Copyright

by

Michael Shannon Neely

1996

An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation

by

Michael Shannon Neely, B.A.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

May 1996

An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation

Approved by
Supervising Committee:

Acknowledgments

This thesis is part of a joint effort by the Object Oriented Programming Systems Research Group. Much credit is due to Mark Johnstone and his related study of memory allocation in garbage collection systems [9]. Credit also goes to David Boles for initial work on the heap simulator.

MICHAEL SHANNON NEELY

The University of Texas at Austin
May 1996

An Analysis of the Effects of Memory Allocation Policy on Storage Fragmentation

Michael Shannon Neely, M.A.
The University of Texas at Austin, 1996

Supervisor: Paul R. Wilson

The study of dynamic memory allocation has been dominated by measurement of performance of allocators with random input streams of requests. This study introduces a new methodology that separates the issue of policy from implementation and focuses on the effects of placement policy on fragmentation. It studies the effects of policy decisions such as object placement, splitting, and coalescing, on storage fragmentation. A useful and accurate measurement of fragmentation is presented that is based on the amount of waste at the point of peak memory usage. We have attempted to pick a representative set of allocators from the space of reasonable combinations of known policies. The allocators are used in memory allocation simulations to determine their respective fragmentation.

Our results show that the best fit (FIFO, LIFO or address-ordered) and address-ordered first fit policies yield the lowest fragmentation on average (14%), and that the overheads associated with these allocators are the largest source of wasted memory. We also explain how best fit can be implemented efficiently. A representative set of real program allocation traces is used in the simulations, and compared with randomized traces, to show that the application's patterns of allocation are an important factor in the allocator's performance and that studies based on synthetic traces are fundamentally flawed.

Contents

Chapter 1

Overview

Memory allocation is the problem of maintaining an application's heap space by keeping track of allocated and freed blocks. In this study, we are only interested in *non-moving* memory allocation, the kind that is used in most conventional computer systems. Allocators that can move objects are called *compacting* allocators because they can move allocated objects together to make more contiguous free space available. Compaction complicates pointer manipulation and makes applications slower due to the time overhead of copying blocks of memory, and the costs of tracking and updating pointers to moved blocks [23].

In particular, we are interested in allocation policies that use the application's heap space efficiently. Some allocators leave unusable holes in memory known as fragmentation. These holes are created when objects are allocated contiguously, but one object is freed while its neighbors are not. Fragmentation is just one way to measure an allocator's performance. Other measures are speed and locality of reference. This study focuses primarily on fragmentation, but speed and locality issues are also considered. There is also an important distinction between wasted memory due to fragmentation and waste due to allocator overheads.

1.1 The importance of Efficient Heap Management

Every year, more computers are sold, and demands for more memory are increased by applications and operating systems that use more system resources. By a rough estimate, there are many billions of dollars worth of computer memory in use today [25]. As the demand for memory increases, it is important that those resources are used efficiently. An allocator that reduces fragmentation generally allows applications to run with less memory and lowers the overall memory requirements of a computer system. Programming systems that use non-moving garbage collection can also benefit from better memory allocation.

1.2 Methodological Problems with Previous Studies

Previous studies of memory allocation have relied too heavily on random streams of allocation requests to measure the performance of an allocator. Random streams of requests distort the measure of fragmentation and yield misleading or erroneous results. Patterns of allocation in real applications generally do not resemble smooth distributions. In this study we have collected memory request traces from a set of real applications in order to more accurately assess the performance of an allocator.

Another problem with previous studies is that they often confuse the issues of allocation policy with the mechanism for implementation. The speed of an allocator is mostly a property of its implementation. Many studies are too concerned with speed without understanding the algorithmic issues of allocation. Without a theory of allocation policies, the study of allocation is reduced to trial and error. We have attempted to focus on the policy issues in this study, while supporting good policy choices with suggestions for efficient implementation.

1.3 Our Experiments

In order to study the performance of various allocation policies with real application trace data, we built an experimental framework for trace-driven simulation and analysis. This framework (described in detail in Chapter 5) allows us to accurately compare allocation policies with all the applications in our test suite. The tools are portable and adaptable to wide range of systems and platforms. Our application trace data and simulation and analysis tools will be made available for public access.

1.3.1 Effects of Policy Variation with Real Program Behavior

In our experiments, we tested various basic allocation policies together with some orthogonal features to find out which, if any, of the features improve the performance of the basic allocators. Some interesting results are that some of the basic allocators (e.g., best fit or first fit) have very low fragmentation, far lower than was previously believed. Fragmentation in these allocators has been overestimated because there was no distinction between fragmentation and allocator *overhead*—space used for bookkeeping by the allocator itself.

Our results show that the overheads associated with most common implementations account for a large part, if not most[1], of the wasted memory. The features we tested improve fragmentation for some of the basic allocators whose performance was lacking (e.g., next fit), but have little or no effect, in terms of fragmentation, on the best allocators. This is still a useful result since some of the features have other beneficial effects that can be taken advantage of without increasing fragmentation.

1.3.2 Biasing Effects of Traditional Simulation Methodology

We wrote a tool to randomize the order of requests in an allocation trace. The distributions of object sizes and lifetimes are preserved; only the order is changed.

[1] The remaining wasted memory is due to actual fragmentation.

We used this tool to create a set of randomized traces from our application traces. The results show that in some cases, randomization inflates fragmentation results by several times. Randomization also seems to favor some allocators and penalize others. The simple segregated store allocator performs better on the randomized traces than the original because the randomization eliminates patterns that cause more fragmentation with that allocator. Other allocators perform worse with randomized traces because the patterns of allocation that they take advantage of are destroyed.

1.3.3 A Sound Basis for Future Work

We have attempted to create a sound model for theoretical and empirical study of memory allocation. An important part of this effort is the separation of the strategy, policy, and mechanism associated with an allocator. It is not well understood how various policy choices interact and combine to affect fragmentation. Our model allows us to abstract away the details that are not relevant to a particular experiment. The simulation and analysis tools will enable more research into allocation studies involving locality and speed. Our results so far give us a good picture of which allocators are likely to work well in practice. Further results on the impact of allocator overhead on fragmentation will be published in a forthcoming dissertation by Mark Johnstone [9].

1.4 Structure of this Thesis

This thesis is organized in a top-down fashion. Some important results are introduced at the beginning along with a discussion of the more abstract issues associated with the results. This is followed my more detail in later sections. Chapter 2 introduces some background concepts in memory management. Chapter 3 is an overview of allocators, organized by mechanisms. Chapter 4 presents the experimental design, including details about the application suite used for the simulations and the allocation features that were tested. Chapter 5 documents the simulation and profiling tools that were used in the experiments, as well as the kinds of data that were collected. Chapter 6 presents the results, organized by allocation features. An analysis of the statistical significance of the results is also included. Chapter 7 is the conclusions and future work. The complete results of the experiments are listed in the appendix.

Chapter 2

Introduction

Computer programs that use dynamic memory allocation depend on the operating system and higher-level memory management routines. The operating system typically provides a mechanism for making relatively large blocks of memory (e.g., 4KB or more) available to the program, e.g., the `sbrk()` system call in UNIX. It is the function of the allocator routines which lie between the operating system and the user application to decide how to allocate the application's memory requests within the memory obtained from the operating system.[1] The decision of where to place objects is largely an educated guess. Once an object is allocated, it cannot be moved and there is no way to know when the application may free it. Free blocks that cannot be used because they are too small to satisfy a request (and cannot be merged with its neighbors) causes wasted memory known as fragmentation. Bad placement choices by an allocator can lead to excessive fragmentation, and severe fragmentation is always possible if the allocator's placement policy interacts badly with an application's allocation and deallocation behavior.

Programs written in conventional programming languages generally rely on non-moving memory allocation. Programming paradigms such as object-oriented

[1] C and C++ programs typically rely on the library routines malloc() and free() to allocate and free memory.

programming and functional programming rely heavily on dynamic memory allocation. Some kinds of garbage collection also depend on good non-moving memory allocation [24]. For these reasons, it is important that the allocator function be efficient in its use of memory.

The performance of an allocator depends crucially on the input stream of requests (i.e. the sequence of allocations and frees requested by the application). Due to the unpredictability of allocation requests, it is difficult to create a useful mathematical model with which to study it. The simplifying assumptions used in most earlier work have been shown to be invalid and systematically biased (see [25] and section 6.4 of this thesis). Therefore we have used an experimental methodology to analyze the behavior of allocators under real workloads. We compared the results of these experiments with those using synthetic traces, and found that allocator performance on synthetic traces is not a good predictor of the performance on real workloads.

2.1 What an Allocator Does

An allocator must manage an application's heap space by keeping a pool of free blocks with which to satisfy the application's allocation requests. The allocator has no information about when the blocks will be freed after they are allocated. The order of these requests is entirely up to the application. The decision to be made by the allocator is where to place the requested block in the heap. A conventional allocator cannot *compact* memory by moving blocks after they are allocated. Once the decision is made, it cannot be revoked; the block will remain in its place until it is freed by the application. If the blocks are placed in such a way that later freeing creates "holes" that cannot be used to satisfy any requests, then memory is wasted. The effect of these unusable holes is known as *fragmentation*, because free memory is fragmented into unusable blocks. It is the goal of an allocator to reduce fragmentation by making the right placement choices. It is also desirable for the allocator to accomplish this with as little extra space and time as possible.

Since the order of applications requests cannot be known in advance, the allocator must make some assumptions about the request stream. Most applications exhibit some regularities in their allocation, such as allocating many requests for blocks of the same size, or freeing many blocks at the end of a phase. These regularities can be exploited by the allocator to help in the choice of placement of blocks[25]. The allocator may also be able to *split* and *coalesce* free blocks in order to maintain the availability of large blocks of memory. Splitting is breaking up a larger free block into smaller free blocks and coalescing is merging two smaller free blocks into a larger one.

2.2 Costs of Memory Allocation

Costs associated with an allocator include fragmentation, space and time overheads, and locality effects. This study focuses on the space costs. We will show that best fit and address-ordered first fit allocation policies yield very low fragmentation with most workloads. Special features such as deferred coalescing, clustering, wilderness preservation (saving the most recently allocated sbrk region for last), and splitting thresholds can be used to fine-tune an allocator. The interaction between these features has not been well understood. We will study these interactions with a variety of analysis tools. Some important implications for locality effects will be suggested, but are not within the scope of the study.

From the viewpoint of the applications program, the purpose of the allocation function is to return a pointer to a contiguous block of memory of the size specified in the call. The memory may be physical memory, or virtual memory, paged from a disk.

In the case of physical memory, the allocator may be unable to satisfy the request because there does not exist a block of memory of the specified size. Even if there is sufficient free memory in the heap to satisfy the request, this condition may occur because there is not enough *contiguous* free memory. It is up to the application program to deal with such failure in an appropriate manner. In many cases, a graceful halt may be the only possible action. This condition can be avoided in some cases by better memory allocation.

If virtual memory is used, and the heap size can be reduced by better allocation, then it is likely that the program will require fewer page swaps from virtual memory, thus improving the performance of the application (see 2.7).

2.3 Policy vs. Mechanism

An important aspect of this study is the separation between allocation policy and allocator implementation. Policy determines the decisions by which an object's placement in the heap is chosen. Mechanisms that implement a particular policy consist of algorithms and associated data structures. For any particular policy, there may be a variety of choices of implementation mechanisms. We have focused attention on policy in the belief that implementation is largely an independent issue. Past studies have dismissed some allocation policies simply because the best-known implementation is inefficient; they have generally overlooked straightforward algorithmic improvements [25].[2] Thus we have attempted to avoid any biasing of our results due to implementation considerations that we consider relatively minor. Suggestions for efficient implementation will be given for each of the allocator variations later.

[2] For instance, best fit can be implemented with a self balancing binary search tree or splay tree for the free list to speed up searches. A segregated fits allocator also approaches the performance of best fit as the size class ranges are made smaller.

2.4 Real vs. Synthetic Traces

Another notable aspect of this study is that we have relied exclusively on memory allocation traces from real applications, rather than randomly generated requests streams. Traditionally, synthetic traces have been used, statistically preserving size and lifetime distributions believed to represent real programs. (In some cases, these statistics were obtained from real programs, but the ordering of the requests was random).

By comparing results of simulations using real memory traces with simulations using random traces, we are able to show conclusively that an allocator's performance in an application is not accurately reflected by its performance on a random trace. The correlation between fragmentation of the original traces and fragmentation of the randomized traces is only 0.5. This means that *most* of the fragmentation related behavior from the original trace is missing from the randomized trace.

Since the earliest studies of dynamic memory allocation, it was known that studies using random request streams were not methodologically sound, but these warnings were largely ignored [25]. Some studies using real workloads were published, but often the data and allocator details were not available to verify the results. In many cases, the measurement of performance of allocators was not well defined.

2.5 The Test Suite

The applications we used for allocation traces were selected by several important criteria. We attempted to choose a cross-section of applications, not depending too heavily on any one kind of application for our results. Also, they were chosen from publicly accessible sources, so that the experiments can be reproduced by others.

The allocation behavior of the applications displayed a variety of patterns and have a large enough footprint to test all the features of an allocator. More application trace data is needed for a complete study, but we have more data than previous studies.

2.6 A Simple Example Allocator—Best Fit using a Sequential Fits Implementation with Boundary Tags

To make the discussion of allocation issues more concrete, we will introduce a simple example allocator and its implementation. *Best fit* is an allocation policy that has typically been implemented by a *sequential fits* mechanism. Other common allocators that use the sequential fits mechanism are *first fit* and *next fit*. The sequential fits mechanism uses a linear list of free blocks. The allocators typically use Knuth's boundary tag technique (described later) and a doubly linked list to make coalescing simple and fast.

Our simple best fit allocator searches the free list to find the smallest free block large enough to satisfy a request. In the general case, a best fit search is exhaustive, although it may stop when a perfect fit is found. This exhaustive search means that a sequential best fit search does not scale well to large heaps with many free blocks. (Better implementations of the best fit policy therefore generally use indexed fits or segregated fits mechanisms, described later.)

The policy and mechanism of our simple best fit allocator is clear, but the strategy(see section 2.9 for definitions) is not well understood. Most allocators do not have a clearly defined strategy, and the distinction between strategy and policy are not clear. In the case of best fit, one possibility is that the strategy is to minimize the amount of wasted space by ensuring that fragments from split objects are as small as possible. A potential problem with this strategy is that there may be an accumulation of small fragments.[3] Another possibility might be to avoid splitting up larger blocks in order to leave them available for subsequent requests. Both of these strategies seem to be supported by best fit, based on the study of its behavior. However, the density plots for best fit show a pattern of large free blocks being split up into contiguous chunks, and then freed up to return another large free block, which suggests the latter explanation. Best fit's low fragmentation also supports this.

We will show that best fit is one of the allocators in our tests that produces very low fragmentation on average (14%). However, an analytical analysis reveals that this level of performance is not guaranteed. In the worst case, its memory usage is proportional to the product of the amount of allocated data, M, and the ratio between the largest and smallest object size, n (i.e., Mn) [6, 18]. This appears not to happen in practice, or at least not commonly. In comparison, address-ordered first fit has similar fragmentation and much better worst case.

2.7 Real vs. Virtual Memory

In order to understand space costs of memory allocation, we must distinguish between real and virtual memory systems. A good measure of fragmentation should be applicable to either case. In the case of real memory, an application simply can not run if there is not enough memory. With virtual memory, however, the issue is more subtle. Since disk access is so much slower than memory access, paging should be avoided if possible. In general, lower fragmentation will decrease the likelihood of paging, all other things being equal.

There are cases where there is simply not enough memory available for an application. The only solution is to increase the memory of the system. In a system with virtual memory, however, memory can be swapped from secondary storage

[3] This phenomenon is known as "splinters" or "sawdust'.' It was noted by Knuth [10], but seems not to be a serious problem with either real or synthetic workloads.

to free up pages of memory for another process (this is typically achieved using an approximate least-recently-used paging algorithm). Still, if there is not enough swap space, the virtual memory may be insufficient to support the application. It is more often the case with modern computer systems that the swap space is sufficient for most applications, but excessive paging, often called *thrashing*, can lead to drastic decreases in performance. Even with minimal paging, performance is affected, since disk access is several orders of magnitude slower than memory access. If the allocator can be improved so that the application's *footprint*, or the size of its heap, is smaller, then paging is likely to be reduced and performance increased. Similarly, even if the application does not need to page, a smaller footprint can increase overall system performance by providing more available memory for other processes.

In many applications, paging may cause unacceptable delays. In these cases, it is crucial to avoid paging. These applications have a point at which paging starts and the performance drops drastically. An example is a heavily-accessed set of large data structures that do not fit in memory at once. If the locality is not good, such as with hash tables, or structures accessed in a loop, the application will need to bring the same pages in and out many times. If the application's footprint can be reduced by better memory allocation, then it is less likely that paging will occur. The total memory requirements for such a system can also be reduced by better memory allocation.

2.8 Measure of Fragmentation

In studying memory allocation, an essential component is the metric by which allocators are compared. The two most commonly used measurements are speed and fragmentation. Fragmentation can be defined in many different ways. A common definition of fragmentation is the amount of wasted memory at a "steady state." Our study of allocation traces shows that many programs do not exhibit any such steady state behavior, so this metric is unrealistic. Another problem with this definition is that it does not distinguish between unused memory and memory that is used for allocator *overhead* (e.g., space required for bookkeeping data structures used by the allocator).

In our study, we will consider *actual* fragmentation to be the amount of wasted memory, not including allocator overhead, relative to the peak memory usage. The point of this is to separate the effects of mechanism from the effects of policy. The measurement is useful in determining the overall memory requirements of a program. It is also a simple measure that is independent of locality. Locality effects are important, but beyond the scope of this study; total memory requirements are often equally important and sometimes more important.

The amount of fragmentation is calculated as a percentage of the maximum number of live (allocated) bytes.[4] That is, $F = \frac{MaxHeapSize - MaxLiveBytes}{MaxLiveBytes}$. One complication is that the point of maximum live bytes and the point of maximum heap size may coincide. If fragmentation is higher at some point other than the point of maximum live bytes, then the heap size could be inflated. Our experiments revealed that in most cases the two points occur close together and should not distort the comparison of fragmentation between applications and allocators to any significant degree.

The peak memory usage metric is useful in a variety of situations. For applications that use no virtual memory, this metric is a "high water mark" that indicates the most memory that the program will need at any point in its execution.

If there are multiple applications running simultaneously, the system may require the sum of the peak memory usages of each application, in the case that they all reach their peak memory usage concurrently. Conversely, an application may reach its peak alone, when other applications are using little memory. Unfortunately,

[4] Here, maximum live bytes excludes allocator overhead. See section 6.6 for discussion of the treatment of allocator overhead.

in a modern workstation environment, this complementary resource usage doesn't actually happen as often as one might like. If paging is required, then it is likely that a smaller footprint will reduce the amount of paging.

A good allocator can also affect the locality of reference in an application, so that frequently referenced objects are stored close together in memory. Such locality effects will be pursued in future work.

In computing tasks where speed is a high priority, allocator functions can be optimized to provide fast service. An allocator function can also be optimized to reduce fragmentation. There can be a tradeoff between the speed and fragmentation associated with an allocator, since the processing required to manage the heap more efficiently (in terms of space) may cause it to take more time. However, we believe this tradeoff is often overemphasized, and that allocators with excellent memory usage can also be quite fast [25]. Similarly, speed optimizations may affect policy. In this study, we have chosen to use fragmentation as the primary measure of allocator performance, but we have included effects of some important speed-oriented optimizations in our results.

The speed of an allocator is determined more by its implementation than by its policy. A good allocation policy may suffer from a poorly chosen implementation. Some allocation policies that yield low fragmentation have not been given the consideration they deserve because the typical implementation is too slow. The best general allocation policies are best fit and address-ordered first fit (and variations), both of which can be implemented efficiently, despite the fact that the straightforward implementations do not scale well with the number of free blocks [25].

Other issues associated with an allocator function are locality of reference and amount of overhead. We have included some important policy variations that can be expected to affect locality, and have experiments that simulate the memory costs of different implementations. Locality of reference affects an application in the hit/miss ratio of the cache and the frequency of page swapping from virtual memory. The space overhead of an allocator can cost as much or more, in terms of wasted memory, than fragmentation.[5]

In many studies, allocator overhead has been mistaken for fragmentation. The distinction is that overhead is an artifact of the allocator and is not a part of application's data. Other kinds of overhead are free list maintenance overhead, alignment requirements, and minimum block sizes. We will assume that any good implementation of an allocator uses at most 4 bytes of overhead per object.[6] The minimum block size is usually a function of the bookkeeping data required for free blocks.

2.9 Allocation Policy Issues

The study of fragmentation requires a careful analysis of the causes of memory waste. Traditionally, fragmentation has been classified as *internal* or *external* [17]. Internal fragmentation is wasted memory due to an allocator's returning a block that is larger than the requested size. External fragmentation is free memory that cannot be used because it is not the right size to satisfy any requests, and it cannot be merged with any neighboring blocks to make it larger. This is caused when a block is freed, but its neighbors are still allocated. An allocator may choose to allow some internal fragmentation in an attempt to decrease external fragmentation or to support novel coalescing schemes, as in a binary buddies allocator.

A good allocation strategy would be to minimize external fragmentation by placing together objects that are freed at the same (or nearly the same) time. If the freed blocks are contiguous, a larger chunk of memory is available than would be if the freed blocks were separated. Although this is not possible in general due to the

[5] Our most recent results show that allocator overhead may be the most significant source of memory waste in many allocators.

[6] The traditional boundary tag technique can be optimized to use only a header rather than a header and a footer for each object, as described in a later section.

unpredictability of the request stream, it may be approximated by a heuristic.

The design of an allocator can be broken into three parts—strategy, policy, and mechanism [25]. An allocator algorithm can be considered as the *mechanism* that implements a *policy*, which is motivated by a *strategy* for minimizing fragmentation. The strategy must take into account patterns of allocation in the request stream. Since the actual input stream is unpredictable, the allocator must make some assumptions about program behavior in general. Regularities in the request stream can be exploited to predict the behavior of future requests. The policy of an allocator must decide how to place objects so as to satisfy the strategy. The mechanism is the specific implementation of a policy by means of algorithms and data structures. A mechanism must be considered carefully because changes in small details can lead to a different policy.

Some important policy issues are :

- *Fits.* What is the criterion used to select a free block from the heap to satisfy a request? It could be size, locality, or a combination of criteria.

- *Splitting and Coalescing.* When a block is selected to satisfy a request, what is done with the excess portion of the block, if the block is larger than the requested size? The block could be split, and the remainder returned to the free list. Or the entire block could be allocated. If there are adjacent free blocks in the heap, they can be coalesced to create larger free blocks. They can be coalesced immediately after a block is freed, or deferred until it is necessary to satisfy a request. Deferred coalescing can reduce unnecessary splitting and coalescing when small objects are reused often.

- *Patterns of Memory Reuse.* When an object is freed, when will it be reused for another request? An object might be reused at the very next request, or saved for a request of a certain size or other criteria. The object may also be given a priority depending on its location in memory (e.g., objects at one end of the heap may have higher priority than those at the other end).

- *Splitting thresholds.* When is a larger free block split up to satisfy a request? If the remainder is always split off, this could lead to increased external fragmentation. If free blocks are never split, internal fragmentation could increase. A threshold can be used to determine when to split a free block, depending on the its size relative to the requested size, in order to balance internal and external fragmentation.

2.10 Allocation Patterns in Real Applications

By examining the memory allocation behavior of real application programs, we were able to determine which kinds of patterns allocators are able to take advantage of to reduce the overall fragmentation [25]. These patterns are caused by loops, sequential access of arrays, I/O routines, and other common operations in structured programs. Nested phase behavior creates patterns of memory usage that makes efficient allocation easier. Often, many objects die at the end of a phase, making a large free block available for the next phase. Therefore, we find patterns of linearly increasing memory usage, followed by rapid deallocation, or *ramps*. Some applications also have periods of little or no net allocation (only short-lived objects are allocated) known as *plateaus*. In some cases, these patterns appear at a number of different levels of resolution, suggesting nested structures and/or recursion.

Allocation request sizes also tend to fall into a relatively small set of sizes that correspond to repeated allocation of common data structures. Exceptions to these patterns occur in programs that are heavily dependent on input such as `gcc` and `espresso`. In these cases, *spikes* in allocation may appear as a result of varying input.

For a more detailed description of common allocation behaviors of applications, see [25].

2.11 Related Work

Since we have shown the traditional synthetic trace methodology to be unsound, the discussion of related work in this report will be limited to studies using real traces or contributing to a sounder methodology. See [25] for a more general survey of allocation studies.

Some early work that supports allocation studies was published by Totschek [21], Minker [14], and Batson, Ju, and Wood [4], and Batson and Brundage [3]. These studies provided empirical data about memory usage in specific computer systems. These results were used in many subsequent studies. Unfortunately, the data were often used to generate pseudo-random traces with the same approximate distributions, but the effects of smoothing and random ordering make this an unsound methodology. Totschek collected statistics on memory usage of processes in the SDC (Systems Development Corporation) time sharing system. Minker also provided histograms of object sizes in the University of Maryland UNIVAC Exec system. Batson, Ju, and Wood showed segment size and lifetime distributions in the University of Virginia B5500 system. Batson and Brundage provided similar information for a varied collection of 34 Algol-60 programs.

One of the earliest studies to use a sound empirical methodology was Margolin's study of memory allocation in the control program of the IBM System/360 mainframe [13]. This study was concerned with memory used by the operating system, not by applications. They found that many of assumptions underlying other studies were false. Size and lifetime distributions of objects were correlated; successive requests were not independent; and, the distributions were not well-behaved as in an exponential or uniform distribution. Unfortunately, these warnings were not paid much attention; many studies continued to use synthetic traces. Their focus was on speed of the allocator. They considered address-ordered first fit and used deferred coalescing to attempt to optimize the allocator for speed.

Later, Leverett and Hibbard [12] used a set of 5 programs written in Algol-68 to experiment with deferred coalescing and different free list orders. Unfortunately, the programs were all simple textbook-style examples and were less than 100 lines of code; this may have skewed the results since the programs are not representative.

Zorn, Grunwald, Barrett, and Henderson have published a number of studies using sound methodology with real application trace data [1], [2], [8] [7]. These studies focus primarily on the speed of allocation, rather than fragmentation. They also studied locality issues and garbage collection. In one study, they made an effort to create synthetic traces based on real distribution data, that support useful allocation experiments. They concluded that no model of allocation based on synthetic traces is as reliable as trace-driven simulation, in terms of error introduced by the model [26].

In general, the studies by Zorn, Grunwald *et al.* focus on bottom-line space and time performance for fairly complex allocators, including hybrids. The do not clearly separate out basic issues of policy from effects of implementation details, as we do here.

Recently, Vo has studied a variety of allocators in the context of his `vmalloc()` allocator framework, which allows the use of multiple underlying allocators for data with different expected characteristics [22]. Like Zorn, Grunwald, *et al.*, Vo does not clearly separate out issues of policy and mechanism. His focus is on allowing the programmer to control which allocator is used for which data, to adjust various tradeoffs. In contrast, our work is more oriented toward basic understanding of policy, because we believe a single good policy can offer robustly good memory usage without requiring programmer-controlled strategy choices in most cases. (We also believe that once such a policy is identified, an appropriate mechanisms can be found to keep overheads and time costs low.) We would prefer to avoid the need the use of a `vmalloc()`-like framework for most purposes, because it can incur external fragmentation—it is difficult to share freed memory between multiple allocators.

We would like to give credit to Ben Zorn and Dirk Grunwald, et al. for making their test applications available, and to Phong Vo for comments that have improved our understanding and presentation.

Chapter 3

An Overview of Allocator Mechanisms

A wide variety of allocators have been implemented and used in computer systems. Here we will present an overview of types of allocators, and their policies and mechanisms. This is a traditional organization of allocators in which implementation mechanism is the key to the division. Although we feel that this kind of organization is misleading, since it overemphasizes mechanism and underemphasizes policy, it is nonetheless the most familiar description. We will attempt to point out policy issues in the appropriate places. Below is a list of allocator mechanisms along with some policies that are commonly implemented with the mechanism.

- *Sequential Fits* include first fit, next fit, best fit, and worst fit policies
- *Segregated Free Lists* include simple segregated storage and segregated fits
- *Buddy Systems* include conventional binary, weighted, and Fibonacci buddies, and double buddies
- *Indexed Fits* use a structured index to implement a desired fit policy

- *Bitmapped Fits* are a particular kind of indexed fit

For a thorough survey of allocator policies and mechanisms, see [25].

3.1 Sequential Fits

We will begin the discussion of allocator types with an overview of the sequential fits mechanism. It is one of the simplest mechanisms and can be used to implement allocators with very low fragmentation (e.g., best fit, address ordered first fit), although in general there are far more efficient mechanisms for these policies.

Many of the allocators used in our experiments are sequential fits algorithms. We stress, however, that this is purely for convenience in varying parameters for experimental purposes. For the better placement policies, more efficient implementations are known, and should be used for any production allocator where speed is an issue [25].

There are a number of important details in the implementation of a sequential fits allocator that can strongly affect its policy and performance. In many previous studies, the descriptions of allocators lack sufficient detail to adequately differentiate one policy from another. This makes it difficult to interpret the results and nearly impossible to repeat the experiments. Thus we have attempted to parameterize our allocators to allow various explicit choices for these details. In particular, the basic three sequential fits allocators we studied (i.e. first fit, next fit, and best fit) can be described by the following (some settings we considered are listed in parentheses, with the default choice in boldface):

- *Free List Order*—the order in which the free list is maintained (FIFO queue, **a LIFO stack**, or address-ordered)

- *SBRK size*—the unit in which memory is requested from the operating system (**4 KB** or 8 KB)

- *List Mechanism*—the mechanism used for maintaining the free list, (**pointers embedded in the free blocks**, a bitmap index structure)
- *Alignment*—the value of which all object sizes must be a multiple, (**8 bytes**, 4 bytes, 1 byte)
- *Coalescing*—determines when and if free blocks are merged with neighboring free blocks, (**immediate coalescing**, deferred coalescing, no coalescing)
- *Splitting Threshold*—determines whether a block should be split up to satisfy a request, (**always split**, split only if the remainder is large enough)
- *Clustering*—the mechanism use to cluster same-sized objects together in the heap (pre-allocation, multi-fit, segregation, **none**)
- *Wilderness Preservation*—searches the most recently sbrk'd area last (**preserve**, don't preserve)

In order to understand the effect of each feature on an allocator's performance, it is necessary to be able to vary one feature while keeping the other features constant. If the features are not carefully separated, their effects may be confused with one another. In some cases our experimental design allows us to"confound" two or more features in the case that they are dependent on each other in a given allocator. Consequently, our experiments are guided by the study of these features in isolation to the extent that this is practical.

Boundary Tags. In order to coalesce free blocks into larger blocks, a mechanism is needed to flag free blocks and indicate their size. One way this is commonly achieved is by Knuth's boundary tag technique. Each block has a header and a footer with a bit that indicates whether the block is free. (The header is at the beginning of the block and the footer is at the end.) The size of the block is also stored in both header and footer. When a block is freed, it can look at the footer of the preceding block, or at the header of the following block, to see if it can be coalesced. In the case of coalescing with the preceding block, the size information in the footer is used to locate the beginning of the block, and the two blocks are merged by adding their sizes together and updating the header and footer for the new combined block. The following block can be coalesced in the same manner.

This technique requires 8 bytes (two 4 byte fields) of overhead per allocated object.[1] This can be reduced to 4 bytes by a simple optimization. The footers can be eliminated by "stealing" a bit from the header in the next object. If blocks are aligned on 8-byte boundaries, then there are three unused bits in the size field that can be used for this purpose (one is used for the previous block and one for the block itself). This bit is used to indicate if the block is allocated or free. So, when a block is freed, in order to coalesce with the preceding block, the "allocated" bit for the preceding block, which is located in the freed block's header, is checked. If it is free, the size information will be stored in the previous block's footer (free blocks still retain the footer). Coalescing with the following block is the same as before.

3.1.1 First Fit

A first fit allocator searches the free list and selects the first block that is at least as large as the requested size. One rationale for this policy is that by selecting the first available block, the search time is reduced. Another is that by keeping the list ordered in a particular way, a particular placement policy can be implemented—somewhat like a priority queue. The implications for fragmentation are that the blocks at the beginning of the free list tend to get split up into small fragments. Thus a search for a larger block will have to get past these fragments to satisfy the request, resulting in longer service times. Our implementation used embedded

[1] In our implementation, the header and footer are both defined as 4 byte integers. A bit for the in-use flag can be borrowed from the lowest order bit of the size field, assuming that there is at least 2 byte memory alignment of heap-allocated blocks.

pointers for the list mechanism and optimized boundary tags for coalescing. The sorting of the list is activated when a block is freed and added to the list. The block is inserted into its sorted position by searching through the list. While this is certainly not an efficient implementation, it serves its purpose in supporting the desired policy.

We will begin the discussion of sequential fit allocators with a description of the address-ordered first fit allocator. This allocator is interesting in that it yields the lowest average fragmentation of any of the allocators we tested. It is a basic first fit allocator with a linear doubly-linked free list with the additional constraint that the free list is sorted by the addresses of the blocks.

One possible reason that address-ordered first fit performs well is that the blocks at the beginning of the list that get split up first are also located at the beginning of the heap; the remainder of the heap is unaffected until a request for a larger block comes in. This has the effect that smaller blocks tend to be at one end of the heap separated from larger ones, and unable to cause fragmentation in the larger blocks. This is somewhat similar to the best fit policy in that large blocks tend to be preserved. Surprisingly, however, FIFO first fit performs almost as well.

Another advantage of an address ordered free list is that coalescing can be achieved without using boundary tags. If a block has a free neighbor in the heap, the two blocks will also be directly linked in the free list.

A novel mechanism for implementing address-ordered first fit and an approximation of best fit was described by Stephensen [20]. A Cartesian tree of free blocks ordered primarily by address and secondarily by block size was used in the "fast fit" allocator. Depending on which search key is used, the allocator can search for blocks by size or by address. If the search is by address, this implements an address-ordered first fit policy. If the search is by size, it is an approximation of best fit. They found that the "better fit" variant (searching by size) performed similarly to the "left fit" variant (searching by address).

The Cartesian tree fast fits allocator may be somewhat slower than the best implementation of some other allocators such as best fit. The Cartesian tree's access times are somewhat higher than simpler binary trees, and performance may not be robust. The tree access times should scale logarithmically with the total number of free blocks, but the dual ordering constraints inhibit rebalancing; this could matter in some cases. (We ignore the cost of coalescing here.)

Address ordered first fit is an attractive policy for a general purpose allocator because of its low expected fragmentation. It also has a good theoretical bound on fragmentation in the worst case—it is nearly optimal within a small constant [18]. The disadvantage of address-ordered first fit is that it may be intrinsically somewhat slower and less scalable than some other good algorithms, such as best fit [25].

Other variations of first fit include FIFO and LIFO ordered free lists, with deferred coalescing, and splitting thresholds.

3.1.2 Best Fit

The best fit allocator was described in the previous section. Variations of best fit include FIFO and address-ordered free lists, deferred coalescing, splitting thresholds, and pre-allocation. Since best fit searches the free list exhaustively, using an address-ordered free list does not make any difference, unless there are more than one block of the same size to choose from.

Best fit is also a good candidate for a general purpose allocator and there exists an efficient implementation (see [19], [25]). A binary tree of linear lists can be used, where each node in the tree holds a linear (LIFO or FIFO) list of free blocks of a particular size. The time cost to allocate from a linear list (once found) is constant, and the time to find the appropriate list scales logarithmically with the number of distinct sizes of free blocks. (We ignore the cost of coalescing here.)

3.1.3 Next Fit

Next fit has been described as a "modified first fit." The difference is that next fit uses a circular free list with a roving pointer. The motivation behind next fit is to make a more scalable first fit allocator by avoiding searching from the beginning of the free list, where small fragments may accumulate. This scalability argument does not apply to address-ordered or size-ordered lists, because the sorting is itself unscalable. The allocator's policy is (mis)guided by the intended mechanism, but there are better mechanisms that implement better policies.

When a request is made, the free list is searched beginning from the point where the last search ended, and a block is selected that is at least as large as the requested size. This may result in allocated blocks being scattered throughout the heap, which has a negative effect on fragmentation [25], and is likely to destroy locality [8], since freed blocks are likely to be isolated from other freed blocks. Variations of next fit include FIFO and address-ordered free lists, splitting thresholds, and wilderness preservation.

Our results show that there is a large amount of variance in the performance of next fit—the average standard deviation for LIFO ordered next fit over all of our allocation traces is 72%, which is several times higher than the average fragmentation for the best allocators. This instability may be caused by a desynchronization with the phase behavior of the program. (For example, one phase may result in a batch of fragments, and later phases may or may not reuse those fragments, depending on the position of the roving pointer). As a result, large spaces are not preserved, and fragmentation is increased. This, along with the difficulty in its implementation, make next fit unattractive for a general purpose allocator.

For a thorough discussion of the next fit allocator, see [25].

3.2 Simple Segregated Storage

A simple segregated storage allocator divides the heap into pages [2] such that blocks of the same size are always allocated in the same page. Typically, requested sizes are rounded up to the nearest power of two, introducing some internal fragmentation, although smaller size classes spacings can be used to reduce the internal fragmentation. When a request is made, the allocator finds the page of memory associated with the adjusted request size. If no memory has been allocated for that size, then the memory is requested from the operating system, and the page is divided up into blocks of the requested size.

The advantage of this scheme is speed, since no searching is necessary. Another advantage is that there is no per-block overhead required on allocated blocks. The size information can be stored in a page header rather than in the blocks themselves. This allows for smaller minimum block sizes.[3] The disadvantage is that once a page has been dedicated to a certain size, it cannot be reused for any other sizes. If an application allocates many blocks of a certain size, frees them, and then repeats this behavior for other sizes, large amounts of free space become unusable and contribute to severe external fragmentation.

Simple segregated storage is not particularly attractive as a general purpose allocator since it is subject to extreme external fragmentation as well as significant internal fragmentation. Other allocators can probably be made comparably fast using deferred coalescing.

In some situations, however, simple segregated storage could be useful. If it is known that data structures accumulate monotonically, or that ratios of numbers of objects with different sizes will not change, then external fragmentation may not be an issue.

[2] These "pages" need not be identical to virtual memory pages. For example, four 1K allocator pages could be mapped to one 4K virtual memory page.

[3] There is no need for headers or footers, so the size can be stored in an external data structure.

3.3 Segregated Fits

Segregated fits is designed intentionally to approximate best fit ([16]). A segregated fits allocator uses a separate free list for every size, or for *size classes* (i.e., small ranges of sizes). When a request is made, the free list corresponding to that size is searched for a free block. When a free list is empty, the allocator looks for blocks in the next higher size free list until a large enough block is found. If the blocks in each list are the same size, or close in size, then the first block can be selected without any searching. Similarly, freed blocks are returned to the appropriate free list.

If there are enough free lists to satisfy all of the program's requests, then this policy approximates best fit since the block that is returned is the smallest size that is at least large enough to satisfy the request. What this amounts to is an efficient mechanism that approximates a best fit policy under the right conditions (i.e. if there are enough free lists with closely spaced size classes). There is a tradeoff between the amount of storage used for the free lists (the more free lists there are, the more storage must be set aside) and the amount of internal fragmentation due to inexact matches of request sizes. Alternatively, a sequential fits search algorithm can be used to search the appropriate free list.

The segregated fits allocator we used in our experiments was written by Doug Lea (version 2.5.1); for some time, it was the standard malloc distributed with the GNU C++ compiler. Since then, Lea has also written a newer version that is a better approximation of best fit and removes the footer overhead on allocated objects.

3.4 Buddy Systems

A buddy system allocator constrains splitting and coalescing to make these operations fast and simple. It is a variant of a segregated lists allocator. In a buddy system, the heap is divided up recursively into smaller and smaller sizes. The actual organization of the heap at any given time depends on the sizes of allocated objects. At each level, there are two possible blocks, which are further subdivided into two more blocks, and so on. When a request is made, the request size is rounded up to the closest possible block size in the hierarchy of blocks. A block's *buddy* is the other block at the same level of the hierarchy. When a block is freed, it can only be coalesced with its buddy. The buddy is considered allocated if there are any allocated blocks within it (there may be only a small portion of the buddy being used, but it must still be considered allocated for the purposes of coalescing). Consequently, the block resulting from coalescing will be a block at the next higher level of the hierarchy.

The advantage of a buddy system is that a block's buddy can be located quickly and easily. Furthermore, only minimal overhead per object (one bit) is necessary to indicate whether the buddy is completely free. Unfortunately this low space overhead is only possible if the free routine is supplied with an argument specifying the size of the block being freed. The size is needed for coalescing. In practice, an extra word of memory is usually required to store the size.

In a *binary buddies* allocator, all block sizes are a power of two, and each size is divided into two equal parts. A disadvantage of binary buddies is that internal fragmentation is usually high because object sizes must be rounded up to the nearest power of two. A *double buddy system* uses two buddy systems, with staggered sizes, to allow a closer spacing of size classes. (For example, one buddy system might use sizes that are powers of two, while another uses sizes that are spaced in intervals of powers of two that start at a different size). This reduces internal fragmentation by about half, but blocks in one size series cannot be coalesced with blocks in the other.

Other variations of buddy systems exist that use a different scheme to organize the hierarchy of blocks, such as Fibonacci systems and weighted buddy systems. We did not consider these variations in our experiments, since their expected internal fragmentation is higher than other allocators.

Buddy systems are not particularly attractive as general purpose allocators; as we will show, their internal fragmentation alone is generally higher than the total fragmentation of better allocators. For some applications, however, buddy systems have some advantages. For instance, real time applications can take advantage of its fast and predictable coalescing, and the fact that fragmentation in the worst case is comparable to other allocator's worst cases. Applications in which the block sizes can be constrained to the buddy system's block sizes also would not incur any internal fragmentation (e.g., storage of compressed pages of virtual memory, which don't have to be stored contiguously).

3.5 Linear and Segregated Linear Allocation

These allocators are special in that they are not designed to be used as real allocators at all. Rather, they will be used in experiments to study allocation behavior of applications [9].

The linear allocator starts allocating blocks at the beginning of the heap and continues allocating at the next available location without ever reusing freed blocks. This allocator is used only as a test of some inherent fragmentation-related behavior in a trace. It provides a basis to separate program behavior from allocator interactions. Similarly, the segregated linear allocator serves the same purpose for analyzing the segregated lists allocators. It uses the same basic algorithm as the linear allocator, but it separates the heap into pages, each of which is used for a single power of two size class. Blocks are allocated linearly within each page.

Chapter 4

Experimental Design

The design of our experiments was motivated by three factors. The first is that we wanted to use allocation traces collected from real applications. This is important because allocators may perform differently with real applications than with randomly- generated allocation traces. The second issue is to use a variety of allocation policies. This was accomplished by creating a simulation framework which can be linked with any allocator. We also wrote a library of allocators which are parameterized to implement a wide range of policies. The third issue is to test the hypothesis that an allocator's performance on a randomly generated trace predicts its performance with a real application. We will show that the correlation between these quantities demonstrates that the two are not related to a useful extent.

We designed a set of experiments to test the performance of allocators using various additional features such as deferred coalescing, clustering, splitting thresholds, and wilderness preservation. The results show that on average, none of these features make a significant difference in the fragmentation of the basic allocator, although with some application traces the difference was more significant. This is a useful result because some of the features have beneficial side effects, such as improved locality, that can be taken advantage of without increasing fragmentation.

4.1 The Applications

This study has been primarily an empirical one since we have chosen to use allocation traces of real programs. We chose a sample of 8 programs that use dynamic memory allocation. Several criteria were involved in these choices. The program must allocate enough memory to observe an allocator's behavior, including a significant amount of live data. It must also use the standard allocator function interface (i.e. `malloc()`, `free()`, `realloc()`, etc.) and not exhibit memory leakage. The source code must be publicly available for reproduction of results. As much as possible, the programs were chosen from a cross-section of types of applications. The programs also demonstrate a variety of allocation behaviors. The programs that were chosen are :

- *Espresso* is a circuit analyzer used to optimize programmable logic arrays.
- *GCC* is the parsing, code generating, and optimizing stage of the GNU C compiler (version 2.5.1) with some modifications to remove hand tuned allocation mechanisms.
- *Ghost* is GhostScript, a PostScript interpreter written by Peter Deutsch. This version contains modifications by Zorn, *et al.* to remove hand-tuned allocation mechanisms.
- *Grobner* is a mathematical tool that decomposes functions into a linear combination of Grobner basis functions using rewrite rules.
- *Hyper* is a network simulation program for analyzing hypercube communications. Written by Don Lindsay.
- *LRUsim* is a utility for analyzing the locality characteristics of an application whose memory reference trace is provided as input. Written by Douglas Van Wieren.

program	KB alloc'd	run time	max obj's	max KB	avg lifetm	language
Espresso	104,388	146	4,390	263	15,478	C
GCC	17,972	167	86,872	2,320	926,794	C
Ghostscript	48,993	53	15,376	1,110	786,699	C
Grobner	3,986	8	11,366	145	173,170	C++
Hyper	7,378	131	297	2,049	10,531	C++
LRUsim	1,397	29,940	39,039	1,380	701,598	C++
P2C	4,641	30	12,652	393	187,015	Pascal/C
Perl	33,041	114	1,971	69	39,811	Perl/C
Average	27,725	3,823	21,495	966	355,137	—

Table 4.1: Basic Statistics for the Eight Test Programs

- *P2C* is a program that translates Pascal code into C code. Written by Dave Gillespie at CalTech.

- *Perl* is the interpreter for the Perl scripting language. This version contains modifications by Zorn to remove hand tuned allocation mechanisms.

Table 4.1 contains a summary of statistics related to the applications independent of any allocator. The first column, KB alloc'd, is the sum of the sizes of all allocation requests throughout the application's entire execution. Run time is the program's run time measured in seconds. Max obj's is the largest number of objects allocated at any given time in the application's lifetime. Similarly, Max KB is the largest amount of total allocated memory at any given point in the application's lifetime (not necessarily the same time as Max Obj's). Avg lifetm is the average of the lifetime of each object, from the time it is allocated to the time it is freed. Finally, language is the programming language in which the application was written. For more details about the applications themselves, see [9].

4.2 Allocator Features

In order to study the effects of different policy choices in allocators, we devised a set of features that can be turned on or off by parameters to the allocators. Most of these features apply mostly to the sequential fits allocators. The features either don't make sense with the other allocators or the allocators are too complicated to add the feature without affecting its basic policy. The other allocators will be used for comparison. According to our results the other allocators (segregated fits, simple segregated storage, and binary buddies) are outperformed by best fit and address-ordered first fit, and the additional features are not likely to make much difference without affecting the basic character of the allocator. Following is a description of the features we have focused on in this study.

4.2.1 Features Held Constant

Some of the parameters were held constant in all the allocators to support comparison. The minimum block size is 16 bytes (four 32 bit words) because that is lowest possible block size for some allocators. The memory alignment is assumed to be 8 bytes (two 32 bit words), since this is a common alignment on many system architectures. If these values could be changed, fragmentation would likely be affected significantly. For example, if an application allocates many four byte blocks, then rounding these up to the 16 byte minimum block size or padding them to 8 bytes for alignment, could increase memory waste by several times.

4.2.2 List Order

The free list order in a sequential fits algorithm can drastically affect the allocator's policy and performance. Whether they are maintained as FIFO queues, LIFO stacks, or address-sorted lists has an impact on the placement of objects. Previous studies have examined some of these variations, but not as a separate issue from the underlying allocator policy or the implementation. Intuitively, LIFO stacks are attractive in that they are likely to support good locality of reference. On the other hand, our results show that FIFO queues yield lower fragmentation.

4.2.3 Deferred Coalescing

When an object is freed, there is a possibility that the object may be merged with a neighboring free block to create a larger free block in the heap. This is known as *coalescing*. Without coalescing, free blocks are never merged and memory may be wasted because a request for a large object cannot use the space occupied by multiple smaller free blocks. In one kind of coalescing, an attempt to coalesce a block is made when the object is freed. This is called *immediate coalescing*. The disadvantage of immediate coalescing is that it may incur unnecessary work in coalescing small blocks that will be split up again when a new request is made.

With this technique, coalescing is delayed so that blocks can be reused without unnecessary work. The question then is when to coalesce. One solution is to coalesce all freed objects when a request cannot be satisfied with the available free blocks in the heap. This way, if small blocks are repeatedly allocated and freed, they will not be unnecessarily coalesced. They will only be coalesced if a request for a larger object cannot be satisfied. This is the approach we used in our deferred coalescing allocators.

Another possibility would be to defer coalescing until a fixed amount of memory had been allocated, or until a fixed number of requests had been served. We did not include any of these variations in our experiments.

In our deferred coalescing allocators, if no sufficiently large block is available to satisfy a request, full coalescing is attempted (i.e. for every coalescable free block). If there is still not enough memory to satisfy a request, more memory is requested from the operating system. This is always done in units of 4K pages[1]. Experiments

[1] Actually, the double buddy allocator uses 4K and 6K blocks.

with 8K pages made little difference in the results, but the difference was always that the 8K pages caused slightly more fragmentation. The remainder of the page is made available by appending it to the free list. (In the case that a block larger than the page size is requested, the size requested from the operating system is the smallest multiple of the page size that is at least as large as the requested block.) Another way to handle this is to split the memory from the operating system and take only as much as is needed for the request. The remainder of the block can be kept apart from the free list until coalescing is needed again. Coalescing may also be performed at some regular interval of bytes of allocation, or after a certain number of requests have been made. This may cause unnecessary coalescing in some cases, if the requests can be satisfied without the coalescing. (These kinds of deferred coalescing was not included in our experiments.)

Deferred coalescing has been viewed in the past as a means of improving the speed of an allocator. Effects on fragmentation have been largely unexplored. Our results show that deferred coalescing has little impact on fragmentation for our test programs and the policies with which we have applied it. It appears that if there is a significant improvements in speed, there is no reason not to use deferred coalescing in any of the allocators we tested.

4.2.4 Clustering

The main cause of fragmentation is that some blocks are freed, but their neighbors are still allocated. This creates an isolated free block that cannot be coalesced. If there are no more requests for blocks of that size, the space becomes unusable. If an allocator knew when every object would be freed, it could place together those objects which will be freed at the same (or nearly the same) time, theoretically eliminating all fragmentation. Since this information is not known in advance, no such allocator can be written. Other properties of objects may give clues to the time they will be freed, or at least the fact that they are likely to be freed at different times. (We refer to this as *death time discrimination* [25].) If different-sized objects are likely to be of different types, and objects of different types are likely to die at different times, then grouping same-sized objects together may reduce fragmentation.

The allocation traces of our test programs show that for any given application most allocation requests fall into a small set of sizes. An allocator can take advantage of this by setting aside a pool of contiguous blocks of common sizes. If these blocks have similar lifetimes then fragmentation will be lowered by the segregation of sizes. This can be accomplished by a *multiple fit* allocator.

Multiple Fits. A multiple fits allocator uses one of the basic algorithms with a modification. (For simplicity of implementation, we based our multiple fits allocator on the best fit policy.) When the free list is searched, a block is selected whose size is an integral multiple of the request size. If an exact multiple does not exist, then the search reverts to the standard algorithm of the allocator being used. If there is more than one block that can be selected, the allocator may use a function to discriminate among them (e.g., smallest, largest, first, etc.) Again, this works best with deferred coalescing. If enough blocks of the right sizes are available, this technique can reduce fragmentation by eliminating gaps that are due to splitting.

One variation of multiple fit is *half fit* [5]. A half fit allocator searches for a block that is twice the requested size. The search may be for a block that is *exactly* twice as large, or for an approximate match. If an approximate match is used then there may be some external fragmentation caused by splitting the remainder. Our implementation uses an approximate match which specifies that the size of the block returned is at least twice as large, and no more than 5% larger, than the requested size.

4.2.5 Wilderness Preservation

When an allocator cannot satisfy a request from the available free space, it must call on the operating system for more memory. If the system call is successful, the request can be satisfied from the new memory. Then there is a choice in satisfying subsequent requests of whether to start searching in the old heap area or in the new heap area.

An allocator using the *wilderness preservation heuristic* always starts the search in the old heap area [11]. The idea is to save the largest block of free memory until last in order to prevent fragmentation. This is similar to the strategy implemented by a best fit allocator—avoid breaking up large blocks if a smaller one is available. In fact, our implementation of best fit and first fit allocators use wilderness preservation by default, since the new heap area is added to the end of the free list. Since the free list is searched from the beginning, the new heap area will only be considered after every other space has been searched. With next fit, however, this is not the case. There is no "beginning" or "end" of a circular free list. So we constructed a modified next fit allocator that always saves the new heap area as a last resort in satisfying requests. The simulations results show that fragmentation is reduced. However, this does not entirely correct what we believe is a fundamental flaw in the next fit policy—objects are systematically scattered through the heap, causing increased fragmentation.

4.2.6 Splitting Thresholds

In many allocators, the block that is chosen to satisfy a request may be larger than the requested size. One way to deal with the excess part of the block is to allocate the whole block and waste the extra space. This results in an increase in internal fragmentation. Another way is to split the block into a requested part and an excess part and return the excess part to the free list. This may add to external fragmentation if the excess block is not a size that is needed. In order to balance the internal and external fragmentation, the decision to split can be filtered by a splitting threshold. That is, if the excess part of the block is larger than a given fraction (i.e., the splitting threshold) of the entire block then it is split. This is a way to prevent the accumulation of small fragments from split blocks. Theoretically, the splitting threshold can be calibrated to minimize the overall fragmentation. In practice, the value depends upon parameters that are dependent on the allocation behavior of the application program. If splitting is restricted too much with a large threshold, there will usually be more internal fragmentation than is saved in external fragmentation.

We ran simulations with splitting thresholds of 7% and 14% to determine how this parameter affects fragmentation. These values were chosen to give us a sample of the fragmentation curve with varying threshold values. The lower value is just large enough to have some effect (although much of the time it may not be triggered). The larger value is not so high as to overshoot the critical point by far. The goal is to understand the trend of fragmentation with varying thresholds and determine the threshold point where fragmentation is minimized.

4.3 The Experiments

In order to understand the interactions between features of an allocator (i.e. deferred coalescing, splitting threshold, clustering, wilderness preservation) we designed a set of experiments using a fractional factorial design. A factorial design assumes that there are a number of independent factors which can be set on or off. In this case these are our allocator features. In fact, these features are not entirely independent, but the design still applies to a subset of the features. A fractional design allows us to choose a reasonable and promising subset of allocators from all of the possible combinations.

4.3.1 Limiting the Search Space

The number of experiments can be reduced by filtering out those allocators that we expect to yield poor fragmentation regardless of which features are supplied. For example, the internal fragmentation in a binary buddies allocator is greater than the total fragmentation of the best sequential fits allocators, so it is unlikely that any features will improve this without changing the basic character of the allocator. Also, with some allocators, some features are not possible or don't apply. Some features only make sense in combination with other features. For instance, multiple fits works best with deferred coalescing (without deferred coalescing, multiple fits blocks will immediately be coalesced, defeating the multiple fits).

4.3.2 Testing Effects of Randomization

We also designed a second set of experiments to test the assumptions behind the traditional methodology for allocator studies. Many of the previous studies in dynamic memory allocation made the assumption that the performance of an allocator on real application traces is related to its performance on randomly generated traces. We felt that this is an invalid assumption and motivated us to use real application traces in this study. A weaker assumption than that is that the performance of the allocator is independent of the order of requests. This led us to build the trace shuffler tool. By randomizing the order of requests in a trace and leaving the object sizes and lifetimes unchanged, we were able to determine the correlation between allocator performance on the original traces with respect to the performance on the shuffled traces. The results showed that there was a low correlation (0.5), meaning that only 25% of the variation in the randomized trace is explained by the original trace; the rest is noise. This can be explained by the fact that the patterns of allocation present in the original trace are destroyed in the shuffling process. These patterns are therefore essential to the performance of an allocator. Since it is very unlikely that any such patterns are present in a random sequence of requests, allocator studies that use randomized traces are fundamentally flawed.

Chapter 5

Profiling and Simulation Framework

Our methodology for the study of allocators depends on the use of allocation traces from real applications. We used these traces to drive simulations with different allocators. Each simulation returns a set of statistics that indicate the performance of the allocator with the given allocation trace data.

In order to analyze fragmentation in the heap image of a particular program, a simulator was constructed using a bitmap representation of the heap. Each element of the bitmap represents a word (4 bytes) of heap space. The simulator, coupled with a real implementation of an allocator, allows us to examine the fragmentation of the heap at any given time in the life of an application. This tool is used as a framework for profilers and analyzers that return useful information from the data in the heap representation. The primary results given by the analysis tools include:

- Statistics that are independent of the allocator
  - Number of Objects—the total number of allocation requests over the application's lifetime

- Number of Object Sizes—a breakdown of the number of objects by size
- Maximum Number of Objects of Each Size—the maximum number of objects of each size live over the lifetime of the application

- Statistics that depend on the allocator
  - Heap Size—the maximum heap size over the application's lifetime
  - Amount of Live Memory—the amount of allocated memory in the heap at the point of maximum heap size
  - Number of Live Objects—the number of allocated objects at the point of maximum heap size
- Profiles of allocator/trace interaction over time
  - Page Count Data—a multi-resolution measure of fragmentation over time
  - Object Count Data—a profile of memory usage over time by object sizes

Depending on the parameters to the simulator, allocator overhead may be counted as internal fragmentation, as part of the object, or ignored all together. This feature let us determine the amount of wasted heap space due to allocator overhead.

5.1 Data Analysis

Once the data is collected from a simulation for a particular trace-allocator combination, it can be used to create data visualization graphs and used for statistical analysis. We used several kinds of graphs to analyze the data—object count data graphs (*ocd*), page count data graphs (*pcd*), memory density graphs, and neighborhood fragmentation graphs. Note that not all of these are used in this thesis, but the descriptions are included here for completeness. See [9] for. more analysis with these tools.

OCD Graphs. The ocd graphs show the amount of memory used by each of the top n sizes. The Y axis is the amount of memory and the X axis is allocation time. This gives us a visual representation of the memory usage by size classes.

PCD Graphs. The pcd graphs show the amount of memory in use at varying page sizes (from 8 bytes to 64K), where a page is an arbitrary unit of memory. If a page contains memory in use, then the whole page is considered in use. This allows us to examine the distribution of contiguous occupied and vacant regions in the heap. It is used to find clusters of live objects over time at different spatial granularities.

Memory Density Graphs The memory density graphs show a greyscale representation of memory usage in the heap. Darker areas represent more densely used memory, while lighter areas are sparsely used memory. This graph is used to visualize the distribution of objects in the heap over an application's lifetime. The Y axis is the heap address space and the X axis is allocation time. It can be used to find unused areas in the heap that might be used more efficiently by another allocator.

Neighborhood Fragmentation Graphs These graphs are used to determine the percentage of objects that die at around the same time as their neighbors.

5.2 Simulation Tools

Several steps are involved in running a simulation for a particular trace-allocator combination. These steps have been automated to a large degree in our tool environment by the use of shell scripts and make files. The makefiles, in addition to controlling the build process, are used to run batches of simulations and distribute them over a heterogeneous set of systems. Shell scripts are used for task management and for spawning the data analysis tools. The simulations were run on

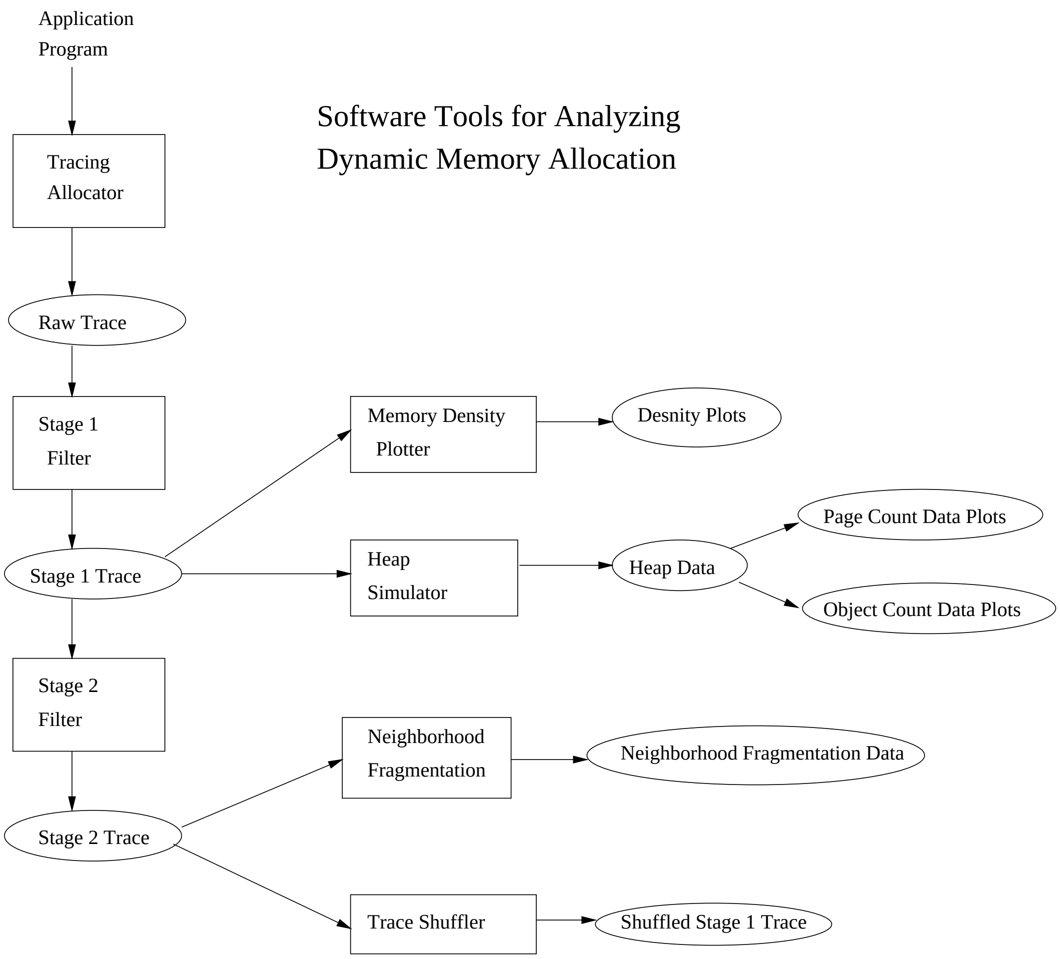

Figure 5.1: Software Tools

an assortment of SPARC and IBM RS6000 workstations. The timing simulations in table 4.1 were all run in the same environment (SPARC), separately with most of the machine's resources available for the application. The various stages of the simulation tools are illustrated in figure 5.1.

In order to streamline the simulations, there are three stages of processing. Allocation trace data collected by the tracing allocator is in the *raw trace* format. This format is the lowest level in that it contains information that is specific to the context of the trace and the allocator used. This data is then filtered to get a *stage 1* trace format. The filter abstracts away allocator-specific information and preserves information needed to use the trace with any allocator. A second filter generates a *stage 2* trace format. This filter preserves all the information in the stage 1 trace, but combines records for allocating and freeing an object. This format is more convenient for some kinds of analysis and supports the creation of synthetic traces with given size and lifetime distributions.

5.2.1 Obtaining the Raw Trace File

First, memory allocation traces are obtained by running the application with a specially instrumented allocator which creates a trace file with a record for each allocator operation—malloc, free, or realloc.[1] The format of the raw trace file that is produced by the tracing allocator is as follows :

[1] Other more specialized allocation functions exist in most implementations, e.g., mem_align, calloc, and valloc, but these were not considered in this study because they were not used in any of the programs that we traced. Also, they do not significantly change the behavior of the allocator. They are more restricted forms of allocation and are usually implemented by placing more constraints on the malloc function.

Field	Bytes	Type
Type	1	(FREE_OP, REFREE_OP[2],
		MALLOC_OP, REALLOC_OP)
Size	4	integer
Address	4	unsigned integer
Time(sec)	4	integer
Time(μsec)	4	integer

5.2.2 Stage 1 Processing

Then the raw trace file is filtered by the stage 1 filter utility to create a stage 1 trace file. The purpose of this filter is to replace the address of each record associated with a particular allocation request with a serial numeric identifier, or ID. That is, the same ID is assigned to the malloc record, realloc record(s) (if any), and free record associated with an object. The resulting trace file is used by the heap simulator. The format of the stage 1 trace format is as follows :

Field	Bytes	Type
Type	1	(FREE_OP, MALLOC_OP, REALLOC_OP)
Identity	4	integer
Size	4	integer
Time(sec)	4	integer
Time(μsec)	4	integer

[2] A REFREE operation is generated by the tracing allocator in order to keep track of the original address of an object to be reallocated. This is always followed immediately by a REALLOC_OP.

5.2.3 Stage 2 Processing

The stage 2 filter utility takes a stage 1 trace file as input, and abstracts away the different allocator operations. The filter records the birth time and death time (measured in allocation time) of each object in the input trace file and writes that information, along with the size, into a single record in the output file. The order of the records in the stage 2 trace is the same as the order of the malloc or realloc records for the corresponding objects in the stage 1 trace. Note that the same ID may appear more than once in the stage 2 trace file only in the case that a reallocate operation was called on an object. Since we are not concerned with real time performance in this part of the study, the time stamp fields of the records in the raw trace file are discarded by the stage 2 filter. The stage 2 trace file is used by the neighborhood fragmentation tool and the shuffler. The format of the stage 2 trace file is as follows :

Field	Bytes	Type
Identity	4	integer
Size	4	integer
Birth Time	4	integer
Death Time	4	integer

5.3 The Trace Shuffler

The trace shuffler takes an allocation trace as input and creates a synthetic trace file with the same object size and lifetime distributions as the input. The only difference is in the order of the allocation requests. The effect of this transformation is that all the information in the input trace is preserved except for the patterns of allocation exhibited by the application. We use this tool to show that the allocation patterns of an application are essential to the allocator in determining the best placement of objects to minimize fragmentation.

The shuffler takes a stage 2 trace file as input and randomizes the order of the requests. The ID field in each record in the input is assigned a random number and the records are then sorted using the random number as the sorting key. The resulting trace file is then converted back into a stage 1 trace file by generating malloc, reallocate, and free records for each object from the birth time and death time information in the input record. This trace can then be used in exactly the same manner as the original trace file.

5.4 Allocator Overhead in the Simulation

Since we want to separate allocator overhead from our fragmentation results, the heap simulator can compensate for header and footer overhead by subtracting it from the request size. This removes the header and footer overhead from the simulator's representation of the heap, but leaves other kinds of overhead.

Several factors can contribute to the overhead of an allocator—extra space for free list pointers, size fields for boundary tag coalescing, memory alignment requirements, and minimum block size requirements. The heap simulator can be given different parameters to specify which of these factors to compensate, except for memory alignment and minimum block size. The are two kinds of data in the results section—those that reflect header overhead for a *good* implementations, and those that ignore header and footer overhead. Results for fragmentation with no overhead (from any of the above factors) will be shown in [9]. Preliminary results show that overhead accounts for more of the wasted space than was previously expected.

Chapter 6

Results

The fragmentation results below are given in terms of the percentage of wasted memory to peak memory usage (see definition of fragmentation in section 2.8), including waste from minimum block sizes and memory alignment requirements (but not headers, footers, or size fields). The data in section 6.6 uses waste measurements, in addition to fragmentation. Waste results include different amounts of header overhead for different allocators, reflecting what we believe to be a *good* implementation of each allocator. This is a bottom-line result to compare good implementations. Fragmentation results attempt to give a clearer picture of *policy* differences. The fragmentation results still reflect some overheads, notably effects of minimum block sizes and 8 byte alignment, but they have been held constant so that the relative performance for policy questions is valid, even though the baseline is higher than a perfect implementation would give.

6.1 List Order

The free list order in a sequential fits allocator is usually FIFO, LIFO, or address-ordered (AO). The results of our experiments show that address-ordered lists are generally the best in terms of fragmentation, followed closely by FIFO. It is a sur-

Application	First Fit			Next Fit			Best Fit		
	AO	FIFO	LIFO	AO	FIFO	LIFO	AO	FIFO	LIFO
Espresso	4.81	4.81	13.93	4.81	4.81	13.93	4.81	4.81	4.81
GCC	8.26	8.43	31.01	8.43	8.43	34.12	8.26	8.26	8.26
Ghostscript	5.56	7.72	186.78	19.61	52.40	212.72	5.20	5.20	5.92
Grobner	54.42	54.42	120.60	70.97	84.76	101.30	54.42	54.42	54.42
Hyper	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16
LRUsim	11.27	11.27	11.27	11.27	11.27	11.27	11.27	11.27	11.27
P2C	15.01	15.01	16.03	19.08	22.14	25.19	15.01	15.01	15.01
Perl	15.77	15.77	15.77	21.56	21.56	21.56	15.77	15.77	15.77
Average	14.41	14.70	49.44	19.49	25.69	52.53	14.36	14.36	14.45
Std Dev	17.01	16.85	67.26	22.15	28.85	71.74	17.03	17.03	16.98

Table 6.1: Fragmentation Results for Sequential Fits Allocators with Different List Orders

prising result that FIFO lists do almost as well as address-ordered. In Fact, for the best allocators, first fit and best fit, the difference between address-ordered and FIFO are not statistically significant. (This is not surprising for best fit, where list order is only used to break ties between common sized blocks, but it is very surprising for first fit. It is useful because FIFO is much cheaper to implement than address-ordered.) LIFO list performance is far worse. LIFO may not work as well as FIFO because it reuses objects immediately, irrespective of size, and does not give their neighbors time to die and be coalesced before reusing the free block.

Our results also show that LIFO next fit has a large amount of variance among the traces (72%). We believe that this is due to the roving pointer's not being synchronized with the phase behavior of the program. FIFO and address-ordered next fit have both lower fragmentation and lower variance than LIFO.

Figures 6.1 and 6.2 show the memory density plots for LIFO first fit, address-ordered first fit, and LIFO best fit with the P2C application. The trace shows a common pattern of ramped memory allocation with phases. Clearly address-ordered first fit has less fragmentation than LIFO first fit. The allocated blocks are more compact and pushed towards the beginning of the heap. LIFO first fit suffers from more fragmentation, possibly due to splitting up large free blocks to satisfy smaller requests. LIFO best fit compares favorably with address-ordered first fit. The similarity between the two plots suggests that the strategy behind these two policies may be related [25].

6.2 Deferred Coalescing

The mechanism we used for implementing deferred coalescing uses "quick lists" to store freed blocks of the same size. There is a quick list for each size from 1 to 64 bytes. Deferred coalescing is restricted to blocks that are less than or equal to 64 bytes because the overhead for coalescing larger blocks is not as high (i.e., the ratio of overhead to block size is lower for large blocks than for small ones). Any object under 64 bytes that is freed is put on the appropriate quick list and no coalescing is attempted. Then whenever an object under 64 bytes is requested, the allocator looks first in the quick list. This effectively prevents unnecessary coalescing of small blocks. If there are no blocks of the requested size on the quick list, or the request is for a size greater than 64 bytes, the allocator searches for a block using its own policy.

There are also three variants of deferred coalescing. The quick lists can be FIFO, LIFO, or address-ordered. It is especially interesting to look at the performance of allocators using different list orderings for the general allocator than for the quick lists.

The question we tried to answer is "When does deferred coalescing affect fragmentation?" In order to find a solution, we ran a number of simulations with various allocators using deferred coalescing, and compared the fragmentation results with those of simulations without deferred coalescing. Some of the results are summarized in the tables 6.2 and 6.3.

Note that there is a small amount of error in the data due to the quantizing of heap space by 4K pages; the differences in the amount of waste between the different variations of deferred coalescing are negligible. In fact, there is no statistically significant difference between the amount of waste with deferred coalescing and

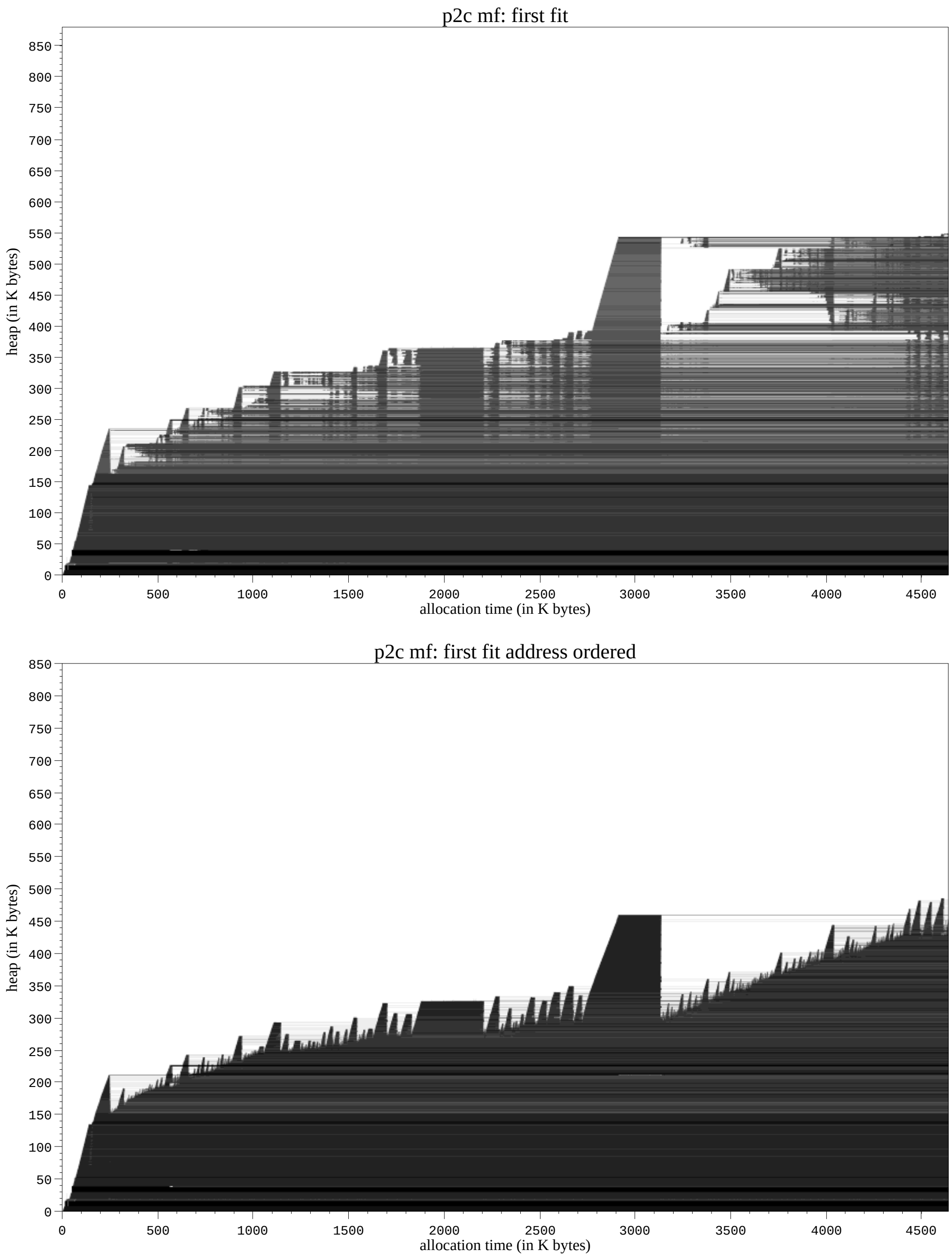

Figure 6.1: Memory Density Plots for P2C—LIFO and Address Ordered First Fit Allocators

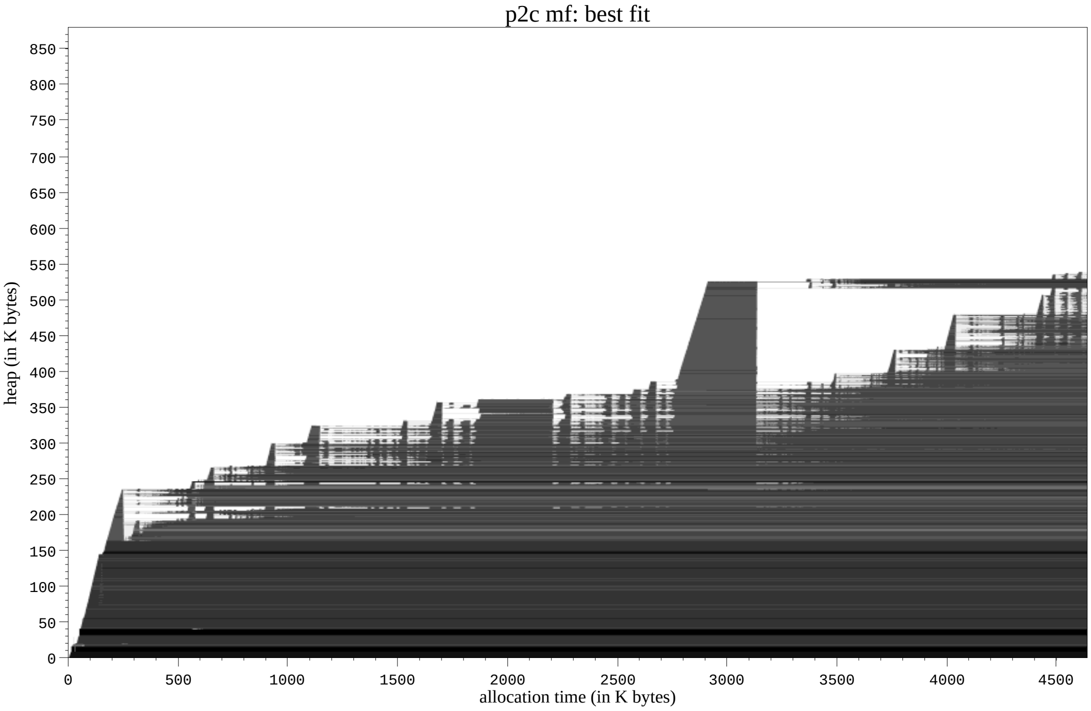

Figure 6.2: Memory Density Plot for P2C and LIFO Best Fit Allocator

Application	Without D.C.	D.C. Quick List Order		
		AO	FIFO	LIFO
Espresso	4.81	4.81	4.81	4.81
GCC	8.26	13.60	24.98	19.29
Ghostscript	5.56	5.56	5.56	5.56
Grobner	54.42	51.67	51.67	51.67
Hyper	0.16	0.16	0.16	0.16
LRUsim	11.27	11.27	11.27	11.27
P2C	15.01	13.99	13.99	13.99
Perl	15.77	15.77	15.77	15.77
Average	14.41	14.60	16.03	15.32

Table 6.2: Fragmentation Results for Address-Ordered First Fit with Deferred Coalescing

Application	Without D.C.	D.C. Quick List Order		
		AO	FIFO	LIFO
Espresso	4.81	4.81	4.81	4.81
GCC	8.26	24.98	24.98	24.98
Ghostscript	5.92	5.92	5.92	5.92
Grobner	54.42	51.67	51.67	51.67
Hyper	0.16	0.16	0.16	0.16
LRUsim	11.27	11.27	11.27	11.27
P2C	15.01	13.99	13.99	13.99
Perl	15.77	15.77	15.77	15.77
Average	14.45	16.07	16.07	16.07

Table 6.3: Fragmentation Results for LIFO Best Fit with Deferred Coalescing

the amount of waste without deferred coalescing for any of the above experiments.[1] While this is a negative result for deferred coalescing, it is useful in that it shows that deferred coalescing does not increase fragmentation significantly for the applications in our test suite. Therefore it can be used for speed increases and to improve locality without negatively affecting fragmentation.

While deferred coalescing did not have a significant effect on fragmentation on average, it did decrease fragmentation with the Grobner and the P2C application traces. In only one trace, GCC, did deferred coalescing increase fragmentation.

The results of these simulations can also be compared with those of the Lea allocator, which uses a more complicated kind of deferred coalescing. Unfortunately, direct comparisons are difficult. Lea's allocator is a segregated fits allocator. It also uses deferred coalescing, but it is different than the kind of deferred coalescing in our other allocators.

When coalescing is triggered by a request that cannot be satisfied from the free list, coalescing is performed only until enough space has been made available for the request, and no more. This optimization is intended to further reduce unnec-

[1] This was calculated by a test of equality of two independent normal distribution at 95% significance level.

essary coalescing of small blocks and decrease service times for requests. Also, the quick lists are searched using a roving pointer as in the next fit algorithm. While the intended effect may be achieved in practice, it is difficult to analyze its impact due to the unknown interactions between other features of the allocator. The average amount of waste is 31.07%, which is significantly higher than the best fit or address-ordered first fit allocators. The deferred coalescing behavior has been changed in newer versions of this allocator.

6.3 Clustering

Several variations of clustering were tested in our experiments. We wrote allocators that implemented multiple fits (two variations) and half fits.. The multiple fits and half fit allocators also use deferred coalescing. Clustering allocators must use some kind of deferred coalescing or else the benefit is lost.

6.3.1 Multiple Fits Results

The goal in a multiple fits allocator is to group together objects with similar lifetimes, using the time of allocation as a heuristic for lifetime. This should create more free space between allocated objects. The key parameter to the multiple fit allocator is the maximum amount of memory to be set aside in a single request. If it is too small, then there may not be enough extra space to use for allocation requests. On the other hand, if it is too big, large free spaces may be used when smaller spaces would cause less fragmentation. There is no easy way to determine the best size for this parameter.

We wrote two variants of the multiple fit allocator that both use LIFO best fit as the basic allocator, with LIFO quick lists for deferred coalescing. The first variant (multifit-max) searches for the largest block whose size is an integral multiple of the request size. The second variant (multifit-min) searches for the smallest such

Application	LIFO Best Fit	Multifit_Min	Multifit_Max	Half Fit
Espresso	4.81	4.81	13.93	4.81
GCC	8.26	24.98	37.39	25.15
Ghostscript	5.92	8.81	15.29	5.92
Grobner	54.42	59.94	68.21	51.67
Hyper	0.16	0.16	0.16	0.16
LRUsim	11.27	11.27	11.27	11.27
P2C	15.01	17.05	16.03	17.05
Perl	15.77	15.77	15.77	15.77
Average	14.45	17.85	22.26	16.47

Table 6.4: Fragmentation Results for Allocators with Clustering

block. The selected block is then split up into blocks of the requested size. The maximum amount of memory split up in this way is defined by a constant to be 1K. If no such block is available, then the allocator reverts to best fit and selects the smallest block that satisfies the request.

These results should be interpreted with some caution since the performance of a multiple fits allocator is very sensitive to having right amount of clustering, and this parameter depends on the input stream.

Half Fit. A special sort of multiple fit allocator is the half fit allocator. Half fit uses best fit with LIFO free list order as its basic allocator. When a request is made, the free list is searched for a block that is approximately twice the requested size. The block that is selected must be at least twice the requested size and no more than 2.05 times the request size. (The constant 2.05 was chosen arbitrarily. The impact of this value has not yet been studied.) Then the block is split, leaving the remainder in the same place as the original (as usual with splitting). If such a block is not available, then a block is selected with the best fit policy.

Table 6.4 shows the results of the experiments with clustering. Half fit seems to be the winner among the clustering allocators. The average performance is still not as good as best fit (with any free list order) or first fit (address-ordered or FIFO).

Application	LIFO Next Fit		FIFO Next Fit	
	w/o WPH	w/ WPH	w/o WPH	w/ WPH
Espresso	13.93	18.48	4.81	4.81
GCC	34.12	31.19	8.43	8.43
Ghostscript	212.72	210.56	52.40	47.36
Grobner	101.30	70.97	84.76	57.18
Hyper	0.16	0.16	0.16	0.16
LRUsim	11.27	11.27	11.27	11.27
P2C	25.19	20.10	22.14	19.08
Perl	21.56	15.77	21.56	15.77
Average	52.53	51.82	25.69	20.51

Table 6.5: Fragmentation Results for Next Fit with Wilderness Preservation

6.4 Wilderness Preservation

Since our implementations of first fit and best fit use wilderness preservation by default, the experiments with this feature are limited to the next fit allocator. We ran simulations with two variants of next fit—LIFO and FIFO ordered free lists—with wilderness preservation. The results are summarized in the table 6.5.

Clearly, wilderness preservation reduced fragmentation on average, especially with the FIFO allocator. However, fragmentation was increased in the case of Espresso with the LIFO allocator. There is always this possibility with next fit due to the unpredictable interaction between the allocator and the program's allocation behavior (other allocators may also interact badly with applications, but next fit is especially unstable).

6.5 Splitting Thresholds

The experiments with splitting thresholds were conducted with all three of the basic sequential fits allocators. In order to get some idea of the effect of varying the value of the splitting threshold, we ran two sets of experiments with different values, and compared them with the results with no splitting threshold (always

Application	LIFO First Fit			LIFO Next Fit			LIFO Best Fit		
	0%	7%	14%	0%	7%	14%	0%	7%	14%
Espresso	13.93	16.96	4.81	13.93	13.93	16.96	4.81	6.33	15.45
GCC	31.01	31.70	31.70	34.12	34.81	34.98	8.26	8.95	8.95
Ghostscript	186.78	187.87	190.03	212.72	212.36	213.09	5.92	5.20	6.28
Grobner	120.60	117.85	126.12	101.30	98.54	101.30	54.42	54.42	54.42
Hyper	0.16	0.74	0.74	0.16	0.74	0.74	0.16	0.74	0.74
LRUsim	11.27	11.27	11.27	11.27	11.27	11.27	11.27	11.27	11.27
P2C	16.03	16.03	16.03	25.19	25.19	25.19	15.01	15.01	15.01
Perl	15.77	15.77	15.77	21.56	21.56	21.56	15.77	15.77	15.77
Average	49.44	49.77	49.56	52.53	52.30	53.14	14.45	14.71	15.99

Table 6.6: Fragmentation Results for Sequential Fits Allocators with Splitting Thresholds

split). The results in table 6.6 show that in general, a splitting threshold of 7% or 14% causes more fragmentation than no splitting threshold, though the difference is not generally statistically significant. In many cases, there was little or no difference in fragmentation with or without splitting thresholds.

6.6 Synthetic Traces vs. Real Workloads

We found that simulations with randomized request streams yielded higher average fragmentation than the simulation with the original trace. Clearly the order of the request stream is a critical piece of data in the allocator's choice of placement. Also, the relative performance of allocators is different between the original and randomized traces. Some allocators are helped by the randomization, but most yield higher fragmentation. Thus the effects of randomization lead to both quantitative and qualitative errors.

We used the trace shuffling tools described in Chapter 5 to generate synthetic traces with the same size and lifetime distributions as the original trace. We then ran simulations using the same allocators with the original traces and with the randomized traces. The results show that many allocators perform differently on the two sets of traces. We found that the correlation between fragmentation on the original traces and fragmentation on the randomized traces was 0.50. This means that 25% of the variation in the randomized data is due to variation in the original data. Some of this proportion can be explained by the fact that some allocators perform almost equally well in both sets of experiments, while others perform quite differently.

The table below shows the difference in allocator performance with the original traces and with the randomized traces. The list is sorted on the "Frag" field, separately for the original traces and the randomized traces. The "Waste" field is the percentage of wasted memory at maximum heap usage given one word of overhead for every object, except for the simple segregated store allocators, which never use any overhead per object (see definition of fragmentation in section 2.8).

Another interesting result in the randomized data is that FIFO best fit has 40% more fragmentation than with the original trace and address-ordered first fit has 53% more fragmentation. In fact there is a 82% increase in the average fragmentation over all the allocators with the randomized data.

Some notable exceptions are the simple segregated storage allocators, which both have *less* fragmentation with the randomized traces. The variation with 2^n sized blocks has 15% less fragmentation and the variation with $3 * 2^n$ sized blocks has 13% less fragmentation, lower than next fit and binary buddies. These results are misleading since simple segregated storage has much greater fragmentation with the original traces. We believe that shuffling traces helps allocators that don't deal well with splitting and coalescing, or with strong phase behavior in programs [25].

In general, shuffled traces cause more "holes" in the heap, but also smooth the distribution of objects in the heap so that the importance of coalescing is reduced. A shuffled trace is less likely to exhibit the pattern of allocation that hurts simple segregated storage—freeing many objects of one size and allocating many objects of another size [25]. When a page of memory is set aside for one size, it cannot be reclaimed or reused for other sizes. If the objects of that size are mostly freed, then that memory is wasted. This effect can accumulate if the different phases of an application use mutually exclusive sizes of objects.

The randomization of the request stream is essentially a "random walk" that smoothes out the peaks and phase behavior in the original trace. The inherent clustering of sizes in the heap is lost and contiguous objects are no more likely to be freed at the same time than noncontiguous ones.

Note the change in positions of the simple segregated storage allocators between the original and randomized traces. Both allocators jump up past several allocators. including binary buddies. The $3 * 2^n$ sized blocks variation jumps past binary buddies and both allocators jump past several of the next fit and first fit allocators. This instability leaves casts serious doubt on the validity of allocator performance measurements based on randomized traces.

Allocator Naming Convention. The naming scheme we used in the table is intended to specify the important parameters. The first part is typically the type of allocator (e.g., first fit, simple-seg, multifit) and the remaining parts specify the parameters that are specific to that allocator. In the case of the sequential fits allocators, the second parameter is the list order, followed by "def" (deferred coalescing) and the quick list order, or in some cases a splitting threshold (split7 or split14) or "wph" (wilderness preservation heuristic). "Binbud" is a binary buddies allocator and "Dblbdy" is a double buddy allocator (with a parameter for the SBRK unit). "Lea" refers to the segregated fits allocator written by Doug Lea (version 2.5.1).

Original			Randomized		
Allocator	Frag	Waste	Allocator	Frag	Waste
best-ao-4k	14.36	22.49	first-ao-4k-def-lifo	18.60	26.13
best-fifo	14.36	22.44	best-lifo-def-fifo	19.11	25.08
first-ao-4k	14.41	22.14	best-lifo-def-lifo	19.14	24.69
first-ao-8k	14.41	24.08	first-ao-4k-def-ao	19.18	26.71
best-lifo	14.45	22.44	first-ao-4k-def-fifo	19.18	25.96
first-ao-4k-def-ao	14.60	21.87	best-lifo-def-ao	19.33	24.89
first-fifo	14.70	22.82	halffit	19.74	26.27
best-lifo-split7	14.71	23.53	best-fifo	20.18	25.22
first-ao-4k-def-lifo	15.32	22.71	best-lifo	20.23	25.41
best-lifo-split14	15.99	23.41	best-ao-4k	20.34	25.80
first-ao-4k-def-fifo	16.03	24.00	best-lifo-split7	20.96	25.56
best-lifo-def-ao	16.07	23.24	best-lifo-split14	20.97	27.00
best-lifo-def-fifo	16.07	23.96	best-ao-8k	21.73	27.38
best-lifo-def-lifo	16.07	23.96	first-ao-4k	22.04	28.12
halffit	16.47	22.44	first-ao-8k	24.26	29.58
best-ao-8k	17.38	24.08	first-fifo	33.41	41.05
multfit-min	17.85	25.92	multfit-min	34.14	43.62
next-ao-4k	19.49	27.58	Lea 2.5.1	46.36	48.91
next-ao-8k	20.30	28.91	dblbdy-10k	49.32	56.74
next-fifo-wph	20.51	28.49	next-fifo-wph	50.51	57.28
multfit-max	22.26	29.02	dblbdy-5k	53.58	57.56
next-fifo	25.69	32.60	next-ao-4k	53.84	59.70
Lea-2.5.1	26.20	31.07	next-ao-8k	55.37	64.12
dblbdy-10k	45.14	51.23	next-fifo	59.27	65.90
dblbdy-5k	48.77	55.95	simpseg $3 * 2^N$	62.59	63.47
first-lifo	49.44	54.60	binbud	69.24	79.36
next-lifo-wph	49.48	51.67	multfit-max	72.38	93.58
first-lifo-split14	49.56	54.73	simpseg 2^N	72.76	72.29
first-lifo-split7	49.77	54.96	first-lifo	95.31	104.83
next-lifo-split7	52.30	62.74	first-lifo-split14	95.59	105.51
next-lifo	52.53	58.68	first-lifo-split7	95.82	104.59
next-lifo-split14	53.14	59.53	next-lifo-wph	106.13	113.20
binbud	62.29	74.01	next-lifo	115.48	126.35
simpseg $3 * 2^N$	72.95	73.39	next-lifo-split14	118.40	126.94
simpseg 2^N	85.00	85.62	next-lifo-split7	141.09	126.81

Table 6.7: Allocator Performance with Original vs. Shuffled Traces

Allocator	espresso	gcc	ghost	grobner	lindsay	perl	p2c	LRUsim	Avg	S. D.
LIFO best fit	4.81	8.26	5.92	54.42	0.16	15.77	15.01	11.27	14.45	16.98
LIFO first fit	13.93	31.01	186.78	120.60	0.16	15.77	16.03	11.27	49.44	67.26
LIFO next fit	13.93	34.12	212.72	101.30	0.16	21.56	25.19	11.27	52.53	71.74
binary buddy	45.96	8.26	38.38	71.21	99.94	62.59	58.87	77.94	62.29	20.52
double buddy	107.27	48.16	21.58	56.36	50.06	37.19	35.83	33.71	48.77	26.05
simp-seg 2^n	169.06	100.16	40.56	98.90	26.54	80.19	84.36	80.27	85.00	42.98
simp-seg $3 * 2^n$	182.16	83.20	28.53	84.07	26.66	89.57	52.42	36.99	72.95	51.08
Lea	13.93	28.43	47.72	65.45	0.16	21.56	21.12	11.27	26.20	21.07
average	68.88	42.70	72.77	81.54	25.48	43.03	38.60	34.25		
std dev	73.71	33.29	79.71	23.78	35.29	30.17	24.65	29.63		

T-test for fragmentation results (percent probability):

Allocator	best fit	LIFO first fit	next fit	bin bud	dbl bud	simple-seg 2^n	simple-seg $3 * 2^n$
LIFO best fit							
LIFO first fit	0.08						
LIFO next fit	0.08	0.25					
binary buddy	0.00	0.39	0.43				
double buddy	0.01	0.49	0.45	0.27			
simple-seg 2^n	0.00	0.15	0.18	0.12	0.00		
simple-seg $3 * 2^n$	0.01	0.25	0.29	0.27	0.03	0.06	
Lea	0.02	0.12	0.12	0.03	0.06	0.00	0.02

Table 6.8: Unshuffled Fragmentation and T-Tests

6.7 Statistical Significance of the Results

As with any empirical study, there is some question about the amount of difference in the performance of an allocator that should be considered significant. One problem is that there is not enough application trace data to make the results definitively representative. Still, we believe that our results are more representative than any previous study. Furthermore, our allocators are described in enough detail to understand the results of small changes in mechanisms. We will also offer our data and allocators for public access so that the results can be reproduced. Still, we can use a statistical analysis to aid in the interpretation of the data. Below is a table of fragmentation data for selected allocators and a table of t-test results that show how much confidence we can attach to the fragmentation data.

The second part of tables 7.1 and 7.2 list the percentage probabilities that the difference in fragmentation between two allocators is due to chance. These tables show that the fragmentation results are statistically significant.

Allocators	espresso	gcc	ghost	grobner	lindsay	perl	p2c	LRUsim	Avg	S. D.
LIFO best fit	11.32	23.24	9.93	63.74	0.14	20.18	20.99	12.33	20.23	19.10
LIFO first fit	68.53	65.59	291.37	247.53	0.34	35.86	40.94	12.33	95.31	110.59
LIFO next fit	66.99	94.25	314.47	324.39	0.34	51.53	59.55	12.33	115.48	129.37
binary buddy	48.57	23.24	47.18	77.40	100.30	67.67	70.31	79.77	69.24	17.31
double buddy	120.26	57.29	31.29	62.74	50.33	40.83	31.90	35.23	53.58	29.32
simp-seg 2^n	96.56	78.69	48.48	93.81	26.57	78.33	74.35	80.71	72.76	23.81
simp-seg $3*2^n$	139.27	55.00	35.48	82.95	26.69	86.80	43.26	38.35	62.59	37.14
Lea	87.09	79.28	56.55	83.79	0.14	25.41	26.31	12.33	46.36	34.61
average	79.82	59.57	104.34	129.54	25.61	50.83	45.95	35.42		
std dev	40.47	25.80	123.52	99.24	35.39	24.62	20.08	29.66		

T-test for fragmentation results (percent probability):

Allocator	best fit	LIFO first fit	next fit	bin bud	dbl bud	simple-seg 2^n	simple-seg $3*2^n$
LIFO best fit							
LIFO first fit	0.04						
LIFO next fit	0.03	0.03					
binary buddy	0.00	0.24	0.16				
double buddy	0.01	0.17	0.12	0.25			
simple-seg 2^n	0.00	0.29	0.19	0.29	0.05		
simple-seg $3*2^n$	0.01	0.23	0.15	0.48	0.11	0.19	
Lea	2.01	9.49	6.29	18.97	24.14	1.70	8.29

Table 6.9: Shuffled Fragmentation and T-Tests

6.8 Actual Fragmentation

Since our simulations tools ordinarily compensate for object overhead introduced by the allocator, the only other sources of memory waste due to the allocator are memory alignment requirements and minimum object size. Based on the observation that the traces that caused the most fragmentation (i.e., ghostscript, grobner) usually used a relatively large number of very small objects (8 bytes or less), we made the hypothesis that the wasted space that comes from rounding these small objects up to the minimum block size may be a significant source of wasted memory. To test this hypothesis, we are developing a modified simulator that eliminates these requirements altogether. Preliminary results of experiments in progress suggest that good allocators, such as best fit or address-ordered first fit, yield very low fragmentation (less than 5%) on any of the applications we tested when the alignment and minimum object requirements are removed. Results of these experiments will be presented in [9].

Chapter 7

Conclusions and Future Work

We made a point of separating the policy and the mechanism associated with an allocator. We believe that for almost any known allocator policy, an efficient mechanism can be found to implement it [25].

Our measure of fragmentation is based on the amount of wasted memory at the point of maximum memory usage. This is a useful measure in both real and virtual memory environments, because the maximum memory usage is the "bottleneck" for program performance. It is also independent of locality of reference. We also defined several variations of this basic measure in order to account for object overheads and alignment requirements.

7.1 Average Fragmentation for the Best Allocators

After compensating for header and footer overheads, we found that the average fragmentation for the best policies (i.e., non-LIFO first fits, best fits) was surprisingly low (around 14%).

Moreover, it's likely that these results still overestimate true fragmentation, because they don't account for minimum block sizes and alignment requirements. We believe that these overheads may account for the majority of the "fragmentation" with the best allocators.

This suggest that seeming minor implementation decisions are important, once a good policy has been found. Future works should address issues of per-object overheads, especially for small objects.

The results for best fit are especially useful since there is a simple, fast implementation [19]. While the best implementation of first fit may not scale as well, it is none the less interesting because it's worst-case fragmentation is much lower.

7.2 Basic Policies

Among the basic sequential fits allocators, we found that several allocators did almost identically well in terms of fragmentation: address- or FIFO-ordered first fit, and address-, LIFO-, and FIFO-ordered best fit. While these simple *policies* are based on obvious uses of a linear list, they can be implemented more efficiently by other mechanisms, and the effects of simple policy veriations are informative.

In general, allocators with address-ordered free lists did as well as or better than allocators with FIFO free lists, and dramatically better than those with LIFO-ordered free lists. LIFO lists incurred much higher fragmentation, but may have better locality of reference otherwise.

For best fit, list order made little difference because it is only used in breaking ties between multiple blocks of equally good fit.

For first fit and next fit, list order is crucial. LIFO lists increase fragmentation, but FIFO lists do surprisingly well.

For first fit, FIFO lists work about as well as address-ordered lists. This may be useful, because a FIFO ordering is cheaper to maintain than an address ordering. This surprising result is also significant in that it suggests that there may be *many* policies with similarly excellent performance; given that FIFO ordering and address-ordering are quite different, it seems that many other policies may do about as well, giving allocator designers freedom to design policies that satisfy other goals without unduly affecting fragmentation.

It is also interesting to speculate as to why these allocators do better than LIFO allocators; the main thing they seem to have in common is *deferred reuse*—freed blocks are typically not reused immediately, but are allowed to age. This may decrease fragmentation by allowing their neighbors time to die and be coalesced with them, rather than scattering objects among transient small holes. In contrast, LIFO allocators show not only high average fragmentation but high variance—they appear to be *unstable* because immediate reuse interacts very differently with different programs' request patterns.

While FIFO first fit seems to work well in terms of fragmentation, we suspect that it may decrease locality considerably relative to more LIFO-like reuse patterns; recently-used (and freed) memory may go unused for a very long time (rather than being reused quickly, while it is still likely to be cached in fast memory).

Next fit allocators performed poorly; we suspect that this is because the roving pointer tends to scatter related objects, and this causes fragmentation. (Related objects may die at about the same time, but if they are scattered among unrelated objects, they may cause fragmentation rather than avoiding it.) As with first fit, LIFO list ordering caused especially high fragmentation. Unlike first fit, FIFO ordering was significantly worse than address-ordering, but not nearly as bad as LIFO.

Buddy systems generally incur higher average internal fragmentation than the total fragmentation of the best allocators. Double buddies reduce internal fragmentation, as they were designed to, but the remaining fragmentation is still fairly high. Buddy systems may be viewed as a coarse approximation of—and poor substitute for—best fit allocation [25]. Since there are now efficient implementation techniques for best fit, including best fit with deferred coalescing to improve speed, buddy systems do not seem attractive for general-purpose allocation.

The simple segregated storage allocators are also subject to significant internal fragmentation, but they are especially susceptible to external fragmentation, because they do no coalescing. These simple allocators are fast, but use memory quite poorly on average. Such allocators are likely to waste considerable memory unless augmented with some form of coalescing.

7.3 Auxiliary Features Designed to Reduce Fragmentation

We considered a number of allocator features and designed a set of experiments to determine whether these features have any effect on the allocator's performance. Splitting thresholds (7% and 14%) cause an increase in average fragmentation. Wilderness preservation clearly decreases fragmentation with the next fit allocator, although next fit's fragmentation is still higher than best fit or first fit.

Among the clustering allocators, half fit yields the lowest fragmentation. We believe that the reason the clustering effects increase fragmentation may be that the allocators are already doing a good job of clustering, and adding the clustering effects on top of that defeat the inherent clustering effects of the allocator.

None of the features that were studied revealed any great improvements to the set of basic allocators that already have good performance. However, there are other measure's of allocator performance which may support the use of some of these features. In particular, locality of reference is likely to be better with a LIFO free list than with a FIFO free list. In a LIFO list, an object that is freed will be reused soon, thus concentrating memory access to that area. FIFO lists distribute memory access by reusing older objects. Another issue is speed. A segregated fits allocator combines the low fragmentation of the best fit policy with the higher speed of simple segregated storage, making it a highly attractive allocator for general use.

7.4 Deferred Coalescing

Our results show that, on average, deferred coalescing has no significant effect on fragmentation. It neither increased nor decreased fragmentation on most of the experiments we ran. The main benefit of deferred coalescing is higher performance in terms of speed.

The order of the quick lists in our implementation of deferred coalescing had little effect on the fragmentation. This is different than with the main free list in a sequential fits scheme, where list order often has dramatic effects.

Deferred coalescing with LIFO quick lists can be used with a FIFO free list allocator to support better locality of reference with no expected adverse effect on fragmentation. The issue of the locality associated with an allocator was not covered in this study, but will be covered in future work.

7.5 Methodology

Despite previous work in the study of memory allocation, there is still much that is not known. The complex interaction between program allocation behavior and allocation policy has been largely unexplored. We believe that we have made a contribution to the problem in a number of ways. We show that the synthetic trace methodology is systematically invalid due to multiple biases. Randomization creates fragmentation by killing objects at random, but can reduce fragmentation by eliminating phase behavior. It affects different allocators in different ways.

We are developing a methodology that is empirical because the nature of memory allocation is still not well enough understood to have an accurate analytical model. Our methodology includes the construction of parameterized allocators that allow details to be explicitly specified to avoid ambiguous results.

7.6 Future Work

There are still many aspects of memory allocation that are not well understood. The locality, speed, and fragmentation associated with allocators, and the relationship between these issues, deserve more study with a sound experimental methodology. Some experiments in progress by the Object Oriented Programming Systems Group are beginning to address these issues [1].

Although we believe we have presented a better methodology for studying memory allocation, we need to have more allocation trace data from representative applications to validate the results. Once this data is collected, it is fairly straightforward to simulate an allocator's performance with the trace using our simulation framework.

[1] For further information on the OOPS Group's research in programming languages and memory management (including allocators, garbage collectors, persistent object stores, and memory hierarchies) see the World Wide Web page at http://www.cs.utexas.edu/users/oops.

Appendix A

Complete Results of Experiments with Original Traces

Allocator	Espresso	Gcc	Ghostscript	Grobner	Lindsay
binbud	45.96	43.44	38.38	71.21	99.94
best-lifo	4.81	8.26	5.92	54.42	0.16
best-ao-4k	4.81	8.26	5.20	54.42	0.16
best-ao-8k	6.33	8.26	5.92	54.42	0.35
best-fifo	4.81	8.26	5.20	54.42	0.16
best-lifo-def-ao	4.81	24.98	5.92	51.67	0.16
best-lifo-def-fifo	4.81	24.98	5.92	51.67	0.16
best-lifo-def-lifo	4.81	24.98	5.92	51.67	0.16
best-lifo-nofeet	4.81	8.26	5.92	54.42	0.16
best-lifo-nofeet-align4	0.26	4.98	4.12	54.42	0.16
best-lifo-split14	15.45	8.95	6.28	54.42	0.74
best-lifo-split7	6.33	8.95	5.20	54.42	0.74
dblbdy-10k	74.99	46.74	20.23	57.74	50.16
dblbdy-5k	107.27	48.16	21.58	56.36	50.06
first-lifo	13.93	31.01	186.78	120.60	0.16
first-ao-4k	4.81	8.26	5.56	54.42	0.16
first-ao-4k-nofeet	4.81	8.26	5.56	54.42	0.16
first-ao-8k	4.81	8.26	5.56	54.42	0.16
first-lifo-def-ao	4.81	13.60	5.56	51.67	0.16
first-lifo-def-fifo	4.81	24.98	5.56	51.67	0.16
first-lifo-def-lifo	4.81	19.29	5.56	51.67	0.16
first-fifo	4.81	8.43	7.72	54.42	0.16
first-lifo-split14	4.81	31.70	190.03	126.12	0.74
first-lifo-split7	16.96	31.70	187.87	117.85	0.74
Lea	13.93	28.43	47.72	65.45	0.16
halffit	4.81	25.15	5.92	51.67	0.16
multfit-max	13.93	37.39	15.29	68.21	0.16
multfit-min	4.81	24.98	8.81	59.94	0.16
next-lifo	13.93	34.12	212.72	101.30	0.16
next-ao-4k	4.81	8.43	19.61	70.97	0.16
next-ao-8k	6.33	8.60	17.45	70.97	0.35
next-fifo	4.81	8.43	52.40	84.76	0.16
next-fifo-wph	4.81	8.43	47.36	57.18	0.16
next-lifo-split14	16.96	34.98	213.09	101.30	0.74
next-lifo-split7	13.93	34.81	212.36	98.54	0.74
next-lifo-wph	18.48	31.19	210.56	70.97	0.16
simple-seg 2^N	169.06	100.16	40.56	98.90	26.54
simple-seg $3 * 2^N$	182.16	83.20	28.53	84.07	26.66
Average	21.28	24.21	48.36	68.00	6.57
Std Dev	41.08	20.21	73.18	21.45	19.42

Table A.1: Fragmentation Results for Allocators and Applications—Part 1

Allocator	Perl	P2c	LRUsim	Average	Std Dev
binbud	62.59	58.87	77.94	62.29	20.52
best-lifo	15.77	15.01	11.27	14.45	16.98
best-ao-4k	15.77	15.01	11.27	14.36	17.03
best-ao-8k	27.35	16.03	11.27	17.38	18.50
best-fifo	15.77	15.01	11.27	14.36	17.03
best-lifo-def-ao	15.77	13.99	11.27	16.07	16.29
best-lifo-def-fifo	15.77	13.99	11.27	16.07	16.29
best-lifo-def-lifo	15.77	13.99	11.27	16.07	16.29
best-lifo-nofeet	15.77	15.01	11.27	14.45	16.98
best-lifo-nofeet-align4	15.77	9.92	0.26	11.24	18.29
best-lifo-split14	15.77	15.01	11.27	15.99	16.38
best-lifo-split7	15.77	15.01	11.27	14.71	16.81
dblbdy-10k	40.09	37.10	34.07	45.14	16.52
dblbdy-5k	37.19	35.83	33.71	48.77	26.05
first-lifo	15.77	16.03	11.27	49.44	67.26
first-ao-4k	15.77	15.01	11.27	14.41	17.01
first-ao-4k-nofeet	15.77	15.01	11.27	14.41	17.01
first-ao-8k	15.77	15.01	11.27	14.41	17.01
first-lifo-def-ao	15.77	13.99	11.27	14.60	15.92
first-lifo-def-fifo	15.77	13.99	11.27	16.03	16.32
first-lifo-def-lifo	15.77	13.99	11.27	15.32	15.99
first-fifo	15.77	15.01	11.27	14.70	16.85
first-lifo-split14	15.77	16.03	11.27	49.56	69.72
first-lifo-split7	15.77	16.03	11.27	49.77	66.86
Lea	21.56	21.12	11.27	26.20	21.07
Lea nofeet	21.56	21.12	11.27	26.20	21.07
halffit	15.77	17.05	11.27	16.47	16.28
multfit-max	15.77	16.03	11.27	22.26	21.21
multfit-min	15.77	17.05	11.27	17.85	18.65
next-lifo	21.56	25.19	11.27	52.53	71.74
next-ao-4k	21.56	19.08	11.27	19.49	22.15
next-ao-8k	27.35	20.10	11.27	20.30	22.16
next-fifo	21.56	22.14	11.27	25.69	28.85
next-fifo-wph	15.77	19.08	11.27	20.51	20.64
next-lifo-split14	21.56	25.19	11.27	53.14	71.53
next-lifo-split7	21.56	25.19	11.27	52.30	71.27
next-lifo-wph	15.77	20.10	11.27	51.82	73.51
simple-seg 2^N	80.19	84.36	80.27	85.00	42.98
simple-seg $3 * 2^N$	89.57	52.42	36.99	72.95	51.08
Average	23.13	21.56	16.16		
Std Dev	16.93	14.24	16.06		

Table A.2: Fragmentation Results for Allocators and Applications—Part 2

Bibliography

[1] David A. Barrett and Bejamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation* [15], pages 187–196.

[2] David A. Barrett and Benjamin G. Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation*, pages 301–314, La Jolla, California, June 1995. ACM Press.

[3] A. P. Batson and R. E. Brundage. Segment sizes and lifetimes in ALGOL 60 programs. *Communications of the ACM*, 20(1):36–44, January 1977.

[4] A. P. Batson, S. M. Ju, and D. C. Wood. Measurements of segment size. *Communications of the ACM*, 13(3):155–159, March 1970.

[5] J. S. Fenton and D. W. Payne. Dynamic storage allocations of arbitrary sized segments. In *Proc. IFIPS*, pages 344–348, 1974.

[6] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. In *Fourth Annual ACM Symposium on the Theory of Computing*, 1972.

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

[8] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation* [15], pages 177–186.

[9] Mark Johnstone. Memory allocation and garbage collection. Work in progress, May 1996.

[10] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973. First edition published in 1968.

[11] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *Proc. USENIX Summer 1985*, pages 489–506, Portland, Oregon, June 1985. USENIX Association.

[12] B. W. Leverett and P. G. Hibbard. An adaptive system for dynamic storage allocation. *Software Practice and Experience*, 12(6):543–556, June 1982.

[13] B. H. Margolin, R. P. Parmelee, and M. Schatzoff. Analysis of free-storage algorithms. *IBM Systems Journal*, 10(4):283–304, 1971.

[14] J. Minker et al. Analysis of data processing systems. Technical Report 69-99, University of Maryland, College Park, Maryland, 1969.

[15] *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993. ACM Press.

[16] P. W. Purdom, S. M. Stigler, and Tat-Ong Cheam. Statistical investigation of three storage allocation algorithms. *BIT*, 11:187–195, 1971.

[17] Brian Randell. A note on storage fragmentation and program segmentation. *Communications of the ACM*, 12(7):365–372, July 1969.

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

[19] Thomas Standish. *Data Structure Techniques*. Addison-Wesley, Reading, Massachusetts, 1980.

[20] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 30–32, Bretton Woods, New Hampshire, October 1983. ACM Press. Published as *Operating Systems Review 17*(5), October 1983.

[21] R. A. Totschek. An empirical investigation into the behavior of the SDC time-sharing system. Technical Report SP2191, Systems Development Corporation, 1965.

[22] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 1995. To appear.

[23] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[24] Paul R. Wilson. Garbage collection. *Computing Surveys*, 1995. Expanded version of [23]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps`. In revision, to appear.

[25] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Work-*

shop on Memory Management, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[26] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):107–131, 1994.

Vita

Michael Shannon Neely is a graduate student at the University of Texas at Austin.

Permanent Address: Dept. of Computer Science
The University of Texas at Austin

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.