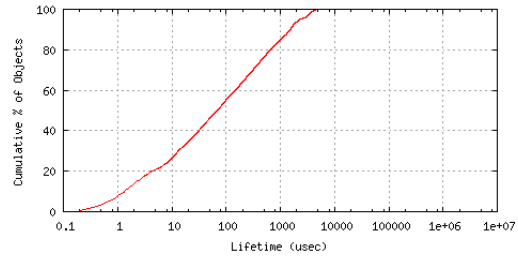
(h) Lindsay

Figure A.4: Bin Size Over Time 2

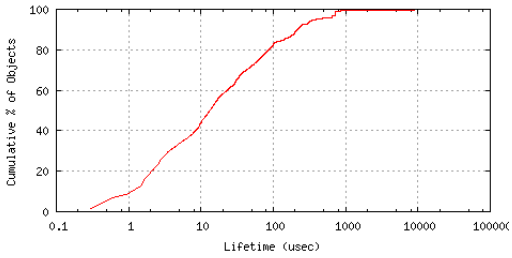


(a) P2C

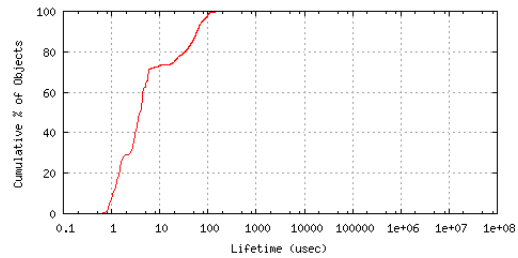
(b) GS



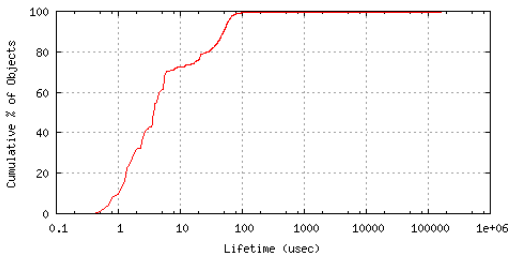
(c) Espresso



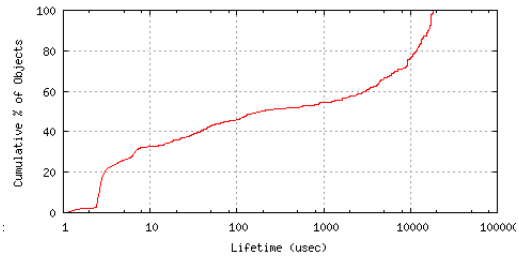
(d) Espresso Input 2



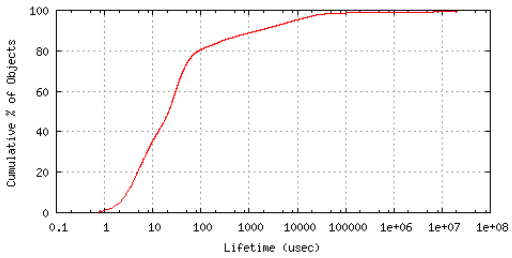
(e) CFRAC



(f) CFRAC Input 2

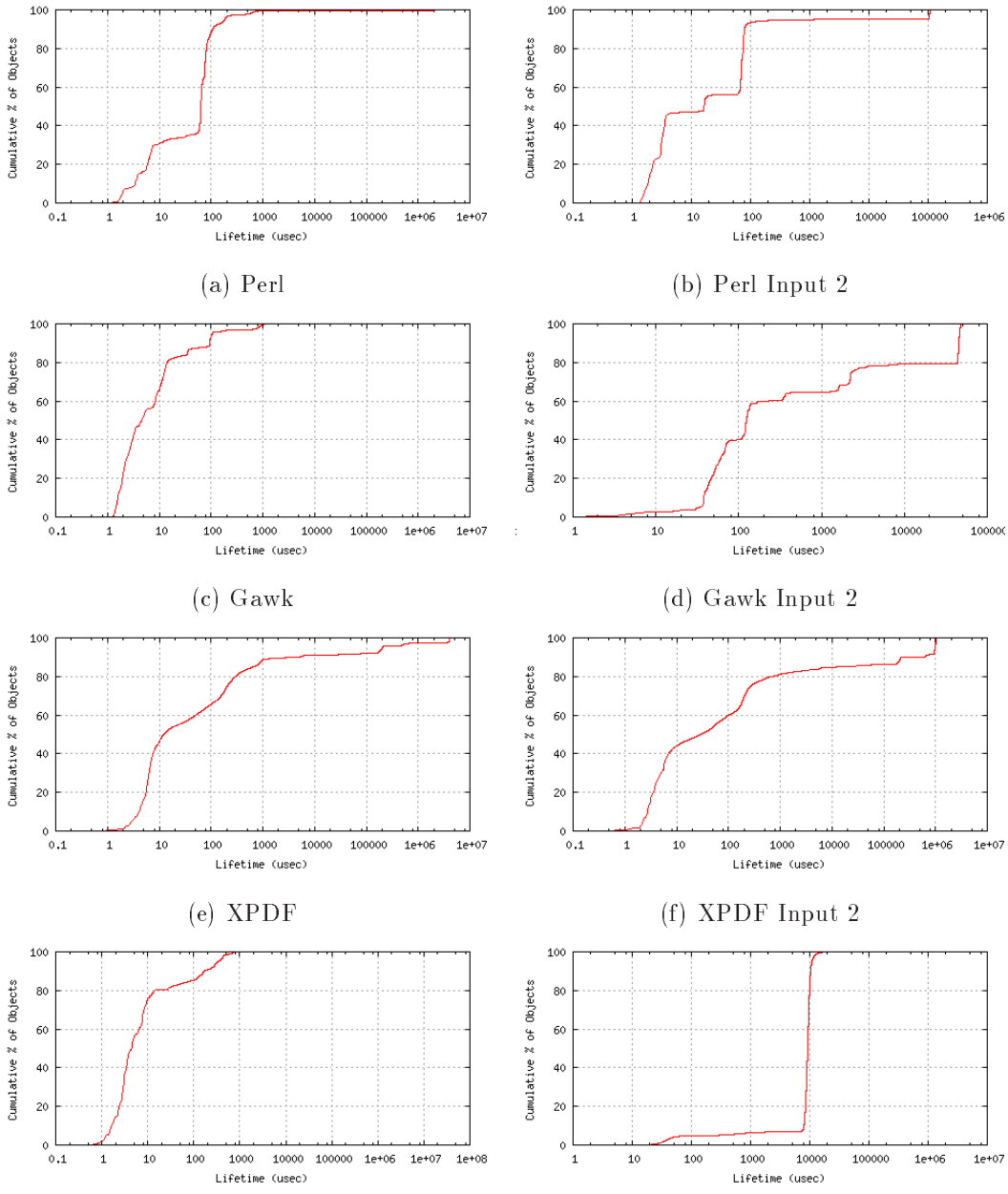


(g) GMake



(h) GCC

Figure A.5: Cumulative Lifetime Distribution



(a) Perl

(b) Perl Input 2

(c) Gawk

(d) Gawk Input 2

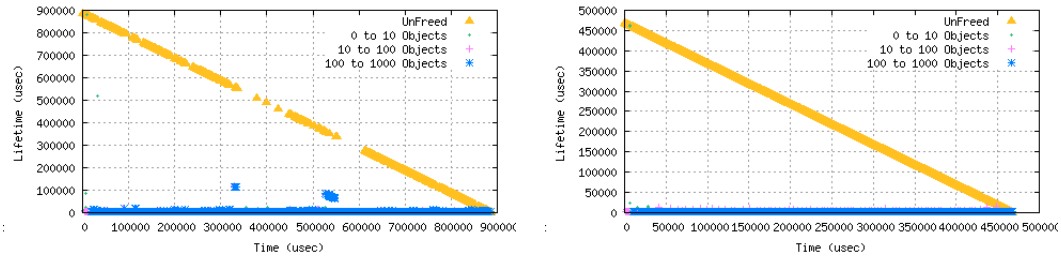
(e) XPDF

(f) XPDF Input 2

(g) ROBOOP

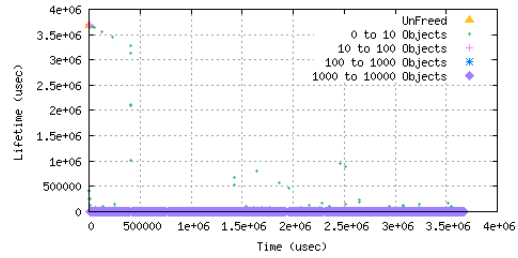
(h) Lindsay

Figure A.6: Cumulative Lifetime Distributions 2

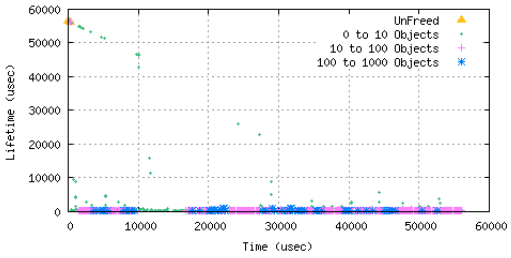


(a) P2C

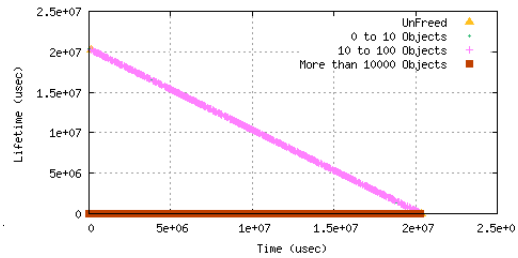
(b) GS



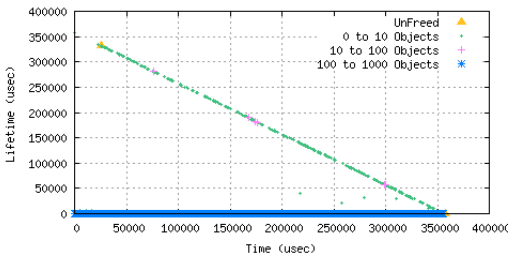
(c) Espresso



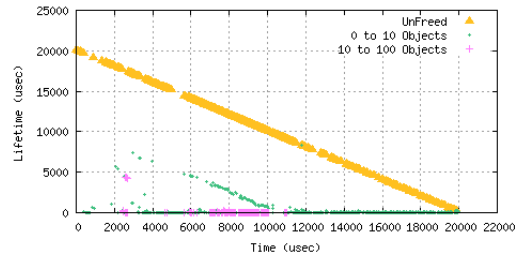
(d) Espresso Input 2



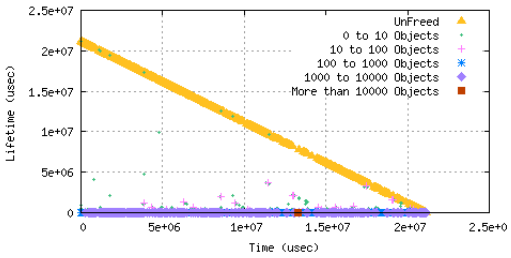
(e) CFRAC



(f) CFRAC Input 2

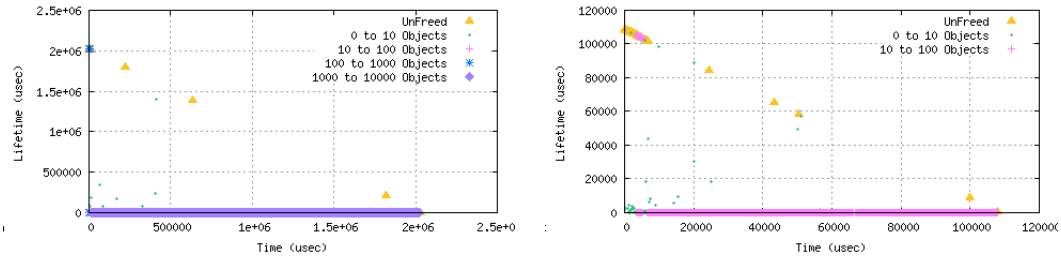


(g) GMake



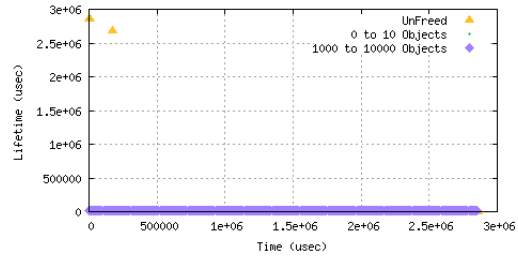
(h) GCC

Figure A.7: Lifetime Over Time

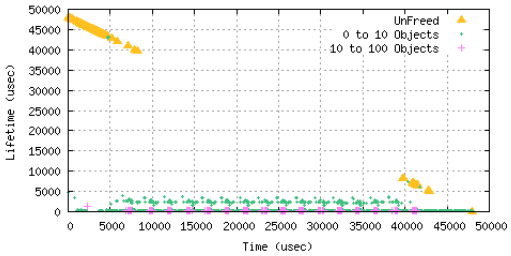


(a) Perl

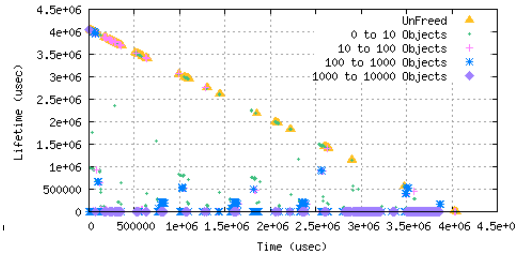
(b) Perl Input 2



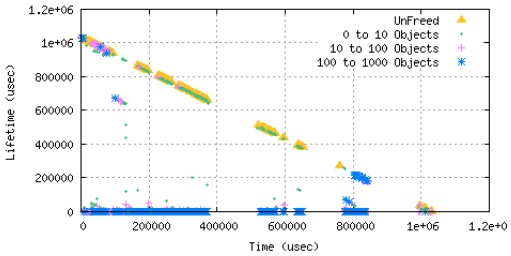
(c) Gawk



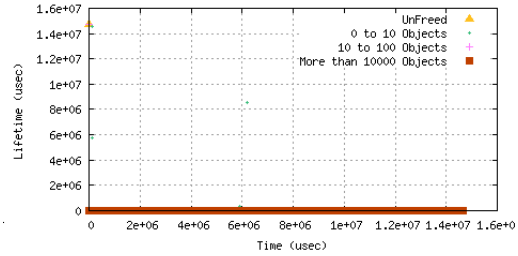
(d) Gawk Input 2



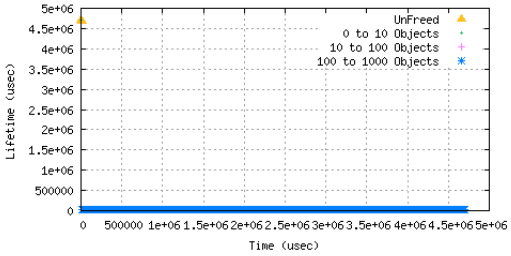
(e) XPDF



(f) XPDF Input 2

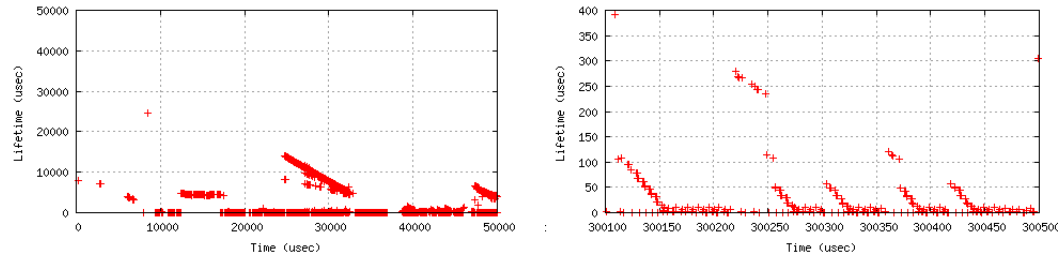


(g) ROBOOP



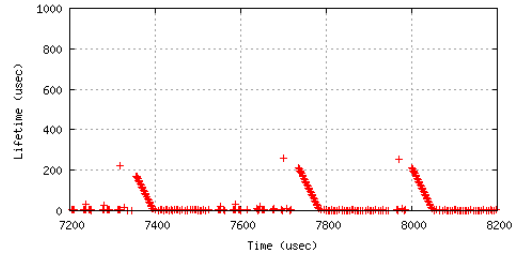
(h) Lindsay

Figure A.8: Lifetime Over Time 2

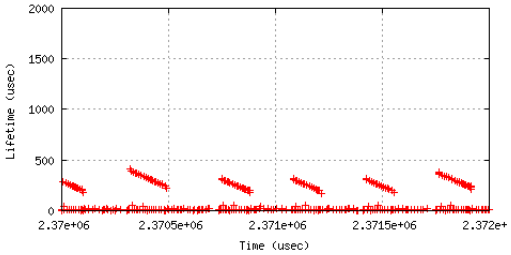


(a) P2C

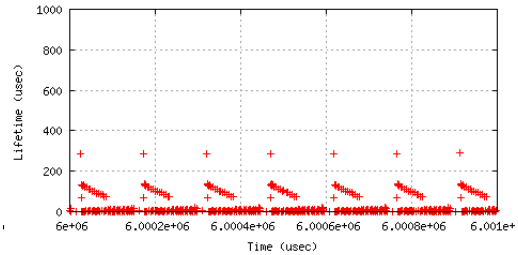
(b) Espresso Input 2



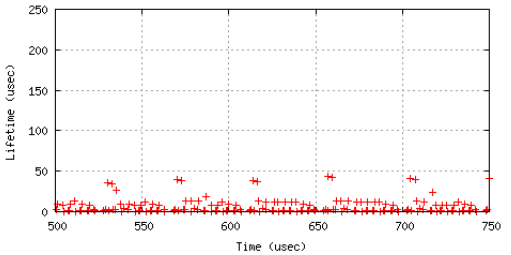
(c) GMake



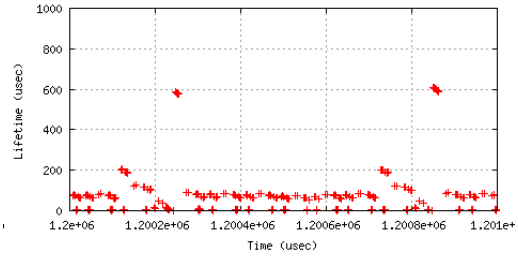
(d) XPDF



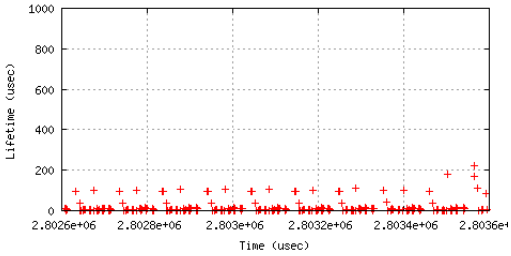
(e) ROBOOP



(f) CFRAC

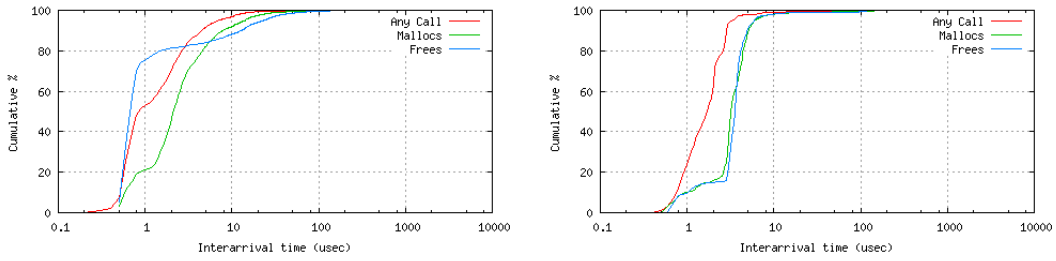


(g) Perl



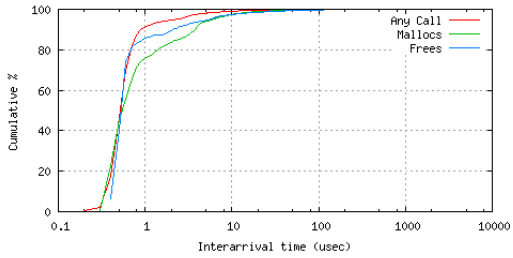
(h) Gawk

Figure A.9: Lifetime Over Time

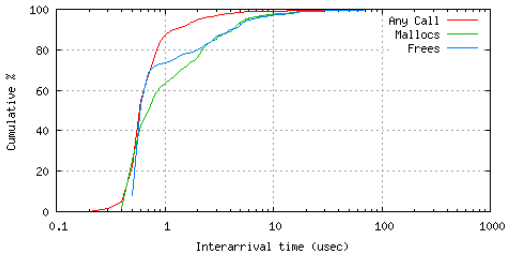


(a) P2C

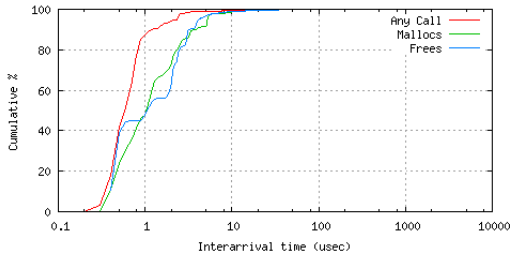
(b) GS



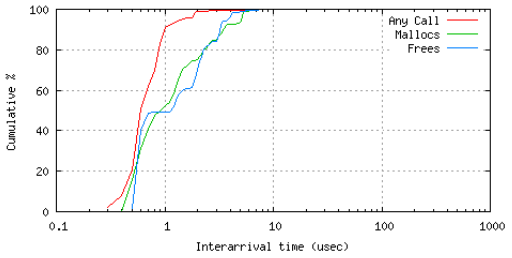
(c) Espresso



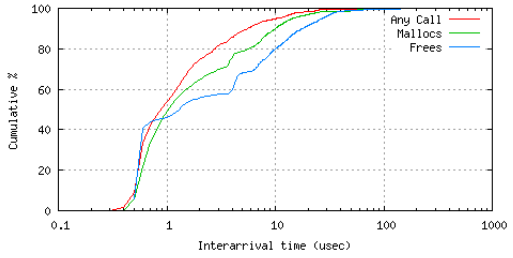
(d) Espresso Input 2



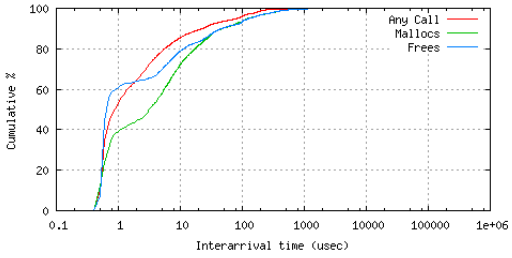
(e) CFRAC



(f) CFRAC Input 2

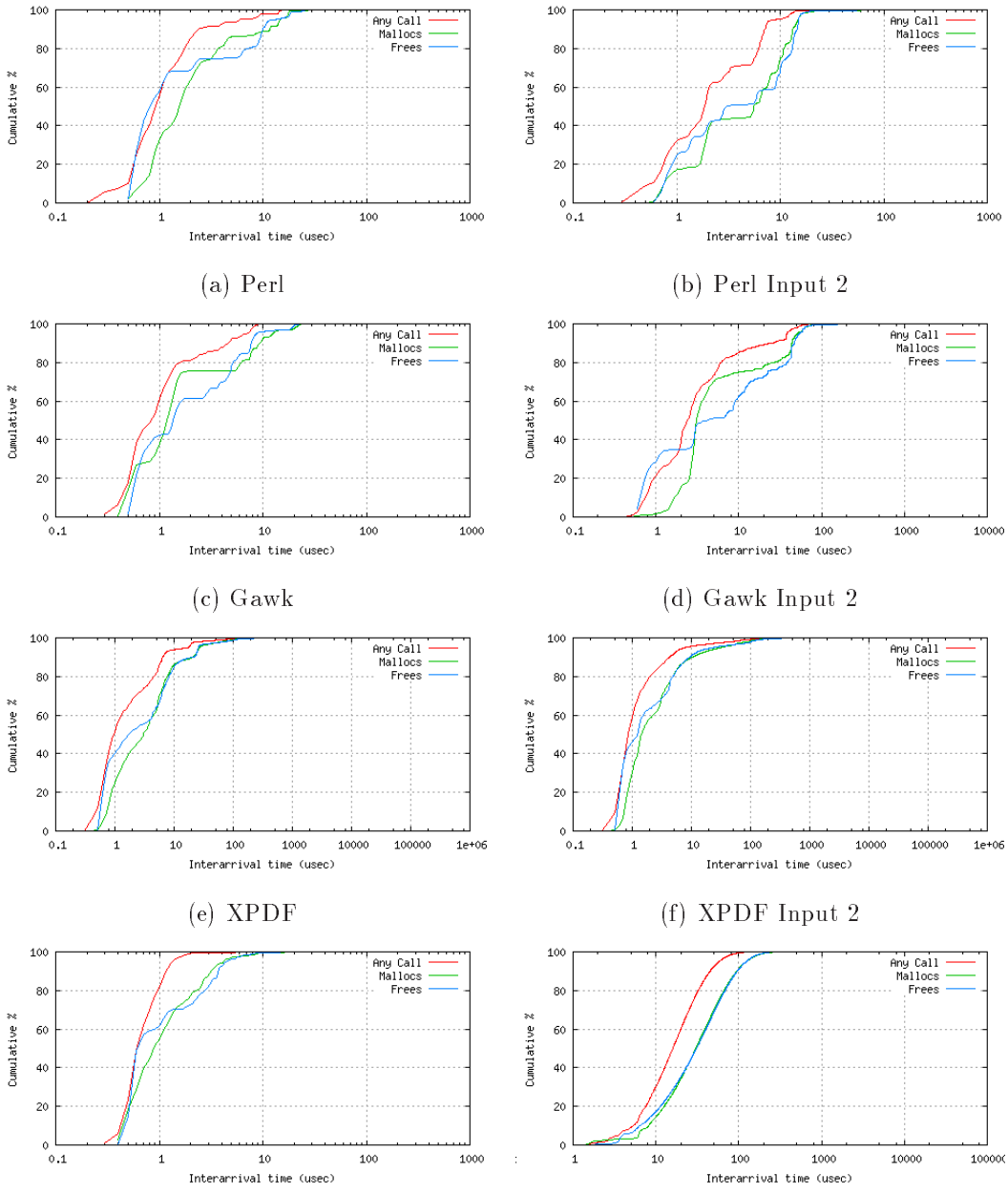


(g) GMake



(h) GCC

Figure A.10: Interarrival Times Cumulative Distribution



(a) Perl

(b) Perl Input 2

(c) Gawk

(d) Gawk Input 2

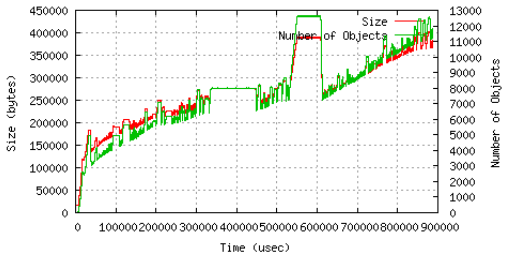
(e) XPDF

(f) XPDF Input 2

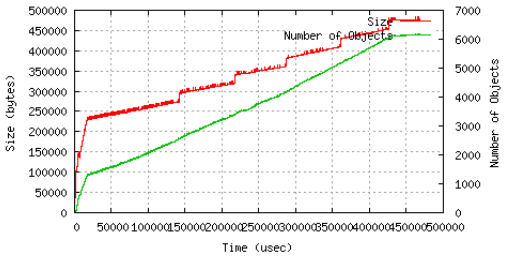
(g) ROBOOP

(h) Lindsay

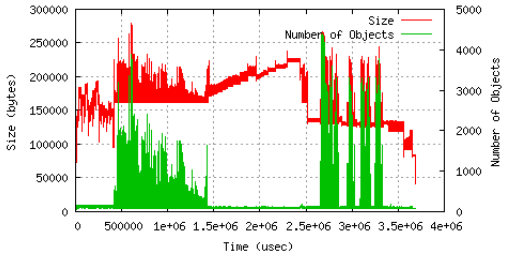
Figure A.11: Interarrival Times Cumulative Distribution 2



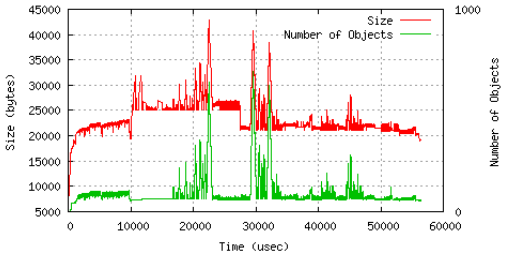
(a) P2C



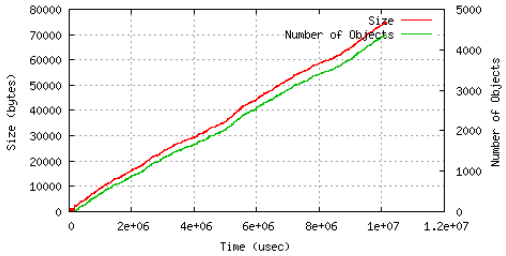
(b) GS



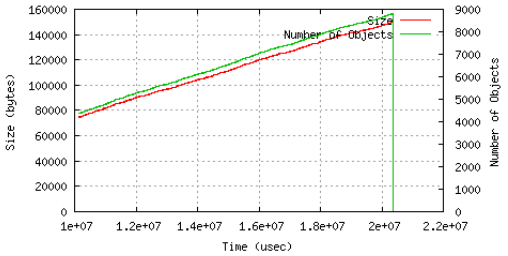
(c) Espresso



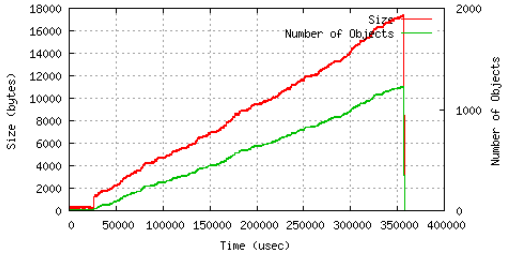
(d) Espresso Input 2



(e) CFRAC Part 1

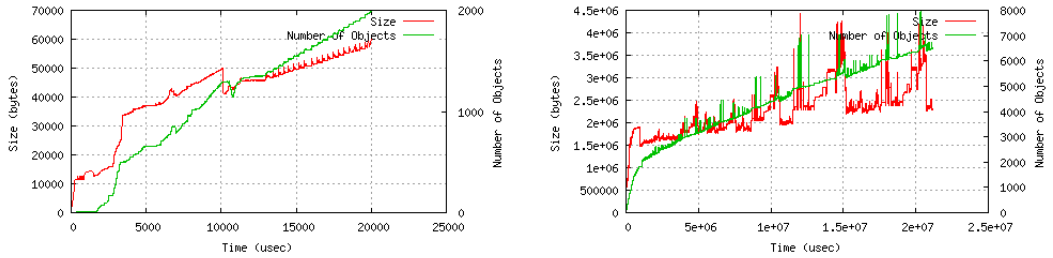


(f) CFRAC Part 2



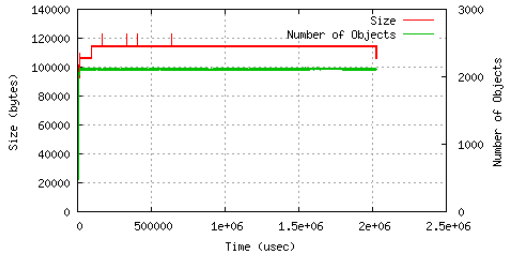
(g) CFRAC Input 2

Figure A.12: Allocation Footprint

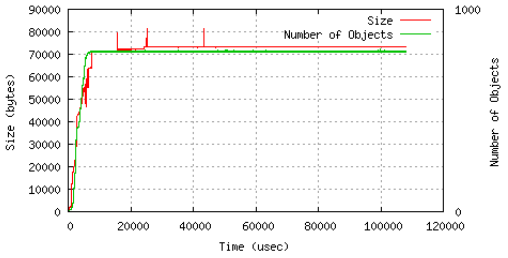


(a) GMake

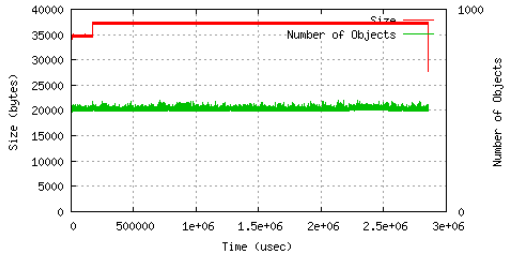
(b) GCC



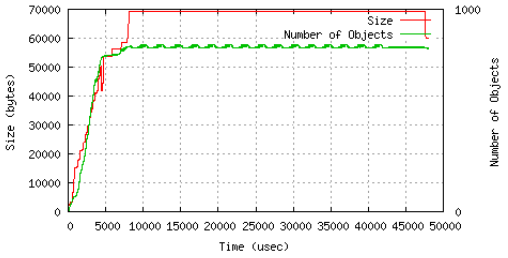
(c) Perl



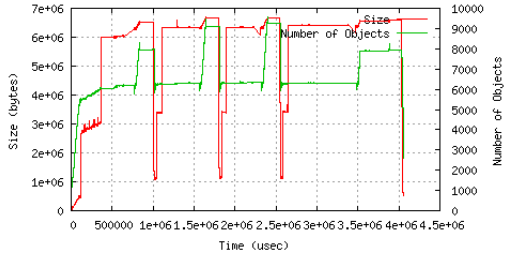
(d) Perl Input 2



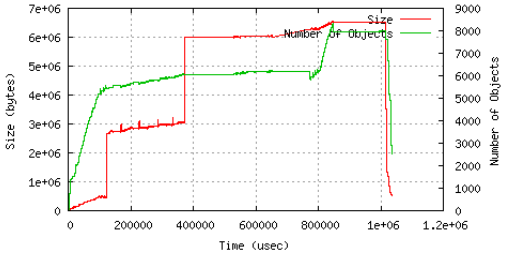
(e) Gawk



(f) Gawk - Input 2

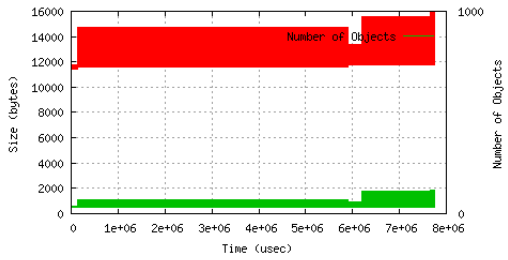


(g) XPDF

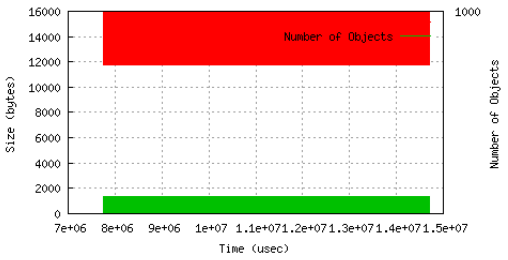


(h) XPDF - Input 2

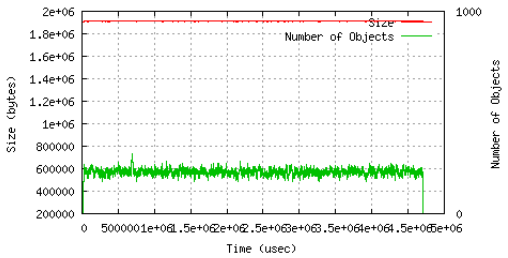
Figure A.13: Allocation Footprint 2



(a) ROBOOP Part 1



(b) ROBOOP Part 2



(c) Lindsay

Figure A.14: Allocation Footprint 3

Bibliography

[Ale01] Andrei Alexandrescu. volatile – multithreaded programmer's best friend. *Dr. Dobb's*, February 2001. 1

[AN03] Joseph Attardi and Neelakanth Nadgir. A comparison of memory allocators in multiprocessors. *Sun Developer Network*, 2003. 50

[Ber] Emery Berger. The hoard memory allocator. http://www.hoard.org/. 51

[BMBW00] 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. 1, 3, 9, 15, 51, 65, 70

[BZM01] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, 2001. 51, 54, 65

[BZM02] 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. 65

[DDZ93] David L. Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, 130 Lytton Avenue, Palo Alto, CA 94301 and Campus Box 430, Boulder, CO 80309, 1993. 65, 68, 69

[Den05] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005. 11, 12

[FB05] Yi Feng and Emery D. Berger. A Locality-Improving Dynamic Memory Allocator. In *Proceedings of the 2005 Workshop on Memory System Performance*, Chicago, Illinois, June 2005. 12, 24, 25

[Fer] Justin N. Ferguson. Understanding the heap by breaking it. https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf. 50

[Glo] Wolfram Gloger. Wolfram gloger's malloc homepage. http://www.malloc.de/en/. 50

[GM] Sanjay Ghemawat and Paul Menage. Tcmalloc : Thread–caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. 3

[GPT04] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating memory in a lock-free manner. Technical Report 2004-04, Computing Science, Chalmers University of Technology, 2004. 16

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 177–186, New York, NY, USA, 1993. ACM Press. 12

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993. 37, 38

[HLM06] Xianglong Huang, Brian T Lewis, and Kathryn S McKinley. Dynamic code management: improving whole program code locality in managed runtimes. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 133–143, New York, NY, USA, 2006. ACM Press. 2

[JW99] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36, 1999. 9, 10, 11, 101

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

[LK99] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. *ACM SIGPLAN Notices*, 34(3):176–185, 1999. 72

[LPB98] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 118–129, New York, NY, USA, 1998. ACM Press. 8

[MHM03] V. Luchangco M. Herlihy and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, may 2003. 37

[Nak01] Greg Nakhimovsky. Improving scalability of multithreaded dynamic memory allocation. *Dr. Dobb's*, 2001. 3, 49, 50

[Sal] Peter Jay Salzman. Memory layout and the stack. 1

[SAN] Scott Schneider, Christos Antonopoulos, and Dimitrios Nikolopoulos. Streamflow. http://people.cs.vt.edu/ scschnei/streamflow/. 52

[SAN06] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *International Symposium on Memory Management (ISSM'06)*, pages 84–94, June 2006. 3, 52, 70

[Sie00] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 9–17, New York, NY, USA, 2000. ACM Press. 8

[Wil] Paul R. Wilson. Locality of reference, patterns in program behavior, memory management, and memory hierarchies. 11

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland, UK, 1995. 8