Non-Compacting Memory Allocation and Real-Time Garbage Collection

by

Mark Stuart Johnstone, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1997

Non-Compacting Memory Allocation and Real-Time Garbage Collection

Approved by
Dissertation Committee:

_____

_____

_____

_____

_____

To Maow

Acknowledgments

This dissertation could not have been completed if not for the help and support of many people. First and foremost I would like to acknowledge my advisor Paul Wilson. It was his extraordinarily deep understanding of the issues studied for this dissertation that kept me looking at the right problems. I would also like to thank the past and present members of the OOPS research group at the University of Texas at Austin who worked on parts of this research: David Boles, Sheetal Kakkad, Donavan Kolbly, Mike Neely, and Jun Sawada. I would like to thank the Motorola Corporation for its financial support during the last year and a half of this research, and my coworkers and supervisors at the Somerset Design Center in Austin, Texas, who were more than understanding when the demands of completing a Ph.D. were at odds with my regular job duties. Finally, I would like to thank Patricia Burson for her time spent proofreading this dissertation, and her incredible patience and support throughout the process.

Mark Stuart Johnstone

The University of Texas at Austin

December 1997

Preface

Dynamic memory management is a very important feature of modern programming languages, and one that is often taken for granted. Programmers frequently place great demands on the memory management facilities of their language, and expect the language to efficiently handle these requests. Unfortunately, memory management systems are not always up to the task. The article which appears below strikingly illustrates how problems with a program's dynamic memory management can cause disastrous results, sometimes years after the program is written. Memory errors like this one are very difficult to prevent, and it is a certainty that they will occur again and again.

It is our hope that the results presented in this research will lead to a better understanding of the nature of memory management problems, and to improved implementations of memory management systems. We believe that improved memory management systems will ultimately lead to more robust software, and problems like the one presented in the following article will become a rare exception rather than the rule.

Why Bre-X Crashed the TSE

By Geoffrey Rowan

Toronto Globe and Mail, 12 April 1997

A software flaw that lay sleeping for 20 years inside the Toronto Stock Exchange's computers woke up mean last week, shutting down the automated trading system repeatedly before technicians could identify it.

The flaw might have passed harmlessly out of existence, since the TSE is replacing its system in a few months, but for the controversy that erupted around Bre-X Minerals Ltd.

The exchange's problems with its dog-eared computer system offer a lesson to other organizations that are patching together mature technology to keep their critical business systems running: It's hard to know exactly what's inside such systems or to know when some hidden glitch might wreak havoc.

The events: The TSE's problems started on March 27, after Calgary-based gold mining company Bre-X reported that there might not be as much gold in its highly touted Busang field as investors had been led to believe.

That triggered a frenzy of trading in Bre-X, which by itself shouldn't have been a problem. The TSE is Canada's largest stock exchange–it can handle a lot of trading and even big increases in volume.

In 1996, the TSE saw a huge increase in share volumes, to 23.2 billion shares traded from 15.8 billion a year earlier. But it had never seen the kind of volume in a single stock that occurred with Bre-X.

The exchange refers to the number of active buy-and-sell orders for a particular stock at any point as the "book." The average book size is about 200 to 300 orders. The Bre-X book size last Thursday–when the exchange first ran into trouble–was 2,500, and it would swell at times to 4,500. Prior to that, the largest

book size ever was about 1,600 orders, which happened once in the late eighties.

With all those Bre-X trades waiting to be executed, the TSE's Computerized Automated Trading System simply ground to a halt. When brokers entered their orders, nothing happened. It was frozen.

Not knowing what the problem was, TSE technicians restarted the system at about 3:40 p.m., but within about eight minutes it crashed again. Just 12 minutes away from the end of the trading day, TSE officials decided not to try to bring it back up again.

Friday was a holiday, giving the technicians three solid days to search through the system, which is essentially three million lines of computer code running on powerful fault-tolerant computers made by Tandem Computers Inc. of Cupertino, Calif.[1]

Working 24 hours a day, they poured over the old code, which was poorly documented because it had been written so long ago. It's had many refinements made to it over the years, and documentation methodology wasn't as stringent two decades ago as it is today.

The technicians concluded that what they had was a memory problem.

When an order is to be executed, the computer's code moves the entire order book for a stock into its active memory. Once that order has been executed, that piece of memory is released, to be reused by the next order book coming in.

With sequential orders for execution on the same book, the entire Bre-X book was being loaded into memory for every order, requiring continuous availability of enough memory to hold the larger-than-usual Bre-X order book.

This past weekend, the TSE technicians expanded the system's memory and

[1]** CORRECTION ** The Toronto Stock Exchange's Computerized Automatic Trading System, which has suffered software problems in recent days, runs on an IBM mainframe, not a Tandem computer. The TSE system is being upgraded and will be moved from IBM hardware to Tandem hardware later this year or early next year. Incorrect information, supplied by the TSE, appeared [on] April 4.

on Monday, the exchange was opened for business, but Bre-X trading was halted until Tuesday.

That day, the system stayed up for about 23 minutes, and in that time, it executed a greater number of Bre-X orders than the other Canadian exchanges did all day, combined. The problem wasn't memory, but it was obviously related to the Bre-X trading volume.

After the Tuesday morning crash, TSE officials decided to reopen the market without reopening trading in Bre-X, and the system was working, though several attempts to restart Bre-X trading have had to be carefully monitored.

Whenever Bre-X volume starts to threaten the system, Bre-X trading is shut down–as happened again yesterday.

The challenge ahead: What technicians are focused on now is a chunk of the TSE's digital code associated with cancelled orders. When an order is executed, the memory that holds the book is released, but when an order is cancelled, the memory is not released. "That piece of code was not written the way it should have been," TSE president Rowland Fleming said. "The problem was buried for 20 years. It has been a sleeping problem." It never surfaced before because the order books were never big enough, and trading in a single issue was never volatile enough that cancelled orders would sink the system.

Mr. Fleming said TSE technicians won't try for an overnight fix.

They'll work on the problem through the weekend and if they can't write a fix in that time, they'll try to figure out a way to work around the cancellation function or to restrict its use.

"At this stage, we think that is the cause of our problem and we'll get the fix," Mr. Fleming said.

Then the exchange just has to hang on until the end of the year or early next year, when its new computer system is scheduled to go on-line.

Non-Compacting Memory Allocation and Real-Time Garbage Collection

Publication No. _____

Mark Stuart Johnstone, Ph.D.
The University of Texas at Austin, 1997

Supervisor: Paul R. Wilson

Dynamic memory use has been widely recognized to have profound effects on program performance, and has been the topic of many research studies over the last forty years. In spite of years of research, there is considerable confusion about the effects of dynamic memory allocation. Worse, this confusion is often unrecognized, and memory allocators are widely thought to be fairly well understood.

In this research, we attempt to clarify many issues for both manual and automatic *non-moving* memory management. We show that the traditional approaches to studying dynamic memory allocation are unsound, and develop a sound methodology for studying this problem. We present experimental evidence that fragmentation costs are much lower than previously recognized for most programs, and develop a framework for understanding these results and enabling further research in this area. For a large class of programs using well-known allocation policies, we show that fragmentation costs are *near zero*. We also study the locality effects of memory allocation on programs, a research area that has been almost completely ignored. We show that these effects can be quite dramatic, and that the best alloca-

tion policies in terms of fragmentation are also among the best in terms of locality at both the cache and virtual memory levels of the memory hierarchy.

We extend these fragmentation and locality results to real-time garbage collection. We have developed a hard real-time, non-copying generational garbage collector which uses a *write-barrier* to coordinate collection work only with *modifications* of pointers, therefore making coordination costs cheaper and more predictable than previous approaches. We combine this write-barrier approach with *implicit non-copying reclamation*, which has most of the advantages of copying collection (notably avoiding both the sweep phase required by mark-sweep collectors, and the referencing of garbage objects when reclaiming their space), without the disadvantage of having to actually copy the objects. In addition, we present a *model* for non-copying implicit-reclamation garbage collection. We use this model to compare and contrast our work with that of others, and to discuss the tradeoffs that must be made when developing such a garbage collector.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Memory management is poorly understood. This research attempts to clarify the issues pertaining to memory management in general, including the effects of fragmentation and locality, for both manual and automatic (i.e., garbage collected) memory management. In doing this, we explore and clarify the basic design issues of allocators, revealing important new insights that have gone overlooked for almost thirty years. We also explore the effect of dynamic memory allocation on locality of reference at both the cache and virtual memory level. In addition, we explore the basic design issues in incremental and real-time garbage collectors, putting them on a sounder footing. Finally, we clarify the performance issues of both copying and non-copying real-time garbage collection.

1.1 Scope of this Dissertation

The overall goals of this dissertation are:

1. to carefully explore the basic design issues in memory allocators and incremental garbage collectors, putting these issues on a sounder footing;

2. to demonstrate that for most programs, fragmentation costs are very close to zero;

3. to explore in detail the effects of memory allocator policy on locality at both the cache and virtual memory levels;

4. to provide a design and implementation for a garbage collector that fulfills all of the requirements for use with a real-time system; and

5. to provide a model for identifying the performance issues in both copying and non-copying real-time garbage collection.

In the remainder of this chapter, we present an overview of our work, and present much of the background material for this area of research, including defining our important terms. A reader already familiar with this research area may wish to skip ahead to the next chapter.

1.2 Memory Allocation

All modern programming languages allow the programmer to use *dynamic memory allocation*. Dynamic memory allocation is the ability to allocate and deallocate memory at run time (dynamically), and comes in two flavors: manual and automatic (garbage collected). Both forms allow the programmer to specify the memory needs of the program at run time by explicitly requesting memory blocks from the programming language. Manual memory management, used in C, C++, Pascal, Ada, and Modula II, requires the programmer to explicitly return memory to the language when it is no longer needed. Automatic memory management, used in LISP, Scheme, Eiffel, Modula III, and Java, frees the programmer from this burden; memory is automatically reclaimed when the run-time system can determine that it can no longer be referenced.

Manual and automatic memory allocation routines have more in common than is generally appreciated. Both kinds of memory management are essentially on-line algorithms, and must choose where to allocate objects in memory based only on information available to the point of allocation. Once these objects are placed, a manual memory management system typically cannot later move these objects if a placement choice turns out to be a bad one. Garbage collected systems, on the other hand, often do have the freedom to later move blocks of memory if necessary. However, as we will discuss in Chapter 4, a garbage collector that does not move memory has many advantages for real-time use over its moving counterpart.

Problems in memory management are due to three factors: programmers' lack of understanding of the cost of dynamic memory management, language implementors' lack of understanding of the issues involved with the design and implementation of memory management systems, and fundamental algorithmic properties of applications that are extraordinarily difficult to correctly implement with manual memory management. In this research, we address the first two problems by clarifying many of the important issues in memory management, and demonstrating that good algorithms can keep memory management costs quite low, thus making the first problem less of a concern. We address the third problem by clarifying many of the issues in automatic memory management (garbage collection). We show how our results for manual memory management algorithms are directly applicable to garbage collected systems, and implement a real-time garbage collector to test these ideas.

1.2.1 Fragmentation

Manual memory management systems and non-moving garbage collected systems use the same choices in the same conditions when deciding on where to allocate requested objects. Because these allocators cannot later move memory, an important

issue is *fragmentation*. Fragmentation is said to be present when sufficient free memory is available, but is unusable because it exists as many small *fragments* of memory rather than as one large block. Traditionally, fragmentation is classified as *external* or *internal* [RK68], and is combatted by splitting and coalescing free blocks.

External fragmentation arises when free blocks of memory are available for allocation, but cannot be used to hold objects of the sizes actually requested by a program. In sophisticated allocators, this is usually because the free blocks are too small, and the program requests larger objects. In some simple allocators, external fragmentation can occur because the allocator is unwilling or unable to split large blocks into smaller ones.

Internal fragmentation arises when a sufficiently large free block is allocated to hold an object, but there is a poor fit because the block is larger than needed. In some allocators, the remainder is simply wasted, causing internal fragmentation. (It is called *internal* because the wasted memory is inside an allocated block, rather than being recorded as a free block in its own right.)

To combat internal fragmentation, most allocators will *split* blocks into multiple parts, allocating part of a block, and then regarding the remainder as a smaller free block in its own right. Many allocators will also *coalesce* adjacent free blocks (i.e., neighboring free blocks in address order), combining them into larger blocks that can be used to satisfy requests for larger objects.

In some allocators, internal fragmentation arises due to implementation constraints within the allocator — for speed or simplicity reasons, the allocator design restricts the ways memory may be subdivided. In other allocators, internal fragmentation may be accepted as part of a strategy to prevent external fragmentation — the allocator may be unwilling to fragment a block, because if it does, it may not be able to coalesce it again later and use it to hold another large object.

1.2.2 Strategy, Policy, and Mechanism

It is important to separate allocator design into three parts: *strategy*, *policy*, and *mechanism*. The basic approach to designing a memory allocator is the strategy. A *strategy* may be: "minimize waste for each allocation," or "sacrifice one area of memory to preserve other areas of memory." These strategies can be realized by many different *policies* for placing dynamically allocated objects. Some familiar policies are: "choose the smallest block that is large enough, breaking ties in Last In First Out (LIFO) order of object deallocation" (known as *LIFO best fit*), or "choose the first free block that large enough, looking from low heap address to high heap address" (known as *first fit address ordered*). These policies are then implemented by a set of *mechanisms*. An example of a mechanism is: "use a linked list, and search from the head of the list; freed blocks are inserted at the front of the list."

The distinction between policy and mechanism is an important one because different policies can be implemented by a variety of mechanisms. So, if a particular policy performs well, but the implementation of that policy has undesirable properties, one can design a different implementation of the same policy. For example, the obvious implementation of first fit address ordered is to maintain a sorted list of free blocks. However, this mechanism is prohibitively expensive. A different mechanism for implementing the same policy is to use a bit map indicating the free blocks, and scan the bit map for a suitable block at allocation time.

The distinction between strategy and policy is also an important one because different policies can have secondary effects, such as affecting the locality of reference of the program. If a particular policy produces low fragmentation, but also has poor locality of reference, then a different policy can be chosen that obeys the same basic strategy, but produces better locality. For example, the strategy "sacrifice one memory area to preserve other memory areas" can be realized by both the best-fit LIFO and the first-fit address-ordered policies.

1.2.3 Experimental Methodology

In surveying the allocation literature we discovered that virtually all past work in this field suffered from one common flaw: almost no one measured how well different allocation policies performed for *actual* programs.[1] In this research, we present results gathered by studying eight large C and C++ programs. Our results show that for these eight programs, fragmentation can be kept very near zero. We argue that the strategy behind the allocation policies that work best is fundamentally strong, and will work well for most real programs.

We devote a large portion of this research to studying issues pertaining to memory fragmentation for non-moving memory allocators. In particular, we study the conditions under which allocators interact with programs to produce fragmentation, and the conditions under which they do not. We also address experimental methodology for studying memory allocation design and point out flaws in traditional methodologies that have been used for at least 30 years.

1.3 Locality

Locality of reference is the property that programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy which instructions and data a program will use in the near future based on its accesses in the recent past [PH96].

There are two fundamental kinds of locality: *spatial* locality and *temporal* locality. Spatial locality is the property that data and instructions whose addresses are near one another tend to be referenced close together in time. Temporal locality

[1]The studies by Zorn [DDZ93, ZG92] and by Vo [Vo95] were the only work we found that used actual programs in their studies. They have made these programs, many of which we used, available by anonymous ftp. We will do the same with the additional programs we used.

is the property that programs tend to access data and instructions that have been accessed in the recent past.

Most modern computer systems are built using a *memory hierarchy*, that is a primary cache, secondary cache, main memory, and disk based paging area, with each level being larger, slower, and cheaper than the previous. If a memory reference at one level fails, then that reference is attempted at the next level. For such computers, locality of reference is very important. The current trend in microprocessor design is for processors to increase in speed much more quickly than the memory systems that support them. Thus, good locality of reference will become increasingly important in order to take full advantage of available computer hardware. Surprisingly, researchers have virtually ignored one of the most important effects on a program's locality of reference: that of the dynamic memory allocator's placement choices.[2]

Grunwald, Zorn, and Henderson [GZH93] show that different allocators can have an important effect on the locality of the programs that use them. However, they failed to separate the locality effects of the allocation *policy* from those of the particular *mechanism*. Thus, for the memory allocation policies that fared worst, they could not be sure if it was because the policy itself has inherently poor locality, or because their implementation of the policy has poor locality. We remedy this problem by carefully filtering out all the locality effects of the memory allocator implementation, and varying the policy decisions so that we can measure the individual effects of these policy decisions on the locality of reference of the application. We show that the best allocation policies in terms of fragmentation are also among the best in terms of locality.

[2]While there has been some work on the locality of reference of memory allocators that can move memory (such as garbage collectors) [WLM90, WLM92, Zor91, PS89, JLS92, Nut87, Ber88], [GZH93] was the only paper on the topic of locality and *non-moving* memory allocation that we were able to locate. The authors of this paper also found it surprising that no one had done work in this area before.

1.4 Garbage Collection

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again. Such data objects are referred to as *garbage*. The basic functioning of a garbage collector consists, abstractly speaking, of two parts:

1. Distinguishing the live objects from the garbage in some way (*garbage detection*).

2. Reclaiming the garbage objects' storage so that the running program can use it again (*garbage reclamation*).

In practice, these two phases may be functionally or temporally interleaved.

In general, garbage collectors use a liveness criterion that is somewhat more conservative than the liveness criterion used by other systems. In an optimizing compiler, for example, a value may be considered dead at the point that it can never be used again by the running program, as determined by control or data flow analysis. A garbage collector, on the other hand, typically uses a simpler, less dynamic criterion of liveness, defined in terms of a *root set* and *reachability* from the roots.

At the moment the garbage collector is invoked, the active variables are considered live. Typically, this includes statically-allocated global or module variables, as well as local variables in activation records on the activation stack(s), and any variables currently in registers. These variables form the *root set* for the traversal. Heap objects directly reachable from any of these variables can be accessed by the running program, so they must be preserved. In addition, since the program might traverse pointers from those objects to reach other objects, any object reachable from a live object is also live. Thus the set of live objects is simply the transitive closure of all variables reachable from the root set.

Any object that is not reachable from the root set is garbage, i.e., useless, because there is no legal sequence of instructions that allow the program to reach that object. Garbage objects therefore cannot affect the future course of computation, and their space may be safely reclaimed.

There are two basic ways to reclaim garbage objects:

1. Find and reclaim all objects known to be garbage (*explicit* garbage reclamation).

2. Find and preserve all objects known to be live. All objects left over are garbage and can be reclaimed in one action (*implicit* garbage reclamation).

An example of explicit reclamation is *mark-sweep collection* [McC60]. In a mark-sweep collector, once the live objects have been distinguished from the garbage objects, memory is exhaustively examined (*swept*) to find all of the garbage objects and reclaim their space.

An example of implicit reclamation is *copying collection* [FY69, Che70]. In a copying collector, the live objects are copied out of one area of memory and into another. Once all live objects have been copied out of the original memory area, that entire area is considered to be garbage and can be reclaimed in one operation. The garbage objects are never examined, and their space is implicitly reclaimed.

While at first these two methods of reclaiming garbage memory may seem fundamentally different, there is a way to combine them to receive many of the advantages of both [Wan89, Bak91]. This "fake copying" approach is fundamental to our real-time garbage collector implementation.

1.4.1 Real-Time Garbage Collection

Real-time programs are usually characterized as being either *hard real-time* or *soft real-time*. Hard real-time programs are programs with very strict bounds on the

running times of program operations. Examples of hard real-time programs are airplane fly-by-wire control, missile guidance, and medical equipment control. The defining characteristic of hard real-time programs is that the consequences of missing a deadline are very great: the airplane crashes, the missile misses its target, the patient dies.

There are a number of programs which can benefit from a real-time collector, but do not have hard real-time requirements. We call these programs "soft real-time." Soft real-time programs are programs that should meet a majority of their deadlines, but it is acceptable if an occasional deadline is missed, as long as the deadlines are not missed too frequently at a time-scale relevant to the program. Examples of soft real-time programs are multimedia applications, graphical user interfaces, and non-critical control software. For these applications, it does not really matter if the occasional frame of video is missed or the mouse cursor occasionally skips a little, as long as this does not happen too often.

Hard Real-Time Garbage Collection Requirements

Hard real-time garbage collection has three requirements:

1. it must be *incremental*,
2. it must allow the application to make *progress*, and
3. it must use *bounded memory*.

Incremental Real-time garbage collection must be *incremental*; that is, it must be possible to perform small units of garbage collection work while an application is executing, rather than halting the application to perform large amounts of work without interruption. Strict bounds on individual garbage collection pauses are often used as the criterion for real-time garbage collection, but for practical applications, the requirements are often even stricter.

Progress A second requirement for real-time applications that has been almost universally overlooked in the real-time garbage collection literature is that the application must be able to make significant *progress*. That is, for a garbage collector to be usefully real-time, not only must the pauses be short and bounded, they must also not occur too often. In other words, the garbage collector must be able to *guarantee* not only that every garbage collection pause is bounded, but that for any given increment of computation, a minimum amount of the CPU is always available for the running application.

The difficulty with incremental garbage collection is that while the collector is tracing out the graph of reachable data structures, the graph may change — the running program may *mutate* the graph between invocations of the collector. For this reason, discussions of incremental collectors typically refer to the running program as the *mutator* [DLM+78]. An incremental scheme must have some way of keeping track of the changes to the graph of reachable objects, perhaps re-computing parts of its traversal in the face of those changes.

An important characteristic of incremental techniques is their degree of conservatism with respect to changes made by the mutator during garbage collection. If the mutator changes the graph of reachable objects, freed objects may or may not be reclaimed by the garbage collector. Some *floating garbage* may go unreclaimed because the collector has already categorized the object as live before the mutator frees it. This garbage *is* guaranteed to be eventually collected, however, just not during the same garbage collection cycle in which it became garbage.

Bounded Memory Finally, because of the critical nature of most real-time applications, it is important to guarantee space bounds. This issue is particularly complicated for garbage collected systems because the programmer no longer has direct control of when a block of memory becomes available for reuse. We present a model for real-time garbage collection that allows the programmer to select a

garbage collection design and reason about the worst case memory usage of his system.

Soft Real-Time Garbage Collection

Hard real-time applications (critical applications with strict deadlines) are very important and largely unaddressed by the garbage collection literature. At the same time, *soft* real-time applications (less critical real-time applications such as multimedia) make up an even larger set of problems that could benefit greatly from a real-time garbage collector. The issues in hard real-time garbage collection are very different from those in soft real-time. Hard real-time applications need *guarantees* on the worst-case time and space cost of any operation. Soft real-time applications, on the other hand, are often more interested in average case performance, even if it is at the risk of missing an occasional deadline, *as long as these deadlines are not missed too often.*

In this work, we develop a model for both hard *and* soft real-time garbage collection, that allows the garbage collector implementor to reason about the performance and memory usage of his collector. We also provide an implementation of a garbage collector that is fully configurable for both types of applications.

1.4.2 A Model for Real-Time Garbage Collection

A major contribution of this work is to provide a model for garbage collection broad enough in its scope to encompass:

- hard and soft real-time requirements,
- read-barrier and write-barrier strategies, and
- copying and non-copying implementations.

This model is based on tricolor marking [DLM+78] and is augmented with the key idea that garbage collection is really the process of marking objects and moving them from one *set* to another [Bak91]. In addition, this model uses two important invariants that allow us to address the issues of consistency and conservatism in incremental collection. Furthermore, this model allows us to make clear decisions about the kind of compiler support that will or will not be useful for the particular garbage collector design that is chosen. Finally, this model allows us to reason about the space, time, and predictability tradeoffs between different read- and write-barrier strategies.

While a detailed comparison of the locality properties of non-copying vs. copying memory allocation algorithms are beyond the scope of this dissertation, we address some important locality issues with our model. In particular, we suggest that non-copying algorithms may have significant locality advantages over their copying counterparts. However, non-copying algorithms are potentially vulnerable to severe memory fragmentation which can cause their memory requirements to explode beyond any reasonable bound. We show that, with some amount of compiler support and/or programmer effort, these costs can be kept small for a majority of programs. In addition, we attempt to characterize the cases where fragmentation will be unacceptably high, and a copying implementation would be more appropriate.

1.4.3 Generational Garbage Collection Techniques

Given a realistic amount of memory, efficiency of simple garbage collection is limited by the fact that the system must traverse all live data during a collection cycle. In most programs in a variety of languages, *most objects live a very short time, while a small percentage live much longer* [LH83, Ung84, Sha88, Zor90, DeT90b, Hay91]. While figures vary from language to language and from program to program, usually

between 80 and 98 percent of all newly-allocated heap objects die within a few million instructions, or before another megabyte has been allocated; the majority of objects die even more quickly, within tens of kilobytes of allocation.

Even if garbage collection cycles are fairly close together, separated by only a few kilobytes of allocation, most objects die before a collection and never need to be processed. Of the ones that do survive to be processed once, however, *a large fraction survive through many collections.* These objects are processed at every collection, over and over, and the garbage collector spends most of its time processing the same old objects repeatedly. This is the major source of inefficiency in simple garbage collectors.

Generational collection [LH83] avoids much of this repeated processing by segregating objects into multiple areas by age, and collecting areas containing older objects less often than the younger ones. Once objects have survived a small number of collections, they are "moved" to a less frequently collected area. Areas containing younger objects are collected quite frequently, because most objects there will generally die quickly, freeing up space; processing the few that survive does not cost much. These survivors are *advanced* to older status after a few collections, to keep processing costs down. [LH83, Moo84, Ung84, Wil92].

For stop-and-collect garbage collection, generational garbage collection has the additional benefit that most collections take only a short time (collecting just the youngest generation is much faster than a full garbage collection). This reduces the frequency of disruptive pauses, and for many programs without real-time deadlines, this is sufficient for acceptable interactive use. The majority of pauses are so brief (a fraction of a second) that they are unlikely to be noticed by users [Ung84]; the longer pauses for multi-generation collections can often be postponed until the system is not in use, or hidden within non-interactive compute-bound phases of program operation [WM89]. Generational techniques are often used as an acceptable

substitute for more expensive incremental techniques, as well as to improve overall efficiency.

Because generational techniques rely on a heuristic—the guess that most objects will die young, and that older objects will not die soon—they are not strictly reliable, and may degrade collector performance in the worst case. Thus, for some purely hard real-time systems, they are not attractive. For other hard real-time applications with well understood object lifetimes and periodic scheduling of tasks, or for general-purpose systems with mixed hard and soft deadlines, the normal-case efficiency gain is likely to be highly worthwhile and the worst case is likely to be manageable.

In this dissertation we will explore some novel generational garbage collection algorithms in an attempt to provide the benefit of generational techniques for many *soft* real-time applications. We propose and implement a design for generational garbage collection that is more amenable to real-time applications than any other design that we know of. The key point of our design is to largely decouple the collection of each generation from that of the others. This allows collection of different generations to run at different speeds, and to be scheduled with minimal coordination.

1.4.4 Performance Issues: Copying and Non-Copying Real-Time Garbage Collection

In this work, we attempt to clarify the different performance issues in both copying and non-copying real-time garbage collection.[3] It is impossible to pick a winning strategy for all real-time applications because different strategies lead to different performance tradeoffs, which are heavily dependent on the characteristics of the

[3]Note that non-copying collection need not incur the cost of the sweep phase of a mark-sweep collector as is commonly assumed. In Section 4.8.1 we explain a technique known as "fake copying" [Wan89] (also known as "implicit reclamation" [Bak91]) which avoids the cost of a sweep phase.

application. However, we attempt to provide guidelines that can be used based on the particular problem at hand.

1.5 Outline of this Dissertation

In the first third of this dissertation, we compare the fragmentation resulting from a number of different traditional memory allocation algorithms. In these experiments, we used actual traces of eight varied programs' allocation and deallocation requests. This is contrary to the standard methodology for studying fragmentation, where random memory requests are generated and used to simulate real traces. We show that using random traces to simulate real workloads is unsound because programs tend to have strong phase behavior, and tend to allocate many objects of only a few sizes rather than a number of objects of many similar sizes (as the random methodology seems to assume).

In the second third of this dissertation, we study the locality effects of the placement choices of non-moving memory allocation algorithms at both the cache and virtual memory level. We show that placement choices can have a large effect on locality, and that the best policies in terms of fragmentation also have the best locality characteristics. Because a memory allocator has complete control of the program's layout of dynamic memory, it seems obvious that the choice of memory allocation policy will have a major effect on the locality of reference of that program. Surprisingly, we were only able to find *a single paper* [GZH93] discussing the effects of non-moving memory allocation algorithms on locality of reference.

Having clarified these issues, we devote the final third of this dissertation to our work on real-time garbage collection. We pay particular attention to developing a model for real-time garbage collection that allows us to compare and contrast our work with that of others. We also discuss our implementation and provide some measurements of the performance of our collector.

Throughout this work, we attempt to provide a sound methodology with which memory management algorithms can be studied and compared. In particular, we are interested in measuring actual costs for actual programs, and characterizing the situations under which different algorithms would be attractive. We also carefully separate policy costs from implementation costs, so that we can focus on the inherent costs associated with a policy and not the noise caused by our particular implementation.

Chapter 2

Memory Allocation Studies

An important part of our research involved studying the "fragmentation problem." In this chapter, we present our results. We show that the problem of programs using excessive amounts of memory due to fragmentation is actually a problem of not recognizing that good allocation policies already exist, and have inexpensive implementations. We show that for most programs fragmentation costs can be far lower than was previously believed, and that for a large class of programs this cost is very near zero. In addition, we invalidate the traditional methodology for studying fragmentation and present a more sound approach, which uses trace-driven simulation of real programs.

This work has been motivated, in part, by our perception that there is considerable confusion about the nature of memory allocators, and about the problem of memory allocation in general. Worse, this confusion is often unrecognized, and allocators are widely thought to be fairly well understood. In fact, we know little more about allocators than was known twenty years ago, which is not as much as might be expected. The literature on the subject is rather inconsistent and scattered, and considerable work appears to be done using approaches that are quite limited.

This problem with the allocator literature has considerable practical importance: aside from the human effort involved in allocator studies *per se*, there are effects in the real world, both on computer system costs, and on the effort required to create real software.

We think it is likely that the widespread use of poor allocators incurs a loss of main and cache memory (and CPU cycles) of over a billion and a half U.S. dollars worldwide *per year*; a significant fraction of the world's memory and processor output may be squandered, at huge cost.[1]

Perhaps an even worse problem is the effect on programming style due to the widespread use of poorly designed allocators—either because better allocators are not widely known or understood, or because allocation research has failed to address the proper issues. Programmers avoid heap allocation in many situations because of perceived space or time costs, while other programmers implement special-case memory allocators for their programs in an attempt to improve upon the default implementation. This practice invariably results in wasted space, subtle bugs, and portability problems.[2]

The overwhelming majority of memory allocation studies to date have been based on a methodology developed in the 1960's [Col61], which uses synthetic traces intended to model "typical" program behavior. This methodology has the advantages that it is easy to implement and allows experiments to avoid quirky behavior specific to a few programs. Often the researchers conducting these studies went to great lengths to ensure that their traces had statistical properties similar to real

[1] According to the World Semiconductor Trade Statistics (WSTS) world-wide DRAM sales for 1996 were $39.8 billion. By 1999, this number is expected to increase to $56.3 billion [Tec97]. If just 20% of this memory is used for heap allocated data, and 20% of that memory is unnecessarily wasted, then over $1.5 billion of the memory sold in 1996 was wasted. This is expected to grow to $2.25 billion by 1999.

[2] It is our impression that UNIX programmers' usage of heap allocation went up significantly when Chris Kingsley's allocator was distributed with BSD 4.2 UNIX—simply because it was much faster than the allocators they'd been accustomed to.

programs. However, none of these studies showed the validity of using a randomly generated trace to predict performance on real programs, no matter how well the randomly generated trace statistically models the original program trace. As we show in Section 2.15, what all of this previous work ignores is that a randomly generated trace is *not* valid for predicting how well a particular allocator will perform on a real program.

We therefore decided to perform simulation studies on various implementations of `malloc()` using memory allocation traces from real programs. Using a large set of tools that we built, we measured how well synthetic traces approximate real program traces, as well as how well these malloc algorithms performed on the real traces. Much to our surprise, some well-known policies actually perform surprisingly well. So well, in fact, that fragmentation appears to already be a solved problem.

Another factor often overlooked in memory allocation research is that seemingly minor variations in policy can have dramatic effects on fragmentation. We have carefully to separated the costs of different policies, and present detailed descriptions of the policies that we study.

We will begin this chapter with a discussion on the basic issues in memory allocation research. Next, we will discuss the basic issues in memory allocator design. Following that, we will describe the allocation policies that we studied. We will do this in two sections, the first being an overview of memory allocation policies, and the second being a detailed description of the actual policies that we studied. In the subsequent sections we will describe our test programs and our experimental methodology. We will conclude this chapter with a presentation of our results.

2.1 Basic Issues in Memory Allocation Research

Allocators are sometimes evaluated using probabilistic analyses. By reasoning about the likelihood of certain events, and the consequences of those events for future

events, it may be possible to predict what will happen on average. For the general problem of dynamic storage allocation, however, the mathematics are too difficult. Unfortunately, to make probabilistic techniques feasible, important characteristics of the workload, such as the probabilities of *relevant* input events, must be known. The relevant characteristics are not understood, so the probabilities are simply unknown.

This is one of the major points of this work: the paradigm of statistical mechanics has been used in theories of memory allocation, but we believe that it is the wrong paradigm, at least as it is usually applied. Typically, researchers make strong assumptions that frequencies of individual events (e.g., allocations and deallocations) are the base statistics from which probabilistic models should be developed, and we believe that this is false.

The great success of statistical mechanics in other areas is due to the fact that such assumptions make sense in those areas. Gas laws, for example, are pretty good idealizations because aggregate effects of a very large number of individual events (e.g., collisions between molecules) do concisely express the most important regularities.

This paradigm is inappropriate for memory allocation, for two reasons. The first is simply that the number of objects involved is usually too small for asymptotic analyses to be relevant. However, this is not the most important reason. The main weakness of the statistical mechanics approach is that there are important *systematic* interactions that occur in memory allocation, due to phase behavior of programs. No matter how large the system is, basing probabilistic analyses on individual events is likely to yield the wrong answers if there are systematic effects involved which are not captured by the theory. Assuming that the analyses are appropriate for "sufficiently large" systems does not help here—the systematic errors will simply attain greater statistical significance.

The traditional methodology of using random program behavior implicitly

assumes that there is *no* ordering information in the request stream that could be exploited by the allocator—i.e., there is nothing in the sequencing of requests which the allocator can use as a hint to suggest which objects should be allocated adjacent to which other objects. Given a random request stream, the allocator has little control: no matter where objects are placed by the allocator, they die at random, and randomly create holes among the live objects. If some allocators do in fact exploit some real regularities in the request stream, the randomization of the order of object creation (in simulations) *ensures that this information is discarded before the allocator can use it.* Likewise, if an algorithm tends to systematically make mistakes when faced with real patterns of allocations and deallocations, randomization may hide that fact.

2.1.1 Random Simulations

The traditional technique for evaluating allocators is to construct several *traces* (recorded sequences of allocation and deallocation requests) thought to resemble "typical" workloads, and use those traces to simulate the performance of a variety of actual allocators. Since an allocator's performance is dependent only on the sequence of allocation and deallocation requests, this method can produce very accurate results *provided that the request sequence accurately models the behavior of real programs.*

Typically, however, the request sequences are not traces of the behavior of actual programs. They are "synthetic" traces that are generated automatically by a small subprogram; the subprogram is designed to *resemble* real programs in certain statistical ways. In particular, object size distributions are thought to be important, because they affect the fragmentation of memory into blocks of varying sizes. Object lifetime distributions are often, but not always, thought to be important because they affect when areas of memory are occupied and when they are free.

Given a set of object size and lifetime distributions, the small driver subprogram is used to generate a sequence of requests that obeys those distributions. This driver is typically a simple loop that repeatedly generates requests, using a pseudo-random number generator; at any point in the simulation, the next data object is chosen by randomly picking a size and lifetime, with a bias that probabilistically preserves the desired distributions. The driver also maintains a table of objects that have been allocated but not yet freed, ordered by their scheduled deallocation time. At each step of the simulation, the driver deallocates any objects whose deallocation times indicate that they have expired. One convenient measure of simulated "time" is the volume of objects allocated so far—i.e., the sum of the sizes of objects that have been allocated up to that step of the simulation.

An important feature of these simulations is that they tend to reach a steady state. After running for a certain amount of time, the volume of allocated objects reaches a level that is determined by the size and lifetime distributions. After that point, objects are allocated and deallocated in approximately equal numbers, and the memory usage tends not to vary much. Measurements are typically made by sampling memory usage at points after the steady state has presumably been reached.

There are three common variations of this simulation technique. The first is to use a simple mathematical function, such as a uniform or negative exponential distribution, to determine the sizes and lifetimes of objects. Exponential size distributions are often used because it has been observed that programs typically allocate more small objects than large ones. Historically, uniform size distributions were the most common in early experiments; exponential distributions then became increasingly common, as new data became available showing that real systems generally used many more small objects than large ones. Other distributions, notably Poisson and hyper-exponential, have also been used. Surprisingly, relatively recent papers

have used uniform size distributions, sometimes as the only distribution. Exponential lifetime distributions are also often used because programs are more likely to allocate short-lived objects than long-lived ones. As with size distributions, there has been a shift over time away from uniform lifetime distributions, often towards exponential distributions.

The second variation is to pick distributions in ways thought to resemble real program behavior. This variation is based on the observation that many programs allocate the majority of their objects from just a few different sizes. In general, this has not been a very precise model of real programs. Sometimes the sizes are chosen at random and allocated in uniform proportions, rather than in skewed proportions reflecting the fact that on average, programs allocate many more small objects than large ones.

The third variation is to use statistics gathered from real programs, to make the distributions more realistic. In almost all cases, size and lifetime distributions are assumed to be independent—the fact that different sizes of objects may have different lifetime distributions is generally assumed to be unimportant.

In general, there has been something of a trend toward the use of more realistic distributions, but this trend is not dominant. Even now, researchers often use simple and smooth mathematical functions to generate traces for allocator evaluation.[3] The use of smooth distributions is questionable, because it bears directly on issues of fragmentation. If in real programs objects of only a few sizes are allocated, then the free (and uncoalesceable) blocks are likely to be of those sizes, making it possible to find a perfect fit.[4] On the other hand, if the object sizes are smoothly distributed, then the requested sizes will almost always be slightly different, thus increasing the chances of fragmentation.

[3]We are unclear on why this should be, except that a particular theoretical and experimental paradigm [Kuh70] had simply become thoroughly entrenched by the early 1970's. (It is also somewhat easier than dealing with real data.)

[4]We show in Section 2.12 that this is in fact the case for the programs we studied.

2.1.2 Probabilistic Analyses

Since Knuth's derivation of the "fifty percent rule" [Knu73], there have been many attempts to reason probabilistically about the interactions between program behavior and allocator policy, and to assess the overall cost in terms of fragmentation and/or CPU time.

These analyses have generally made the same assumptions as random-trace simulation experiments (e.g., random object allocation order, independence of size and lifetime, and steady-state behavior). These simplifying assumptions were generally used in order to make the mathematics tractable. In particular, assumptions of randomness and independence make it possible to apply well-developed theories of stochastic processes (Markov models, etc.) to derive analytical results about expected behavior. Assumptions of randomness and independence make the problem very smooth (hence mathematically tractable) in a probabilistic sense. This smoothness has the advantage that it makes it possible to derive analytical results, but it has the disadvantage that it turns a real and deep scientific problem into a mathematical puzzle that is much less significant. Because these assumptions tend to be false for most real programs, these results are of limited usefulness.

2.1.3 What Fragmentation Really Is, and Why the Traditional Approach Is Unsound

Fragmentation is the inability to reuse memory that is free, *when that memory is needed*. This can be because of policy choices by the allocator, which may choose not to reuse memory that in principle could be reused. More importantly, this may be because the allocator does not have a choice at the moment an allocation request must be serviced: the free areas may not be large enough to service the request.[5]

[5]Beck [Bec82] makes the only clear statement of this principle which we have found in our exhausting review of the literature. His paper is seldom cited, and its important ideas have generally gone unnoticed.

Note that for this latter (and more fundamental) kind of fragmentation, the problem is a function both of the program's request stream and the allocator's choices of where to allocate the requested objects. In satisfying a request, the allocator usually has considerable leeway; it may place the requested object in any sufficiently large free area. On the other hand, the allocator has no control over the ordering of requests for different-sized pieces of memory, or over when those objects are freed.

In order to develop a sound methodology for studying fragmentation, it is necessary to understand what really causes fragmentation.

Fragmentation is caused by isolated deaths.

A crucial issue is the creation of free areas whose neighboring areas are not free. This is a function of two things: *which objects are placed in adjacent areas*, and *when those objects die*. Notice that if the allocator places objects together in memory, and they die at the same time (with no intervening allocations), no fragmentation results: the objects are live at the same time, using contiguous memory, and when they die they free contiguous memory. An allocator that can predict which objects will die at approximately the same time can exploit that information to reduce fragmentation by placing those objects in contiguous memory.

Fragmentation is caused by time-varying behavior.

Fragmentation arises from *changes* in the way a program uses memory—for example, freeing small blocks and requesting large ones. This much is obvious, but it is important to consider patterns in the changing behavior of a program, such as the freeing of large numbers of objects of one size and the subsequent allocation of large numbers of objects of a different size. Many programs allocate and free different kinds of objects in different stereotyped ways. Some kinds of objects accumulate

over time, but other kinds may be used in bursts. The allocator's job is to exploit these patterns, if possible, or at least not to let the patterns undermine its strategy.

Real programs do not generally behave randomly—they are designed to solve actual problems, and the methods chosen to solve those problems have a strong effect on the programs' patterns of memory usage. To begin to understand the allocator's task, it is necessary to have a general understanding of program behavior. This understanding is almost entirely absent in the literature on memory allocators, apparently because many researchers consider the infinite variation of possible program behaviors to be too daunting.

2.2 Basic Issues in Allocator Design

The main technique used by allocators to keep fragmentation under control is *placement choice.*

Placement choice is simply the choosing of where in free memory to allocate a requested block. The allocator has huge freedom of action—it can place a requested block anywhere it can find a sufficiently large range of free memory, and anywhere within that range. (It may also be able to simply request more memory from the operating system.) An allocator algorithm therefore should be regarded as the *mechanism* that implements a *placement policy*, which is motivated by a *strategy* for minimizing fragmentation. We believe that this is an important distinction to make, and that by carefully separating these issues, it will be easy to design memory allocators that have a number of desirable properties, such as high speed, low fragmentation, and good locality of reference.

2.2.1 Strategy, Policy, and Mechanism

Strategy takes into account regularities in program behavior, and determines a range of acceptable *policies* for placing requested blocks. The chosen policy is implemented

by a *mechanism*, which is a set of algorithms and data structures. This three-level distinction is quite important. In the context of general memory allocation,

- a *strategy* attempts to exploit regularities in the request stream,
- a *policy* is an implementable decision procedure for placing blocks in memory, and
- a *mechanism* is a set of algorithms and data structures that implement the policy, often called "the implementation."

An ideal strategy is "put blocks where they will not cause fragmentation later"; unfortunately this is impossible to guarantee, so real strategies attempt to heuristically approximate that ideal, based on assumed regularities of application programs' behavior. For example, one strategy is: "if a block must be split, potentially wasting what's left over, minimize the size of the wasted part." This is commonly believed to be the strategy for the best-fit family of allocators. However, as we will show in Section 2.10, this is *not* the strategy that makes best fit work well. The best-fit strategy is actually: "preferentially use one area of memory for allocation requests so that other areas will have more time for the neighboring objects to die and be coalesced."

The corresponding best-fit policy is more concrete—it says "always use the smallest block that is at least large enough to satisfy the request." This is not a complete policy, however, because there may be several equally good fits; the complete policy must specify which of those should be chosen.

The chosen policy is implemented by a specific mechanism, which should be efficient in terms of time and space overheads. For best fit, for example, either a linear list or an ordered tree structure might be used to record the addresses and sizes of free blocks, and a list or tree search could be used to find the next block to be allocated, as dictated by the policy.

These levels of the allocator design process interact. A strategy may not yield an obvious complete policy, and the seemingly slight differences between similar policies may actually implement interestingly different strategies. The chosen policy may not be obviously implementable at reasonable cost in space, time, or programmer effort; in that case some approximation may be used instead.

2.2.2 Splitting and coalescing

Two general techniques for supporting a range of (implementations of) placement policies are *splitting* and *coalescing* of free blocks. The allocator may split large blocks into smaller blocks arbitrarily, and use any sufficiently-large sub-block to satisfy the request. The remainders from this splitting can be recorded as smaller free blocks in their own right and used to satisfy future requests.

The allocator may also coalesce adjacent free blocks to yield larger free blocks. After a block is freed, the allocator may check to see whether the neighboring blocks are free as well, and merge them into a single, larger block. This is often desirable, because one large block is more likely to be useful than two smaller ones.

The cost of splitting and coalescing may not be negligible, however, especially if splitting and coalescing work *too well*—in that case, freed blocks will usually be coalesced with neighbors to form large blocks of free memory, and later allocations will have to split smaller chunks off those blocks to obtain the desired sizes. It often turns out that most of this effort is wasted, because the sizes requested later are largely the same as the sizes freed earlier, and the old small blocks could have been reused without coalescing and splitting (see Section 2.12). Because of this, many modern allocators use the policy of *deferred coalescing*—they avoid coalescing and splitting most of the time, but use it intermittently, to combat fragmentation.

2.2.3 Space vs. Time

It is well known that it is easy to write a memory allocator that is very fast, as long as space issues are not important. Kingsley's BSD 4.2 UNIX memory allocator is an example of such an allocator [Kin]. It is a simple segregated storage allocator (Section 2.4.1) that rounds all object request sizes up to powers of two minus a constant. Allocation and deallocation consists of just popping off from and pushing on to an array of linked lists, which can be implemented in just a couple of machine instructions. However, as we will show in Section 2.9, this allocation policy is among the worst that we studied in terms of fragmentation.

What is not well known is that it is easy to write a very fast memory allocator even when space issues are important. As we will show in Section 2.9, best fit and first fit address ordered are among the best allocation policies in terms of fragmentation. Stephenson described how to efficiently implement first fit address ordered using a cartesian tree[6] [Ste83]. Standish and Tadman showed how to efficiently implement best fit using two sets of free lists: an array of free lists of same-sized objects for small blocks, and a binary tree of free lists for larger blocks [Sta80, Tad78]. Unfortunately, this work seems to have gone unnoticed.

These allocation policies can be implemented even more efficiently if deferred coalescing (Section 2.4.4) is used in addition to the techniques described above. To date, it has been unclear whether deferred coalescing would affect fragmentation. However, because deferred coalescing changes the order of object reuse, there is every reason to believe that it could have a non-negligible effect. On the other hand, our results (Section 2.9.2) show that deferred coalescing does not appreciably increase fragmentation for the better allocation policies in our study.

For deferred coalescing to be effective, programs must repeatedly request the same sized objects, and these objects must have been recently freed. Our results

[6] Cartesian trees were first described in [Vui80].

(Sections 2.12 and 2.6.2) show in fact that most programs do allocate only a few different sizes of objects, and that these objects are only live for a short time. Thus, it is quite likely that deferred coalescing can be used, as we will describe in Section 2.4.4, to make the usual case allocation and deallocation times for good allocation policies as fast as simple segregated storage, at the cost of only a couple of percent in fragmentation.

In summary, by using good, scalable data structures such as those described in [Ste83] or [Sta80, Tad78], memory allocators with very low fragmentation need not be slow. In addition, by using deferred coalescing, the usual case can be optimized to be very fast with very little increase in fragmentation.

2.3 A Sound Methodology for Studying Fragmentation

The traditional view has been that the program behavior responsible for fragmentation is determined only by the distributions of object sizes and lifetimes. Recent experimental results show that this is false [ZG94, WJNB95], because the *ordering* of requests has a large effect on fragmentation. Until a much deeper understanding of program behavior is reached, and until allocator strategies and policies are as well understood as allocator mechanisms, the only reliable method for allocator simulation is to use real traces—i.e., the actual record of allocation and deallocation requests from real programs—as we describe in Section 2.8.

A sound methodology must also separate policy costs from implementation costs. When simulating real traces, it is important to measure the true costs of the policy being studied and not the overheads of the particular implementation of that policy. Finally, many policies are a composition of several simpler policies. For example, the policy best-fit with a LIFO free list, deferred coalescing, and a FIFO quick list is actually the combination of four policies: the best-fit policy with the LIFO-ordered free-list policy, the deferred coalescing policy, and the FIFO quick list

policy. It is important to try to separate as many of these costs from each other as possible in order to understand the effect of each policy choice.

Finally, a sound methodology must be clear about what it is attempting to study. As we will see in Section 2.9, small variations in policy can produce large variations in fragmentation. It is therefore important for allocation studies to carefully describe the exact policies under consideration. In the next two sections, we will describe the allocation policies that we study in this work. The first section is an overview of memory allocation policies in general, and the second section is a description of the particular policies that we studied for this work.

2.4 Overview of Memory Allocation Policies

In this section, we give an overview of allocator terminology.[7] The basic kinds of allocation policies we discuss are:

- *Segregated Free Lists*, including simple segregated storage and segregated fit.
- *Sequential Fits*, including first fit, next fit, and best fit.
- *Buddy Systems*, including conventional binary and double buddies.

In addition, we discuss the many policy decisions which must be made when implementing one of these allocators: order of object reuse, deferred coalescing, splitting thresholds, and preallocation. As stated earlier, an important point of this research is the separation of *policy* from *mechanism*. We believe that research on memory allocation should first focus on finding good policies. Once these policies are identified, it is relatively easy to develop good implementations. All of the measurements presented in this dissertation are for the memory allocation *policy* under

[7] For a much more extensive discussion on these issues, see [WJNB95]

consideration, independent of any particular *implementation* of that policy. Unfortunately, many good policies are discounted because the obvious implementation is inefficient. We will therefore devote some of this section to describing alternative implementations that are quite efficient for many of these policies.

2.4.1 Segregated Free Lists

One of the simplest allocation policies uses a set of free lists, where each list holds free blocks of a particular size. When a block of memory is freed, it is simply pushed onto the free list for that size. When a request is serviced, the free list for the appropriate size is used to satisfy the request. There are several important variations on this *segregated free lists* policy.

One common variation is to use *size classes* to group similar object sizes together in a single free list. Free blocks from a list are used to satisfy any request for an object whose size falls within that list's size class. A common size-class scheme is to use size classes that are a power of two apart (e.g., 4 words, 8 words, 16 words, and so on) and round the requested size up to the nearest size class.

Simple Segregated Storage

In this variant, no splitting of larger free blocks is done to satisfy requests for smaller sizes, and no coalescing of smaller free blocks is done to satisfy requests for larger sizes. When a request for a given size is serviced, and the free list for the appropriate size class is empty, more storage is requested from the underlying operating system (e.g., using UNIX `sbrk()` to extend the heap segment). Typically, one or two virtual memory pages are requested at a time, and split into same-sized blocks which are then strung together and put on the free list. Since the result is that pages (or some other relatively large unit) contain blocks of only one size class, we call this *simple segregated storage*.

An advantage of this simple policy is that it naturally leads to an implementation where no headers are required on allocated objects: the size information can be recorded for a page of objects, rather than for each object individually. This may be important if the average object size is very small.

Simple segregated storage can also be made quite fast in the usual case, especially when objects of a given size are repeatedly freed and reallocated over short periods of time. Because this policy does not split or coalesce free blocks, almost no work is done when an object is freed, and subsequent allocations of the same size can be quickly satisfied by removing that block from its free list.

The disadvantage of this scheme is that it is subject to potentially severe external fragmentation, as no attempt is made to split or coalesce blocks to satisfy requests for other sizes. The worst case is a program that allocates many objects of one size class and frees them, then does the same for many other size classes. In that case, separate storage is required for the maximum volume of objects of all sizes, and none can be reused for the others.

There is some tradeoff between expected internal fragmentation and external fragmentation with this scheme. If the spacing between size classes is large, more different sizes will fall into each size class, allowing space for some sizes to be reused for others. (In practice, very coarse size classes generally lose more memory to internal fragmentation than they save in external fragmentation. We will discuss this further in Section 2.9.2.)

A crude but possibly effective form of coalescing for simple segregated storage is to maintain a count of live objects for each page, and notice when a page is entirely free. If a page is free, it can be made available for allocating objects in a different size class, preserving the invariant that all objects in a page are of a single size class.

Multiple Sizes Per Page

At the expense of having per-object rather than per-page overheads, the simple segregated storage policy can be changed to allow objects from a larger size class to be split into smaller sizes, and objects from smaller size classes to be merged into larger sizes. In keeping with the simple segregated storage policy, this splitting and coalescing is constrained such that the resulting blocks are the exact size for another size class. For example, with powers-of-two size classes, a 64-byte object can be split into two 32-byte objects, or into one 16-byte object and into one 48-byte object, but not into one 50-byte object and one 14-byte object. This is similar to, but less constrained than, the buddy system which we describe in Section 2.4.3.

Segregated Fit

Another variation on the segregated free lists policy relaxes the constraint that all objects in a size class be exactly the same size. We call this segregated fit. This variant uses a set of free lists, each list holding free blocks of any size between the current size class and the next larger size class. When servicing a request for a particular size, the free list for the corresponding size class is searched for a block at least large enough to hold it. The search is typically a sequential-fit search, and many significant variations are possible (we describe a number of these variations in Section 2.4.2). Typically a first-fit or next-fit policy is used. It is often pointed out that the use of multiple free lists makes the *implementation* faster than searching a single free list. What is often *not* appreciated is that this also affects the *policy* in a very important way: the use of segregated lists excludes blocks of very different sizes, meaning *good* fits are usually found. The policy is therefore a *good-fit* or even a *best-fit* policy, despite the fact that it is usually described as a variation on first fit, and underscores the importance of separating policy considerations from implementation details.

2.4.2 Sequential Fits

Several classic allocator algorithm *implementations* are based on having a doubly-linked linear (or circularly-linked) list of all free blocks of memory. Typically, sequential-fit algorithms use Knuth's *boundary tag* technique to support coalescing of all adjacent free areas [Knu73]. The list of free blocks is usually maintained in either FIFO, LIFO, or address order (AO). Free blocks are allocated from this list in one of three ways: the list is searched from the beginning, returning the first block large enough to satisfy the request (*first fit*); the list is searched from the place where the last search left off, returning the next block large enough to satisfy the request (*next fit*); or the list is searched exhaustively, returning the smallest block large enough to satisfy the request (*best fit*).

These *implementations* are actually instances of allocation *policies*. The first-fit policy is to search some ordered collection of blocks, returning the first block that can satisfy the request. The next-fit policy is to search some ordered collection of blocks *starting where the last search ended*, returning the next block that can satisfy the request. Finally, the best-fit policy is to exhaustively search some collection of blocks, returning the best fit among the possible choices, and breaking ties using some ordering criteria. The choice of ordering of free blocks is also a policy decision. The three that we mentioned above as implementation choices (FIFO, LIFO, and address ordered) are also policy choices.

What is important is that each of these policies has several different possible implementations. For example, best fit can be implemented using a tree of lists of same sized objects [Sta80], and first fit address ordered can be implemented using a Cartesian tree [Ste83]. For concreteness and simplicity, we describe the well-known implementations of sequential-fit algorithms, but we stress that the same policies can be implemented more efficiently.

First fit

A first-fit policy simply searches the list of free blocks from the beginning, and uses the first block large enough to satisfy the request. If the block is larger than necessary, it is split and the remainder is put on the free list. A problem with this *implementation* of the first fit policy is that the larger blocks near the beginning of the list tend to be split first, and the remaining fragments result in having a lot of small blocks near the beginning of the list. This can increase search times because many small free blocks accumulate, and the search must go past them each time a larger block is requested. In terms of policy, this implementation of first fit may tend toward behaving like best fit over time, because the smallest blocks end up near the front of the list, so that blocks are effectively searched in size order, and the smallest chosen first.[8]

Next fit

A common "optimization" of first fit is to use a *roving pointer* for allocation [Knu73]. The pointer records the position where the last search was satisfied, and the next search begins from there. Successive searches cycle through the free list, so that searches do not always begin in the same place and result in an accumulation of small unusable blocks in one part of the list. The usual rationale for next fit is to decrease average search times, but this *implementation* consideration has other effects on the policy for memory reuse. Since the roving pointer cycles through memory regularly, objects from different phases of program execution may become interspersed in memory. This may affect fragmentation if objects from different phases have different expected lifetimes. (It may also seriously affect locality. The roving pointer itself may have bad locality characteristics since it examines every free block before touching any block again. Worse, it may affect the locality of the

[8]This has also been observed by Ivor Page (personal communication, February 1994).

program for which it is allocating memory by scattering objects used by certain phases and intermingling them with objects used by other phases.)

Best fit

A best-fit sequential-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.

Because of the time costs of an exhaustive search, the best-fit policy is often unnecessarily dismissed as being impossible to implement efficiently. This is unfortunate, because, as we will show in Section 2.9, best fit is one of the best policies in terms of fragmentation. By taking advantage of the observation that most programs use a large number of objects of just a few sizes, a best-fit policy can be quite efficiently implemented as a binary tree of lists of same-sized objects. In addition, segregated-fit algorithms (Section 2.4.1) can be a very good approximation to best fit and are easy to implement efficiently.

Boundary Tags and Per-Object Overheads

Sequential-fit techniques are usually implemented using *boundary tags* to support the coalescing of free areas [Knu73]. Each block of memory has a header and a footer field, both of which record the size of the block and whether it is in use. When a block is freed, the footer of the preceding block of memory is examined to see if it is free; likewise, the header of the following block is examined. Adjacent free areas are merged into larger free blocks. (This is where doubly-linked lists are useful—a block can be unlinked from anywhere in a doubly-linked list in constant time.)

There is a simple optimization which allows us to remove the footer boundary tag from an object *while it is allocated.* As we said above, the footer holds two

different pieces of information: the size of the block, and whether it is free or allocated. We make the observation that we need the size information only when the block is free because when the block is allocated, we cannot coalesce with the next block. Thus, we are left with the case that when the object is live, we only need one bit in the footer telling us that the object is live. Since memory is usually only allocated on word or double word boundaries, the size of all objects is some multiple of four or eight bytes. Thus the bottom 2 or 3 bits of the size are always zero. We can therefore store the allocated/free bit in the header of the *following* object, together with the allocated/free bit for that object. When an object is freed, we do not need the memory that occupied the last four bytes of that object, and can copy the object size from the header into the footer. This still leaves us with the case of a two word minimum object size, but when an object is allocated, the overhead is just one word.

Order of Object Reuse

One important detail for sequential-fit algorithms is the ordering of the objects on the free list. There are three common variations: FIFO, LIFO, and address ordered.

For FIFO free lists, objects returned to the free list are located in such a place that they will be the last object considered for the next allocation. In the case of first fit or best fit, this usually means the end of the free list. In the case of next fit, this means the location just before the roving pointer (such that the pointer will have to rove all the way around the list before coming to this block).

For LIFO free lists, objects returned to the free list are located in such a place that they will be the next object considered for allocation. In the case of first fit or best fit, this usually means the front of the free list. In the case of next fit, this means the location just after the roving pointer (such that this block will be the next block reached by the pointer).

For address ordered free lists, free objects are placed in the list in sorted order corresponding to the address of the start of the object. It may seem that the run-time costs of sorting the free list with every deallocation would be prohibitively expensive. However, another *implementation* of this *policy* is possible: if a bitmap is maintained, with one bit for every word or every two words, then freeing an object and placing it into sorted order is as simple as setting the corresponding bits in the bit map. This approach has the added benefit that no boundary tags are needed for coalescing—it happens automatically when the bits are set. Finally, free regions of memory can be searched quickly by looking at several bits at a time and using a table to determine if that bit pattern could possibly hold an object of the desired size.

2.4.3 Buddy Systems

Buddy systems [Kno65, PN77] are a variant of segregated lists, supporting a limited but efficient kind of splitting and coalescing. In the simple buddy schemes, the entire heap area is conceptually split into two large areas which are called buddies. These areas are repeatedly split into two smaller buddies, until a sufficiently small chunk is achieved. This hierarchical division of memory is used to constrain where objects are allocated, and how they may be coalesced into larger free areas. A free area may only be merged with its *buddy*, the corresponding block at the same level in the hierarchical division. The resulting free block is therefore always one of the free areas at the next higher level in the memory-division hierarchy. At any level, the first block of a buddy pair may only be merged with the following block of the same size; similarly, the second block of a buddy pair may only be merged with the first, which precedes it in memory. This constraint on coalescing ensures that the resulting merged free area will always be aligned on one of the boundaries of the hierarchical division of memory.

The purpose of the buddy allocation constraint is to ensure that when a block is freed, its (unique) buddy can always be found by a simple address computation, and its buddy will always be either a whole, entirely free chunk of memory, or an unavailable chunk. (An unavailable chunk may be entirely allocated, or may have been split and have some or all of its sub-parts allocated.) Either way, the address computation will always be able to locate the buddy's header—it will never find the middle of an allocated object.

Buddy coalescing is relatively fast, but perhaps the biggest advantage in some contexts is that it requires little space overhead per object—only one bit is required per buddy, to indicate whether the buddy is a contiguous free area. This can be implemented with a single-bit header per object or free block. Unfortunately, for this to work, *the size of the block being freed must be known*—the buddy mechanism itself does not record the sizes of the blocks. This is workable in some statically-typed languages, where object sizes are known statically and the compiler can supply the size argument to the freeing routine. In most current languages and implementations, however, this is not the case due to the presence of variable-sized objects and/or because of the way libraries are typically linked.

Several significant variations on buddy systems have been devised. Of these, we studied binary buddies and double buddies.

Binary Buddy

Binary buddy is the simplest and best-known of the buddy systems [Kno65]. In this scheme, all buddy sizes are a power of two, and each size is divided into two equal parts. This makes address computations simple, because all buddies are aligned on a power-of-two boundary offset from the beginning of the heap area, and each bit in the offset of a block represents one level in the buddy system's hierarchical splitting of memory—if the bit is 0, it is the first of a pair of buddies, and if the bit is 1, it

is the second. These operations can be implemented efficiently with bitwise logical operations.

A major problem with binary buddies is that internal fragmentation is usually relatively high—the expected case is about 25%, because any object size must be rounded up to the nearest power of two (minus a word for the header, if a bit cannot be stolen from the block given to the language implementation).

Double Buddy

Double buddy [Wis78, PH86] uses a different technique to allow a closer spacing of size classes. It uses two different buddy systems, with staggered sizes. For example, one buddy system may use powers-of-two sizes (2, 4, 8, 16, ...) while the other uses a powers-of-two spacing starting at a different size, such as 3, (the resulting sizes are 3, 6, 12, 24, ...). Request sizes are rounded up to the nearest size class in either series. This reduces the internal fragmentation by about half, but means that a block of a given size can only be coalesced with blocks in the same size series. [9]

2.4.4 Deferred Coalescing

As we will show in Section 2.12, most programs tend to allocate lots of objects of just a few sizes, repeatedly. We can take advantage of this behavior by waiting a while before coalescing free objects, and hoping that another request for an identically-sized object will occur soon. If such a request for an identically-sized object does occur soon, then we have saved the cost of first coalescing and then immediately splitting a chunk of memory. If at some point a request comes in for a block that cannot be satisfied by any existing free chunk of memory, all free objects are then coalesced and another attempt is made to satisfy the request. Note that if we keep

[9] To our knowledge, the implementation we built for the present study may actually be the only double buddy system in existence, though Page wrote a simulator that is almost an entire implementation of a double buddy allocator [PH86].

the uncoalesced objects in a separate area, we only need to coalesce these objects when we need more memory, and coalescing costs are no higher than if we had done the work immediately after the blocks were freed.

Quick Lists

One way to separate free objects that have not been coalesced from those that have is to create a special list for these objects, and then search this list before looking for a chunk in the coalesced list. However, a list search can be quite expensive. An optimization is to pick some small size (say 32 words) above which the allocator will always immediately coalesce, and create a list for every object size below this limit. These lists can be accessed from an array with one entry for every size, making the search extremely fast in the average case. Only if this search fails do we need to use the more general purpose mechanism.

Even if we have one list for every possible chunk size, such that no list search is ever necessary, it is still important to specify the order in which free objects are stored in these quick lists. As we will see in Section 2.9, the order of the quick lists can have a measurable effect on locality.

2.4.5 Splitting Thresholds

Once a block is chosen, the next decision to make is whether to use the entire block, or to split the chunk into two pieces and save the remainder for a later request. If the policy dictates that the chunk should be split, it is necessary to determine how much of the unneeded memory to keep with the object, and how much to keep with the free chunk. This is essentially the choice of increasing internal fragmentation to decrease external fragmentation. There are several ways to make this decision:

- always keep blocks at a predetermined size (such as powers of two, or a Fibonacci number),

- try to split the block into two equal sizes, or
- split the block with a given percentage of the request size as internal fragmentation.

It has long been believed that increasing internal fragmentation to reduce external fragmentation is a good tradeoff. In fact, buddy systems and simple segregated storage systems depend on this trade-off as a part of their basic policy. However, one of the results of our research is that this appears to never be a good choice. We discuss this result in more detail in Section 2.9.

2.4.6 Preallocation

One possible way to speed up the *implementation* of a memory allocator is to preallocate a number of blocks of a size that is expected to be heavily used. This heuristic is often compared to getting water from a well: when one needs to get a cup of water from a well, one does not just get one cup, one gets a bucket full. In memory allocator terms, if a request comes in for a particular object size, the allocator finds a suitably large block, splits it into several blocks of this size, and puts them into a quick list.

What is often not understood about this heuristic is that it also has important *policy* implications. Notice that for this heuristic to work, deferred coalescing must also be implemented. Also, the blocks that are pre-split are no longer available if a request for a different size needs to be fulfilled. This can eventually lead to a very different set of blocks being used than if this heuristic had not been implemented.

Half fit & Multi-fits

Another variation on sequential fits, which is also a variation on preallocation is called a *multiple-fit*. In this variation, the list of free memory chunks is searched

for a block that is exactly some multiple of the request size. If such a block is found, then it is split into several free blocks each being exactly the request size. This variation also relies on the heuristic that if there is a request for a particular size, then there is likely to be another request for that same size soon. However, it is different from normal preallocation in that it attempts to minimize remainder chunks that are not of a size that can be easily used.

The simplest version of this policy, which we call *half fit*, is to always look for a block that is exactly twice the request size and split it into two blocks of the same size. This version attempts to gain the benefit of preallocating some memory, without over-committing to block sizes in case the heuristic is wrong.

2.4.7 Wilderness Preservation

The treatment of the last block in the heap—the memory that the allocator most recently obtained from the operating system—can be quite important. This block is usually rather large, and a mistake in managing it can be expensive. Since such blocks are allocated whenever heap memory grows, consistent mistakes could be disastrous [KV85]. Thus, there is the very important question of how to treat a virgin block of significant size, to minimize fragmentation. (This block is sometimes called the "wilderness" [Ste83] to signify that it is as yet unspoiled.)

Consider what happens if a first-fit or next-fit policy is being used, and the wilderness block is placed at the beginning of the free list. The allocator will most likely carve many small objects out of the wilderness immediately, greatly increasing the chances of being unable to recover the contiguous free memory of the block. On the other hand, putting it on the opposite end of the list will tend to leave it unused for at least a while, perhaps until it gets used for a larger block or blocks. An alternative strategy is to keep the wilderness block out of the main ordering data structure entirely, and only carve blocks out of it when no other usable memory can

be found.

Korn and Vo call this a "wilderness preservation heuristic," and report that it is helpful for some allocators [KV85] (however, no quantitative results are given). Our results show that for the best allocation policies (best fit and first fit address ordered), special treatment of the wilderness block is unnecessary. We will describe this in more detail in Section 2.9.

2.5 Allocator Descriptions

We obtained and/or constructed a variety of allocators, representative of the classes of allocation policies we described earlier: segregated free lists (simple segregated storage and segregated fit), sequential fit, and buddy systems, which we describe here in detail.

The reader may find this section tedious, and it would be acceptable to skim it on the first reading. However, we have found that seemingly inconsequential differences in policy can lead to dramatically different fragmentation results (see Section 2.13) and have taken great pains to adequately describe our allocators. One of the great disappointments we had while reading the related work was that very few of the allocators studied were described in enough detail for us to recreate their results. Thus, we encourage the reader to eventually return to this section, and to pay careful attention to the details outlined here. We particularly encourage future researchers to follow our example and explain their allocation policies in sufficient detail that their experimental results can be repeated.

In the descriptions which follow, unless otherwise noted, all object sizes are rounded up to the nearest double word (8 bytes or 32 bits),[10] and the minimum object size is four words (16 bytes). Memory is requested from the operating system in units of 4KB, except for double buddy, which requests an average of 5KB at a

[10] As required by the alignment of double floats on the Sparc architecture.

time.[11]

2.5.1 Segregated Free Lists

In this section, we present descriptions of our segregated free list allocators: simple segregated storage (2^N and 2^N & $3 * 2^N$) and segregated fit (Doug Lea's 2.5.1 and 2.6.1) allocators.

Simple Segregated Storage

This is a very simple segregated storage algorithm that does no coalescing. It maintains an array of free lists for size classes. The implementations of this allocator used in our fragmentation studies have no header or footer overhead because no coalescing is done, and because all objects in a page are of the same size.[12] The versions of this allocator used in our locality studies (Chapter 3) do have a header, but still have no footer. Objects are placed on and removed from their free lists in LIFO order. The minimum object size is 16 bytes. We have two implementations of this algorithm:

- *Simple Seg* 2^N allocates objects in size classes that are powers of two (e.g., 16, 32, 64, etc., bytes). This allocator was originally written by Sheetal Kakkad for use in the Texas Persistent Store [SKW92], but is very similar to the widely used and venerable BSD UNIX allocator written by Chris Kingsley and studied by Zorn and Grunwald [ZG94]. (However, Zorn and Grunwald incorrectly describe this allocator as a "buddy-based algorithm.")

[11] Recall that double buddy actually uses two heap areas. In one heap area memory is requested from the operating system in units of 4KB, and in the other, memory is requested in units of 6KB.

[12] Only one word of overhead is required per page (about a tenth of a percent of the heap size). This word is used for encoding the sizes of objects in a page so that when objects are freed, they can be placed on the appropriate free list. We ignored this cost because it is negligible given the slight imprecision in our measurements (see Section 2.8.3).

- *Simple Seg* 2^N *&* $3 * 2^N$ is very similar to Simple Seg 2^N, but the size classes are closer together, to decrease internal fragmentation at a possible expense in external fragmentation. Size classes are powers of two, plus intermediate size classes that are three times powers of two (e.g., 16, 24, 32, 48, 64, etc., bytes). The minimum object size is 16 bytes. A simple table lookup technique is used to make size class determination fast for small objects. In places in this text where we are constrained for space, we will often abbreviate this allocator as Simple Seg $3 * 2^N$. This should not be mistaken for an allocator that omits the 2^N size classes. This is another version of the Texas allocator, implemented by Sheetal Kakkad and Michael Neely.

Segregated Fit

These memory allocators are from Douglas Lea, and are widely distributed and used with g++ (the GNU C++ compiler). We used three versions: 2.5.1, 2.5.1 with the footer overhead optimized away, and 2.6.1 (which has no footer overhead). At the time of this writing, the most recent version is 2.6.4, which we did not study.

- *Lea 2.5.1:* a "segregated storage" algorithm in the (rather misleading) sense of Purdom, Stigler, and Cheam [PSC71]. Actual storage is *not* segregated, and one-word header and footer fields support boundary-tag coalescing. A set of free lists is maintained, "segregating" (indexing) free objects by approximate size to speed up searches. Size classes are powers of two divided linearly in 4 (powers of two give a logarithmic set of size classes, and those sets are subdivided into 4 smaller ranges by simple linear division, i.e.: 4, 5, 6, 7; 8, 10, 12, 14; 16, 20, 24, 28; ... words). Each of the resulting size classes has a conventional doubly-linked free list searched using first fit. Several optimizations support a limited form of deferred coalescing and deferred reuse. Note that despite using a first-fit mechanism, the use of fairly precise size classes ensures

that it implements a policy that is very close to best fit.[13] (This has generally been overlooked.) Minimum object size is 16 bytes.

- *Lea 2.5.1 no footer:* the same allocator as Lea 2.5.1 but with the one-word footer overhead optimized away as described in Section 2.4.2.

- *Lea 2.6.1:* a revision of previous versions of this allocator. Free blocks are separated into 128 bins, with one bin for each block size less than 512 bytes. Objects are sorted by size within bins, with ties broken by an oldest-first rule (FIFO). Free blocks are immediately coalesced using boundary tags, and the smallest chunk size is 16 bytes. There are no footers on allocated objects, making the per-object overhead just 4 bytes. This algorithm more closely resembles best fit than previous versions with one important modification: when a block of the exact desired size is not found, the most recently split object is used (and re-split), if it is big enough; otherwise best fit is used. For very large objects (greater than 1 megabyte), if the requested space is not already available the memory is obtained via `mmap` rather than `sbrk`, and treated separately.

2.5.2 Sequential Fits

These allocators use a single free list and Knuth's boundary tag technique with a one word header to support coalescing. The versions with "no footer" in their names have no footer overhead on allocated blocks, whereas the other versions have a one word footer. The minimum object size is 16 bytes. Block are only split if the remainder is at least 16 bytes, and the remainder is put back on the free list. No other splitting threshold is used to trade internal fragmentation for reduced external

[13] The "first-fit" search within a size class looks for a very good fit (within less than the minimum object size) and forces coalescing if one is not found. Because blocks of very different sizes are not considered unless no other free blocks are available, most of the time a good fit will be selected.

fragmentation. When memory is requested from the operating system, it is always in 4K increments. The code for these allocators is based on code from Douglas Lea's g++ allocator, version 2.5.1, extracted and modified by Michael Neely.

There are three basic policies for searching the free list for a suitable block:

- *First fit:* a classic first-fit algorithm from Knuth, where the free list is always searched from the beginning, and the searching always stops as soon as the first block that is large enough is found.
- *Next fit:* a modified first-fit algorithm, using a roving pointer to avoid searching the list from the beginning each time, in an attempt to prevent the accumulation of small fragments at the beginning of the list. Thus, the search for a suitable free block begins where the search for the last block left off. The search always stops as soon as the first block that is large enough is found.
- *Best fit:* another modified first-fit algorithm. The free list is searched exhaustively or until an exact fit is found. If no exact fit is found, then the *smallest* block *larger* than the requested size is used.[14]

In these policies, newly freed objects, remainders from splitting, and new memory from the operating system are placed on the free list in one of three ways:

- *LIFO:* they are the first blocks to be considered for allocation,
- *FIFO:* they are the last blocks to be considered for allocation, or
- *Address Ordered (AO):* they are placed on the free list in increasing order of address, and are only considered for allocation when the normal search mechanism (first fit, next fit, or best fit) reaches them in the free list.

[14]This is not intended to be a realistic mechanism; it is simply a test of the best-fit *policy*.

Coalescing of both split remainders and/or freed objects is either immediate or deferred. In the case of deferred coalescing, separate free lists (called *quick lists*) are maintained for every size up to 32 words, and objects of 32 words or less are only coalesced if no suitable block is found for a request. Objects of greater than 32 words are always immediately coalesced. The quick lists can be maintained in LIFO, FIFO, or address order, independently of whether the main free list is in LIFO, FIFO, or address order.

The following is a description of each of our sequential fits allocators:

- *Best fit AO.* Uses the best-fit policy, and free memory is maintained in address order.
- *Best fit AO 8K.* Uses the best-fit policy, free memory is maintained in address order, and new memory is requested from the operating system in 8K increments.
- *Best fit AO deferred AO.* Uses the best-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in address order.
- *Best fit AO deferred FIFO.* Uses the best-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in FIFO order.
- *Best fit AO deferred LIFO.* Uses the best-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in LIFO order.
- *Best fit AO no footer.* Uses the best-fit policy, free memory is maintained in address order, and there is no footer on allocated blocks.

- *Best fit FIFO.* Uses the best-fit policy, and free memory is maintained in FIFO order.

- *Best fit FIFO no footer.* Uses the best-fit policy, free memory is maintained in FIFO order, and there is no footer on allocated blocks.

- *Best fit LIFO.* Uses the best-fit policy, and free memory is maintained in LIFO order.

- *Best fit LIFO deferred AO.* Uses the best-fit policy, free memory is maintained in LIFO order, uses deferred coalescing, and the quick lists are maintained in address order.

- *Best fit LIFO deferred FIFO.* Uses the best-fit policy, free memory is maintained in LIFO order, uses deferred coalescing, and the quick lists are maintained in FIFO order.

- *Best fit LIFO deferred LIFO.* Uses the best-fit policy, free memory is maintained in LIFO order, uses deferred coalescing, and the quick lists are maintained in LIFO order.

- *Best fit LIFO no footer.* Uses the best-fit policy, free memory is maintained in LIFO order, and there is no footer on allocated blocks.

- *Best fit LIFO split-14.* Uses the best-fit policy, free memory is maintained in LIFO order, and blocks are only split if the remainder is at least 14% of the request size.

- *Best fit LIFO split-7.* Uses the best-fit policy, free memory is maintained in LIFO order, and blocks are only split if the remainder is at least 7% of the request size.

- *First fit AO.* Uses the first-fit policy, and free memory is maintained in address order.

- *First fit AO 8K.* Uses the first-fit policy, free memory is maintained in address order, and memory is requested from the operating system in 8K increments.

- *First fit AO deferred AO.* Uses the first-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in address order.

- *First fit AO deferred FIFO.* Uses the first-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in FIFO order.

- *First fit AO deferred LIFO.* Uses the first-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in LIFO order.

- *First fit AO no footer.* Uses the first-fit policy, free memory is maintained in address order, and there is no footer on allocated blocks.

- *First fit FIFO.* Uses the first-fit policy, and memory is maintained in FIFO order.

- *First fit FIFO no footer.* Uses the first-fit policy, memory is maintained in FIFO order, and there is no footer on allocated blocks.

- *First fit LIFO.* Uses the first-fit policy, and memory is maintained in LIFO order.

- *First fit LIFO deferred LIFO.* Uses the first-fit policy, free memory is maintained in LIFO order, uses deferred coalescing, and the quick lists are maintained in LIFO order.

- *First fit LIFO no footer.* Uses the first-fit policy, memory is maintained in LIFO order, and there is no footer on allocated blocks.

- *First fit LIFO split-14.* Uses the first-fit policy, free memory is maintained in LIFO order, and blocks are only split if the remainder is at least 14% of the request size.

- *First fit LIFO split-7.* Uses the first-fit policy, free memory is maintained in LIFO order, and blocks are only split if the remainder is at least 7% of the request size.

- *Half fit.* This is a best-fit policy with the addition that blocks that are exactly twice as large as the request size are preferentially selected.

- *Multi-fit Max.* This is a best-fit policy with the addition that the largest block that is an exact multiple of the request size is preferentially selected.

- *Multi-fit Min.* This is a best-fit policy with the addition that the smallest block that is an exact multiple, and at least twice as big as, the request size is preferentially selected.

- *Next fit AO.* Uses the next-fit policy, and free memory is maintained in address order.

- *Next fit AO 8K.* Uses the next-fit policy, free memory is maintained in address order, and memory is requested from the operating system in 8K increments.

- *Next fit AO deferred AO.* Uses the next-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in address order.

- *Next fit AO deferred FIFO.* Uses the next-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained

in FIFO order.

- *Next fit AO deferred LIFO.* Uses the next-fit policy, free memory is maintained in address order, uses deferred coalescing, and the quick lists are maintained in LIFO order.

- *Next fit AO no footer.* Uses the next-fit policy, free memory is maintained in address order, and there is no footer on allocated blocks.

- *Next fit FIFO.* Uses the next-fit policy, and free memory is maintained in FIFO order.

- *Next fit FIFO no footer.* Uses the next-fit policy, memory is maintained in FIFO order, and there is no footer on allocated blocks.

- *Next fit LIFO.* Uses the next-fit policy, and free memory is maintained in LIFO order.

- *Next fit LIFO deferred LIFO.* Uses the next-fit policy, free memory is maintained in LIFO order, uses deferred coalescing, and the quick lists are maintained in LIFO order.

- *Next fit LIFO no footer.* Uses the next-fit policy, memory is maintained in LIFO order, and there is no footer on allocated blocks.

- *Next fit LIFO split-14.* Uses the next-fit policy, free memory is maintained in LIFO order, and blocks are only split if the remainder is at least 14% of the request size.

- *Next fit LIFO split-7.* Uses the next-fit policy, free memory is maintained in LIFO order, and blocks are only split if the remainder is at least 7% of the request size.

- *Next fit LIFO WPH.* Uses the next-fit policy, free memory is maintained in LIFO order, and uses the wilderness preservation heurstic.

2.5.3 Buddy Systems

We have implemented three buddy system allocators. All have a one word header and no footer overhead. The minimum object size for all three allocators is 16 bytes.

- *Binary Buddy:* a classic binary buddy system. Memory is allocated in size classes that are powers of two, (i.e., 4, 8, 16, 32, ... words). Memory is requested from the operating system in 4K increments. This memory allocator was originally implemented for the COSMOS circuit simulator [BBB+88, Bea97].

- *Double Buddy 5K:* a double buddy system, using a pair of buddy systems to manage memory for two different (staggered) sets of power-of-two size classes. One buddy system manages memory for size classes that are powers of two, (i.e., 4, 8, 16, 32, ... words) the other for three times powers of two (i.e., 6, 12, 24, 48, ... words). In this implementation, memory reclaimed in one buddy system is not available for use in the other, sometimes limiting the effectiveness of coalescing. Memory is requested from the operating system in 4K and 6K increments (averaging to 5K increments).

- *Double Buddy 10K:* the same allocator as double buddy 5K, except that memory is requested from the operating system in 8K and 12K increments (averaging to 10K increments).

2.5.4 The Selected Allocators

The following ten allocation policies are a representative sampling of the major allocation policies we studied for this dissertation:

- Binary buddy
- Double buddy 5K
- Best fit LIFO no footer (nf)
- First fit AO no footer (nf)
- First fit LIFO no footer (nf)
- Half fit
- Lea 2.6.1
- Next fit LIFO no footer
- Simple segregated storage 2^N
- Simple segregated storage 2^N & $3 * 2^N$

We will present numbers for this subset of our allocation policies in the main body of this dissertation, in order to keep the discussion manageable. We present the full results in Appendices A and B. However, when small policy changes *do* make a large difference, and these differences are not reflected in our selected allocators, we will point them out in the body of this dissertation.

2.6 The Test Programs

For our test programs, we used eight varied C and C++ programs that run under UNIX (SunOS 5.5). These programs allocate between about 1.3 and 104 megabytes of memory during a run, and have a maximum of between 69 KB and 2.3 MB of live data at some point during execution. On average they allocate 27 MB total data, and on average have a maximum of about 966K live data at some point during their run. Three of our eight programs were used by Zorn and Grunwald, *et al.*, in earlier

studies. We use these three to attempt to provide some points of comparison while also using new and different memory-intensive programs.

2.6.1 Test Program Selection Criteria

We chose allocation-intensive programs because they are the programs for which allocator differences matter most. Similarly, we chose programs that have a large amount of live data because those are the ones for which space costs matter most. Another practical consideration is that some of our measurements of memory usage may introduce errors of up to 4 or 5 KB in bad cases; we wanted to ensure that the errors were generally small relative to the actual memory usage and fragmentation.

More importantly, some of our allocators are likely to incur extra overhead for small heap sizes, because they allocate in more than one area; they may have several partly-used pages, and unused portions of those pages may have a pronounced effect when heap sizes are very small. We think that such relatively fixed costs are less significant than the allocators' scalability to medium- and large-sized heaps.[15]

We obtained a wide variety of traces, including several that are widely used as well as CPU- and memory-intensive. In selecting the programs from many that we had obtained, we ruled out several for various reasons. We attempted to avoid over-representation of particular program types, i.e., too many programs that did the same thing. In particular, we avoided having several scripting language interpreters—even though such programs are generally portable, widely available, and widely used, they typically are not performance-critical; their memory use typically does not have a very large impact on overall system resource usage.

We ruled out some programs that appeared to "leak" memory, i.e., failed to

[15]Two programs used by Zorn and Grunwald [ZG92] and by Detlefs, Dosser, and Zorn [DDZ93], which we did not use, have heaps that are quite small: Cfrac only uses 21.4 KB and Gawk only uses 41 KB, which are only a few pages on most modern machines. Measurements of CPU costs for these programs are interesting, because they are allocation-intensive, but measurements of memory usage are less useful, and have the potential to obscure scalability issues with boundary effects.

discard objects at the proper point, and led to a monotonic accumulation of garbage in the heap. One of the programs we chose, P2C, is known to leak under some circumstances, and we left it in after determining that it could not be leaking much during the run we traced. Its basic memory usage statistics are not out of line with our other programs: it deallocates over 90% of all allocated bytes, and its average object lifetime is lower than most. Our justification for including this program is that many programs do in fact leak, so having one in our sample is not unreasonable. It is a fact of life that deallocation decisions are often extremely difficult for complex programs, and programmers often knowingly choose to let programs leak on the assumption that over the course of a run the extra memory usage is acceptable.[16] They choose to have poorer resource usage because attempts at plugging the leaks often result in worse bugs—dereferencing dangling pointers and corrupting data structures.

We should note here that in choosing our set of traces, among the traces we excluded were three that did very little freeing, i.e., all or nearly all allocated objects live until the end of execution. (Two of these were the PTC and YACR programs from Zorn *et al.*'s experiments.)[17] We believe that such traces are less interesting because any good allocator will do well for them. This biases our sample slightly toward potentially more problematic traces, which have more potential for fragmentation. Our suite does include one almost non-freeing program, LRUsim, which is the only non-freeing program we had that we were sure did not leak.

[16] One very memory-intensive program which we considered, we did not use because it had serious leaks. These leaks survived three months of highly-skilled programmers' attempts at fixing them. Rather than restructuring their entire program and losing much of its modularity solely to allow objects to be correctly allocated, they eventually chose to use the Boehm-Weiser conservative garbage collector.

[17] Other programs were excluded because they had too little live data (e.g., LaTeX), or because we could not easily figure out whether their memory use was hand-optimized, or because we judged them too similar to other programs we chose.

program	Kbytes alloc'd	run time	max objects	num objects	max Kbytes	avg lifetime
Espresso	104,388	146	4,390	1,672,889	263	15,478
GCC	17,972	167	86,872	721,353	2,320	926,794
Ghostscript	48,993	53	15,376	566,542	1,110	786,699
Grobner	3,986	8	11,366	163,310	145	173,170
Hyper	7,378	131	297	108,720	2,049	10,531
LRUsim	1,397	29,940	39,039	39,103	1,380	701,598
P2C	4,641	30	12,652	194,997	393	187,015
Perl	33,041	114	1,971	1,600,560	69	39,811
Average	27,725	3,823	21,495	633,434	966	355,137

Table 2.1: Basic statistics for the eight test programs

2.6.2 The Selected Test Programs

We used eight programs because this was sufficient to obtain statistical significance for our major conclusions. (Naturally it would be better to have even more, but for practicality we limited the scope of these experiments to eight programs and a comparable number of basic allocation policies to keep the number of combinations reasonable.) Whether the programs we chose are "representative" is a difficult subjective judgment: we believe they are reasonably representative of applications in conventional, widely-used languages (C and C++), however we encourage others to try our experiments with new programs to see if our results continue to hold true.

Table 2.1 gives some basic statistics for each of our eight test programs:

- the *Kbytes alloc'd* column gives the total allocation in Kilobytes over a whole run;
- the *run time* column gives the running time in seconds on a Sun SPARC ELC, an 18.2 SPECint92 processor, when linked with the standard SunOS allocator (a Cartesian-tree based "better-fit" (indexed-fits) allocator);
- the *max objects* column gives the maximum number of live objects at any time

during the run of the program;

- the *num objects* column gives the total number of objects allocated over the life of the program;
- the *max Kbytes* column gives the maximum number of kilobytes of memory used by live objects at any time during the run of the program[18] (note that if the average size of objects varies over time, the maximum live objects and maximum live bytes might not occur at the same point in a trace); and
- the *avg lifetime* column gives the average object lifetime in bytes, which is the number of bytes allocated between the birth and death of an object, weighted by the size of the object (that is, it is really the average lifetime of an allocated byte of memory).

Descriptions of the programs follow, to allow others to assess how representative our sample is for their own workloads.

- *Espresso* is a widely used optimizer for programmable logic arrays. The file `largest.espresso`, provided by Ben Zorn, was used as the input.
- *GCC* is the main process (`cc1`) of the GNU C compiler (version 2.5.1). We constructed a custom tracer that records *obstack*[19] allocations to obtain this trace, and built a postprocessor to translate the use of obstack memory into equivalent `malloc()` and `free()` calls.[20] The input data for the compilation was the the largest source file of the compiler itself (`combine.c`).[21]

[18] This is the maximum of the number of kilobytes *in use* by the program for actual object data, not the number of bytes used by any particular allocator to service those requests.

[19] Obstacks are an extension to the C language, used to optimize the allocation and deallocation objects in stack-like ways. A similar scheme is described in [Han90].

[20] It is our belief that we should study the behavior of the program without hand-optimized memory allocation, because a well-designed allocator should usually be able to do as well as or better than most programmers' hand optimizations. Some support for this idea comes from [Zor93], which showed that hand optimizations usually do little good compared to choosing the right allocator.

- *Ghost* is Ghostscript, a widely-used portable interpreter for the Postscript (page rendering) language, written by Peter Deutsch and modified by Zorn, *et al.*, to remove hand-optimized memory allocation [Zor93]. The input was `manual.ps`, the largest of the standard inputs available from Zorn's ftp site. This document is the 127-page manual for the Self system, consisting of a mix of text and figures.[22]

- *Grobner* is (to the best of our very limited understanding) a program that rewrites a mathematical function as a linear combination of a fixed set of Grobner basis functions.[23]

- *Hyper* is a hypercube network communication simulator written by Don Lindsay. It builds a representation of a hypercube network, then simulates random messaging, accumulating statistics about messaging performance. The hypercube itself is represented as a large array, which essentially lives for the entire run; each message is represented by a small heap-allocated object, which lives very briefly—only long enough for the message to get where it is going, which is a tiny fraction of the length of the run.

- *LRUsim* is an efficient locality analyzer written by Douglas Van Wieren. It consumes a memory reference trace and generates a grey-scale Postscript plot of the evolving locality characteristics of the traced program. Memory usage is dominated by a large AVL tree[24] which grows monotonically. A new entry

[21] Because of the way the GNU C compiler is distributed, this is a very common workload—people frequently down-load a new version of the compiler, compile it with an old version, then recompile it with itself twice as a cross-check to ensure that the generated code does not change between self-compiles (i.e., it reaches a fixed point).

[22] Note that this is not the same input set as used by Zorn, *et al.*, in their experiments: they used an unspecified combination of several programs. We chose to use a single, well-specified input file to promote replication of our experiments.

[23] Abstractly, this is roughly similar to a Fourier analysis, decomposing a function into a combination of other, simpler functions. Unlike a Fourier analysis, however, the process is basically one of rewriting symbolic expressions many times, something like rewrite-based theorem proving, rather than an intense numerical computation over a fixed set of array elements.

is added whenever the first reference to a block of memory occurs in the trace. Input was a reference trace of the P2C program.[25]

- *P2C* is a Pascal-to-C translator, written by Dave Gillespie at Caltech. The test input was `mf.p` (part of the Tex release). Note: although this translator is from Zorn's program suite, this is *not* the same Pascal-to-C translator (PTC) Zorn *et al.* used in their studies. This one allocates and deallocates more memory, at least for this input.

- *Perl* is the Perl scripting language interpreter (version 4.0) interpreting a Perl program that manipulates a file of strings. The input, `adj.perl`, formatted the contents of a dictionary into filled paragraphs. Hand-optimized memory allocation was removed by Zorn [Zor93].

2.7 Trace-Driven Memory Simulation

Trace-driven memory simulation [UM97] is the process of capturing a trace of the events of interest (instructions, loads, and stores, or allocation and deallocation requests) of actual programs running on actual hardware, and then using these traces to simulate and study different computer designs. The idea of trace-driven memory simulation is not new. In his survey of cache memories, A. J. Smith [Smi82]

[24]The AVL tree is used to implement a least-recently-used ordering queue. The AVL tree implementation was enhanced to maintain a count at each node of the descendents to the left of the node, used to compute the LRU queue position of a node in logarithmic time, as well as supporting logarithmic time deletion and insertion to move a node to the beginning of the queue when the block it represents is referenced.

[25]The memory usage of LRUsim is not sensitive to the input, except in that each new block of memory touched by the traced program increases the size of the AVL tree by one node. The resulting memory usage is always non-decreasing, and no dynamically allocated objects are ever freed except at the end of a run. We therefore consider it reasonable to use one of our other test programs to generate a reference trace, without fearing that this would introduce correlated behavior. (The resulting fragmentation at peak memory usage is insensitive to the input trace, despite the fact that total memory usage depends on the number of memory blocks referenced in the trace.)

gives examples of trace-driven memory system studies that date back to 1966. Trace-driven memory simulation typically consists of three stages:

1. *Trace collection* is the process of recording the exact sequence of memory references (instruction and data) of a program. A modern computer can generate several hundred million trace elements per second.

2. *Trace reduction* is the process of reducing these trace elements to a more manageable number, and/or selecting the events of interest in the simulation (e.g., the data loads and stores for the simulation of a data cache).

3. *Trace processing* is the process of using the reduced trace to simulate the part of the computer system under study.

We used trace-driven memory simulation in this research for both our fragmentation studies and our locality studies (Chapter 3). For our fragmentation studies, we collected and processed traces of the malloc, realloc, and free calls of our test programs by using a specially modified memory allocator that recorded these events to disk as the test programs ran. For our locality studies, we collected and processed the data loads and stores of our test programs by using the Shade trace gathering tool [CK93]. These traces were processed on-line by piping directly from Shade to the processing tools (see Section 3.5).

2.8 Experimental Design

A goal of this research is to measure the true fragmentation costs of particular memory allocation *policies* independently of their *implementations*. In this section we will describe how we achieved this goal.

The first step was to write substitutes for malloc, realloc, and free that perform the basic malloc functions and, as a side-effect, create a trace of the memory

allocation activity of the program. This trace is made up of a series of records, each containing:

- the type of operation performed (malloc, realloc, free),
- the memory location affected (for malloc, this was the memory location returned by malloc; for realloc and free, this was the memory location passed by the application), and
- the number of bytes requested (for free, this was 0).

The second step was to build a trace processor that reads a trace and produces basic statistics about the trace:

- the number of objects allocated,
- the number of bytes allocated,
- the average object size,
- the maximum number of *bytes* live at any one time for the entire trace, and
- the maximum number of *objects* live at any one time for the entire trace.

The third step was to build a trace processor that reads a trace and calls `malloc`, `realloc`, and `free` of an implementation of the allocation policy under study. We modified each of these allocators to keep track of the total number of bytes requested from the operating system. With this information, and the maximum number of live bytes for the trace, we can determine the fragmentation for a particular program using a particular *implementation* of a memory allocation policy.

However, as we will discuss in the next few subsections, this is *not* a good measure of the actual fragmentation caused by the policy, but instead reflects many artifacts of the implementation. We will present the results for this most simple

approach, and then we will remove each of the artifacts, showing how each affected our experimental results, until we finally arrive at numbers that measure just policy considerations. We will present numbers averaged across all eight of our test programs. The interested reader can see Appendix A for the results of the individual test programs.

Note that we express fragmentation in terms of percentages over and above the amount of live data, i.e., increase in memory usage, not the percentage of actual memory usage that is due to fragmentation. (The baseline is therefore what might result from a perfect allocator that could somehow achieve zero fragmentation.)

2.8.1 Our Measure of Time

Throughout this chapter when we talk about time (unless otherwise specified), we measure time normalized to the rate of allocation. Thus, if we say that something takes one megabyte to happen, we mean that one megabyte of memory has been allocated between the beginning and the end of the event. We believe that this is a more interesting measure of time than standard wall-clock time because it normalizes time to something in which we are interested: namely the rate of allocation. In other words, when we are talking about memory fragmentation, a program that allocates a lot of memory in short bursts with long time periods (in wall-clock) of no memory allocation in between is just as interesting than a program that allocates the same amount of memory slowly and steadily.

2.8.2 Our Measure of Fragmentation

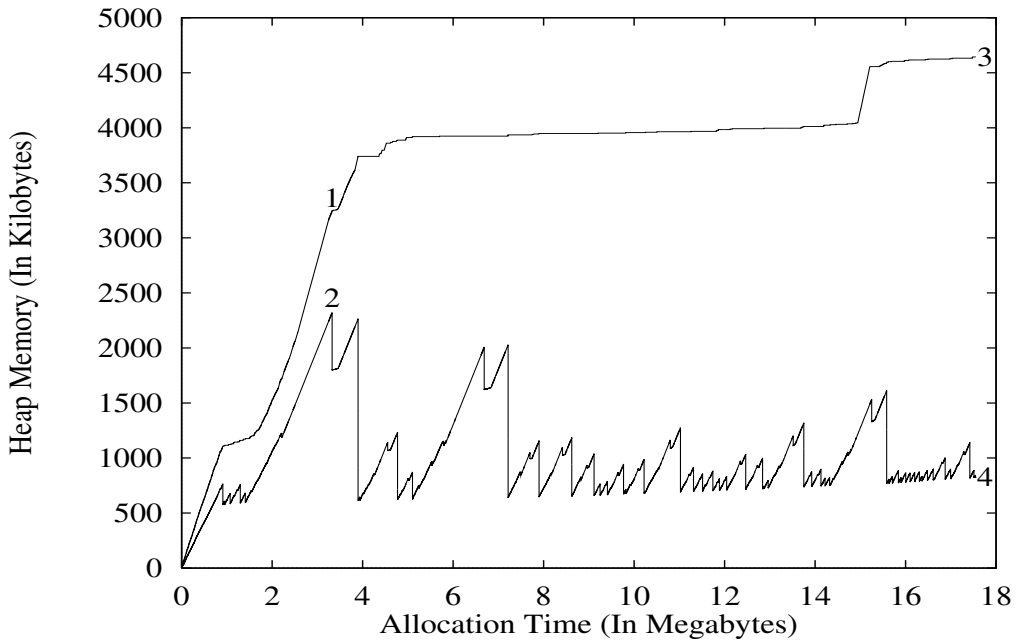

Figure 2.1: Measurements of fragmentation for GCC using simple segregated 2^N (top line: memory used by allocator; bottom line: memory requested by allocator)

There are a number of legitimate way to measure fragmentation. We use Figure 2.1 to illustrate four of these, and to explain the method we chose to use. Figure 2.1 is a trace of the memory usage of the GCC compiler, compiling the combine.c program, using the simple segregated 2^N allocator. The lower line is the amount of memory requested by GCC (in kilobytes) which is currently live. The upper line is the amount of memory actually used by the allocator to satisfy GCC's memory requests.

The four ways to measure fragmentation for a program which we considered are:

1. The amount of memory used by the allocator relative to the amount of memory requested by the program, *averaged across all points in time*: In Figure 2.1,

this is equivalent to averaging the fragmentation for each corresponding point on the upper and lower lines for the entire run of the program. For the GCC program using the simple seg 2^N allocator, this measure yields 258% fragmentation. The problem with this measure of fragmentation is that it tends to hide the spikes in memory usage, and it is at these spikes where fragmentation is most likely to be a problem.

2. The amount of memory used by the allocator relative to the maximum amount of memory requested by the program *at the point of maximum live memory*: In Figure 2.1 this corresponds to the amount of memory at point 1 relative to the amount of memory at point 2. For the GCC program using the simple seg 2^N allocator, this measure yields 39.8% fragmentation. The problem with this measure of fragmentation is that the point of maximum live memory is usually not the most important point in the run of a program. The most important point is likely to be a point where the allocator must request more memory from the operating system.

3. The maximum amount of memory used by the allocator relative to the amount of memory requested by the program *at the point of maximal memory usage*: In Figure 2.1 this corresponds to the amount of memory at point 3 relative to the amount of memory at point 4. For the GCC program using the simple seg 2^N allocator, this measure yields 462% fragmentation. The problem with this measure of fragmentation is that it will tend to report high fragmentation for programs that use only slightly more memory than they request if the extra memory is used at a point where only a minimal amount of memory is live.

4. The maximum amount of memory used by the allocator relative to the maximum amount of live memory: *These two points do not necessarily occur at the same point in the run of the program.* In Figure 2.1 this corresponds to the

amount of memory at point 3 relative to the amount of memory at point 2. For the GCC program using the simple seg 2^N allocator, this measure yields 100% fragmentation. The problem with this measure of fragmentation is that it can yield a number that is too low if the point of maximal memory usage is a point with a small amount of live memory and is also the point where the amount of memory used becomes problematic.

We chose the last of these definitions: the maximum amount of memory used by the allocator relative to the maximum amount of memory requested by the program (points 3 and 2). This measure of fragmentation indicates how much memory is required to run a given program. However, the other measures of fragmentation are also interesting, and deserve future study. Unfortunately, there is no right point at which to measure fragmentation. If fragmentation appears to be a problem for a program, it is important to identify the conditions under which it is a problem and measure the fragmentation for those conditions. For many programs, although fragmentation will not be a problem at all, allocation policy is still important because allocator placement choices can have a dramatic effect on locality (as we show in Chapter 3).

2.8.3 Experimental Error

In this research, we worked very hard to remove as much measurement error as possible. In this section, we will describe the error which remains.

The most important experimental error comes from the way our allocators request memory from the operating system (using the `sbrk` UNIX system call). Most of our allocators request their memory in 4K byte blocks. Thus, any measurement of the heap size of a program using a particular allocator can be an over-estimate by as much as 4K bytes. This error is even larger for four of our allocators (double buddy 5K, double buddy 10K, simple segregated 2^N, and simple segregated 2^N &

$3 * 2^N$). However, for our final numbers, after factoring out all overheads (Section 2.8.7), this error is just 256 bytes.

The double-buddy allocators (double buddy 5K and double buddy 10K) each request memory from the operating system in two different sizes. The double-buddy 5K allocator requests memory from the operating system in 4K and 6K sizes, yielding an average size of 5K. The double-buddy 10K allocator requests memory from the operating system in 8K and 12K sizes, yielding an average size of 10K. Thus, these allocators can yield an over-estimate of the memory used of up to 5K and 10K respectively (320 bytes and 640 bytes for our final numbers).

The simple segregated storage allocators (simple seg 2^N and simple seg 2^N & $3 * 2^N$) both perform *no* coalescing. *Each size class* can contribute to an over-estimate by as much as 4K bytes. Thus, for the simple seg 2^N, and the simple seg 2^N & $3 * 2^N$ allocators, the measure of the amount of memory used can be an over-estimate by as much as 4K times the number of size classes, which is roughly $4K * \ln(largest\_size - smallest\_size)$, and $4K * 2 * \ln(largest\_size - smallest\_size)$ (one sixteenth of this value for our final numbers).

2.8.4 Our Use of Averages

In this dissertation, we follow [FW86, PH96] when we present averages. If the numbers being averaged are simple numbers, such as the fragmentation of a program given a particular allocator, we use the arithmetic mean. If the numbers being averaged are normalized to some consistent reference, such as the fragmentation of a given allocator normalized to the fragmentation of best fit, then we use the geometric mean. Finally, if the numbers being averaged are rates, such as a cache miss rate, then we use the harmonic mean. In all cases, the averages are *unweighted.*

Allocator name	% Waste	Allocator name	% Waste
first fit AO 8K	34.65%	multi-fit min	34.37%
best fit AO 8K	34.47%	next fit AO	38.32%
best fit FIFO	33.41%	next fit AO no footer	26.86%
best fit FIFO no footer	22.44%	next fit AO def AO	38.82%
best fit AO	33.41%	next fit FIFO	42.60%
best fit AO no footer	22.49%	next fit FIFO no footer	31.52%
Lea 2.6.1	23.58%	Lea 2.5.1	41.83%
best fit LIFO	33.54%	Lea 2.5.1 no footer	30.94%
best fit LIFO no footer	22.44%	next fit AO def LIFO	40.34%
first fit AO	33.17%	next fit AO def FIFO	43.41%
first fit AO no footer	22.14%	next fit LIFO def LIFO	56.28%
best fit LIFO split-7	33.66%	first fit LIFO def LIFO	57.40%
best fit LIFO split-14	34.02%	double buddy 5K	46.22%
first fit AO def AO	32.46%	double buddy 10K	46.18%
first fit AO def LIFO	33.58%	next fit LIFO WPH	66.37%
first fit FIFO	33.86%	first fit LIFO	66.25%
first fit FIFO no footer	22.83%	first fit LIFO no footer	56.40%
best fit AO def AO	32.46%	first fit LIFO split-7	67.07%
first fit AO def FIFO	34.61%	first fit LIFO split-14	67.35%
best fit LIFO def AO	32.46%	next fit LIFO	71.78%
best fit LIFO def LIFO	33.90%	next fit LIFO no footer	58.86%
best fit LIFO def FIFO	34.61%	next fit LIFO split-7	70.07%
best fit AO def LIFO	33.90%	next fit LIFO split-14	70.53%
best fit AO def FIFO	34.61%	binary buddy	74.11%
multi-fit max	35.32%	simple seg 2^N & $3 * 2^N$	72.54%
next fit AO 8K	39.28%	simple seg 2^N	84.81%
half fit	33.88%		
Average:			42.50%

Table 2.2: Percentage waste for all allocators averaged across all programs

2.8.5 Total Memory Usage

In Table 2.2 we present the amount of memory wasted by each of our allocators, as a percentage of the amount of live data at peak memory usage (the allocators are sorted from lowest fragmentation to highest fragmentation *for our final experiments* (Table 2.4)). The column labeled "Waste" shows the amount of fragmentation for our implementation of each allocation policy. Here, we can see that the best fit (LIFO, FIFO, and AO) no footer, first fit (FIFO and AO) no footer, next fit AO no footer, and Lea's 2.6.1 allocators all perform relatively well (less than 30% average fragmentation), particularly compared to the average of 42.50% waste.

However, what we want to measure is the fragmentation of the *policy*, and not the *implementation*. In particular, some of these implementations use footers on the objects, and some do not. Additionally, some of these policies can be easily implemented without any headers or footers at all. So, the next step is to account for header and footer overhead to avoid introducing implementation artifacts into our measurements.

2.8.6 Accounting for Headers and Footers

To account for the cost of headers and footers in the implementation of our allocator policies, we modified each memory allocator to tell our simulator how many bytes it had dedicated to header and footer information. We were then able to use the fact that the minimum object size for all of our allocators was 16 bytes (no allocator used more than 16 bytes for its internal data structures) to account for this overhead in the following way: for each malloc or realloc request that our simulator processed, it asked the allocator for the number of bytes in the trace minus the number of bytes in the header and footer for the allocator being simulated. We are able to do this because we are only simulating the program (from an actual trace), and the memory allocated is unused.

Allocator name	% Frag	Allocator name	% Frag
first fit AO 8K	16.91%	multi-fit min	17.62%
best fit AO 8K	16.24%	next fit AO	18.55%
best fit FIFO	14.36%	next fit AO no footer	18.55%
best fit FIFO no footer	14.36%	next fit AO def AO	21.50%
best fit AO	14.36%	next fit FIFO	24.97%
best fit AO no footer	14.36%	next fit FIFO no footer	24.97%
Lea 2.6.1	14.45%	Lea 2.5.1	25.48%
best fit LIFO	14.45%	Lea 2.5.1 no footer	25.48%
best fit LIFO no footer	14.45%	next fit AO def LIFO	24.64%
first fit AO	14.41%	next fit AO def FIFO	24.28%
first fit AO no footer	14.43%	next fit LIFO def LIFO	41.07%
best fit LIFO split-7	14.71%	first fit LIFO def LIFO	43.58%
best fit LIFO split-14	15.99%	double buddy 5K	42.15%
first fit AO def AO	15.23%	double buddy 10K	41.49%
first fit AO def LIFO	13.72%	next fit LIFO WPH	47.31%
first fit FIFO	14.70%	first fit LIFO	49.49%
first fit FIFO no footer	17.52%	first fit LIFO no footer	47.40%
best fit AO def AO	14.51%	first fit LIFO split-7	48.57%
first fit AO def FIFO	15.85%	first fit LIFO split-14	49.41%
best fit LIFO def AO	14.42%	next fit LIFO	51.81%
best fit LIFO def LIFO	14.39%	next fit LIFO no footer	51.81%
best fit LIFO def FIFO	15.85%	next fit LIFO split-7	51.58%
best fit AO def LIFO	14.58%	next fit LIFO split-14	52.54%
best-fit AO def FIFO	17.08%	binary buddy	62.35%
multi-fit max	16.21%	simple seg 2^N & $3 * 2^N$	72.54%
next fit AO 8K	20.44%	simple seg 2^N	84.81%
half fit	14.39%		
Average:			27.85%

Table 2.3: Percentage fragmentation (accounting for headers and footers) for all allocators averaged across all programs

In Table 2.3, we present the fragmentation for each of our allocators with header and footer costs removed. Note that now the best allocators all have around one half of the fragmentation of the average allocator, and that the best allocators all have around 15% fragmentation.

These numbers seem pretty good. Many people would be happy if their memory allocator only wasted an average of 15% of the heap memory due to fragmentation. However, for some applications, even 15% is too much memory to waste. So, this leads to the question: "can we develop a policy that can do even better?" As we will see after we account for the last overhead, *for the measure of fragmentation that we chose* the answer is no.

2.8.7 Accounting for Minimum Alignment and Object Size

All modern hardware requires that objects follow some form of alignment constraints. Some hardware, such as the Sparc architecture, requires that double floating point values be aligned on 8-byte boundaries (e.g., memory location 0, 8, 16, etc., but not memory locations 4, 12, 20, etc.). Since our allocation policies were implemented and tested on Sparc machines, they all obey this 8-byte alignment constraint. In fact, *no allocator can avoid this cost* on this machine.

An additional overhead of our implementations is that the minimum object size is 16 bytes. So, even if the program asked for a mere 1 byte of memory, in all cases it got 16 bytes. This overhead is strictly an implementation cost, and not a policy cost.

To account for these overheads, we multiplied every malloc/realloc request by 16, and then divided the final amount of heap memory used by 16, to account for the factor of 16 in the request sizes. Because all allocation requests are now multiples of 16, and the smallest request is 16 bytes, the allocator need do no rounding of memory requests. This leads us to the results in Table 2.4.

2.9 Actual Fragmentation Results

In Table 2.4, we see that the two best allocation policies, first fit addressed-ordered free list with 8K allocation, and best fit addressed-ordered free list with 8K allocation, both suffer from less than 1% actual fragmentation. This is more than *17 times* better than the average allocator, and more than *88 times* better than the worst allocator. In addition, 25 of our allocators had less than 5% actual fragmentation. The worst of our allocators, those having over 50% fragmentation, tried to trade increased internal fragmentation for reduced external fragmentation, and did not coalesce all possible blocks, giving further evidence that this is not a good policy decision. If these results hold up to further study with additional programs we arrive at a startling conclusion: *fragmentation is a solved problem, and it has been solved for over 30 years.*

In terms of rank order of allocator policies, these results contrast with traditional simulation results, where best fit usually performs well but is sometimes outperformed by next fit (e.g., in Knuth's small but influential study [Knu73]). In terms of practical application, we believe this is one of our most significant findings. Since segregated fit (as exemplified by Lea's 2.6.1 allocator) implements an approximation of best fit fairly efficiently, it shows that a reasonable approximation of a best-fit policy is both desirable and achievable.

2.9.1 Fragmentation for Selected Allocators for Each Trace

Table 2.5 shows the percentage actual fragmentation for each of the selected allocators, for each trace. The complete table of percentage actual fragmentation for all allocators, for each trace, can be seen in Appendix A. It is particularly interesting to note how high the standard deviation is for first fit LIFO and next fit LIFO. These allocators actually perform quite well on two of our test programs: Hyper and LRUsim. However, they perform disastrously on one program: Ghostscript. At the

Allocator name	% Frag	Allocator name	% Frag
first fit AO 8K	0.77%	multi-fit min	6.38%
best fit AO 8K	0.83%	next fit AO	8.04%
best fit FIFO	2.23%	next fit AO no footer	8.04%
best fit FIFO no footer	2.23%	next fit AO def AO	16.60%
best fit AO	2.27%	next fit FIFO	18.37%
best fit AO no footer	2.27%	next fit FIFO no footer	18.37%
Lea 2.6.1	2.27%	Lea 2.5.1	19.38%
best fit LIFO	2.30%	Lea 2.5.1 no footer	19.38%
best fit LIFO no footer	2.30%	next fit AO def LIFO	19.52%
first fit AO	2.30%	next fit AO def FIFO	21.03%
first fit AO no footer	2.30%	next fit LIFO def LIFO	29.82%
best fit LIFO split-7	2.41%	first fit LIFO def LIFO	32.54%
best fit LIFO split-14	3.03%	double buddy 5K	34.25%
first fit AO def AO	3.10%	double buddy 10K	34.27%
first fit AO def LIFO	3.10%	next fit LIFO WPH	34.64%
first fit FIFO	3.14%	first fit LIFO	36.24%
first fit FIFO no footer	3.14%	first fit LIFO no footer	36.24%
best fit AO def AO	3.79%	first fit LIFO split-7	36.59%
first fit AO def FIFO	3.91%	first fit LIFO split-14	38.11%
best fit LIFO def AO	3.98%	next fit LIFO	38.45%
best fit LIFO def LIFO	4.53%	next fit LIFO no footer	38.45%
best fit LIFO def FIFO	4.70%	next fit LIFO split-7	39.05%
best fit AO def LIFO	4.72%	next fit LIFO split-14	39.38%
best fit AO def FIFO	4.94%	binary buddy	53.35%
multi-fit max	5.40%	simple seg 2^N & $3 * 2^N$	61.50%
next fit AO 8K	5.55%	simple seg 2^N	73.61%
half fit	6.01%		
Average:			16.96%

Table 2.4: Percentage fragmentation (accounting for headers, footers, minimum object size, and minimum alignment) for all allocators averaged across all programs

Allocator	Espresso	GCC	Ghost	Grobner	Hyper	Perl	P2C	LRUsim	Average	Std Dev
Lea 2.6.1	0.26%	0.33%	3.40%	2.03%	0.16%	9.98%	1.78%	0.26%	2.27%	3.32%
best fit LIFO NF	0.26%	0.50%	3.40%	2.03%	0.16%	9.98%	1.78%	0.26%	2.30%	3.31%
first fit AO NF	0.26%	0.50%	3.40%	2.03%	0.16%	9.98%	1.78%	0.26%	2.30%	3.31%
half fit	9.37%	17.4%	1.60%	7.54%	0.16%	9.98%	1.78%	0.26%	6.01%	6.14%
double buddy 5K	68.6%	31.3%	20.8%	12.3%	50.0%	24.9%	32.4%	33.7%	34.3%	17.7%
first fit LIFO NF	9.37%	23.7%	179%	62.7%	0.16%	9.98%	4.83%	0.26%	36.2%	61.2%
next fit LIFO NF	9.37%	21.0%	200%	51.7%	0.16%	9.98%	15.0%	0.26%	38.5%	67.3%
binary buddy	45.8%	34.1%	38.4%	36.9%	99.9%	40.0%	55.0%	77.7%	53.4%	23.6%
simp seg $3 * 2^N$	159%	99.9%	32.4%	41.0%	26.0%	33.9%	56.2%	43.2%	61.5%	45.9%
simp seg 2^N	162%	99.5%	39.0%	57.4%	26.0%	51.2%	74.5%	79.0%	73.6%	42.8%
Average	46.5%	32.8%	52.1%	27.6%	20.3%	20.9%	24.5%	23.5%		
Std Dev	64.3%	37.3%	74.1%	24.9%	32.9%	15.4%	28.0%	32.9%		

Table 2.5: Percentage actual fragmentation for selected allocators for all traces

same time, the best fit LIFO no footer, first fit address ordered no footer, and Lea's 2.6.1 allocators all perform quite well on all of the test programs. Perl is the only program for which they have any real fragmentation (10%), and that program only has 70K bytes maximum live data. Because all of the allocators allocate memory in 4K chunks, we have a potential error of 4K in our measurements. Thus, most of the 10% fragmentation could be measurement error.

The next important question is: "are the differences in Table 2.5 statistically significant?" Table 2.6 shows the t-test results for the values in Table 2.5. To find the probability that one allocator *really* performs better than another, find the row for one of the allocators and the column for the other. The value at the intersection point is the probability that the allocator with the lower fragmentation really has lower fragmentation than the other allocator.[26]

From Table 2.6, we can conclude with 90% confidence that the best fit LIFO, first fit address ordered, and Lea's 2.6.1 allocators all perform better than binary buddy, double buddy 5K, first fit LIFO, next fit LIFO, half fit, simple segregated storage 2^N, and simple segregated storage 2^N & $3 * 2^N$. We can not, however, conclude at the 90% confidence level that there is any difference between the performance of the best fit LIFO, first fit address ordered, and Lea's 2.6.1 allocators.

2.9.2 Policy Variations

We will now discuss how the different policy variations affected the actual fragmentation results as reported in Table 2.4. One interesting result is that no version of best fit had more than 5% actual fragmentation. This is also true for all versions of first fit that used an address-ordered free list, and the two versions of first fit that used a FIFO free list. This strongly suggests that the basic best-fit algorithm and the first-fit algorithm with an address-ordered free list are very robust algorithms.

[26]This interpretation of the t-test comes from [Fre84]. The values in Table 2.6 were computed using Microsoft Excel version 7.0's ttest function with paired samples and a single tail distribution.

	binary buddy	best fit LIFO NF	double bdy 5K	first fit AO NF	first fit LIFO NF	half fit	Lea's 2.6.1	next fit LIFO NF	simple seg 2^N
binary buddy	***								
best fit LIFO NF	99.97%	***							
double bdy 5K	97.48%	99.89%	***						
first fit AO NF	99.97%	50.00%	99.89%	***					
first fit LIFO NF	73.18%	92.08%	53.03%	92.08%	***				
half fit	99.90%	92.62%	99.81%	92.62%	89.37%	***			
Lea 2.6.1	99.97%	82.47%	99.89%	82.47%	92.10%	92.59%	***		
next fit LIFO NF	69.33%	91.47%	55.92%	91.47%	73.16%	88.80%	91.48%	***	
simp seg 2^N	83.48%	99.87%	99.24%	99.87%	87.22%	99.90%	99.87%	84.50%	***
simp seg $3 * 2^N$	64.88%	99.53%	96.40%	99.53%	78.52%	99.60%	99.53%	75.24%	98.64%

Table 2.6: Probability that the difference between allocator performance is statistically significant

In addition, it suggests that for these two basic policies the other variations in policy (for best fit, order of free list; for first fit address ordered, immediate or deferred coalescing) do not matter, and should only be considered if they make the implementation more efficient. Because we only have one variation on first fit FIFO (the second first fit FIFO allocator only removed footer costs, which have been removed in the actual fragmentation tests anyway) we cannot make any claims about the robustness of this policy. However, its performance indicates that this policy should be studied in more detail.

A second interesting result is that only three versions of next fit had less than 10% actual fragmentation, and all of those versions used an address ordered free list. This, combined with the observations for first fit, strongly suggests that an address-ordered free list is a very good policy for reducing fragmentation.[27] In addition, these results show that next fit is a poor policy, and should be avoided. Finally, we can see from Table 2.4 that buddy systems and segregated storage systems suffer from considerable fragmentation.

A third interesting result is that for *good allocation policies*, deferred coalescing appears not to increase fragmentation much. For best fit with a LIFO free list, the highest average fragmentation when using deferred coalescing was 4.70% (for a FIFO ordered quick list). While this is more than twice the fragmentation of the immediate coalescing version of this allocator, it is still very acceptable for most applications. For first fit with an address-ordered free list, the highest average fragmentation when using deferred coalescing, which occurred with a FIFO ordered quick-list, was 3.91%. This compares to 2.3% fragmentation when using immediate coalescing. Finally, for next-fit with an address-ordered free list, the highest frag-

[27] Recall that an address-ordered free list can be cheap to implement if a bit-map is used to indicate which memory locations are allocated. Just because a good policy appears expensive to implement, it should not be discarded because of this concern alone. Often further thought can reveal an efficient implementation of the desirable policy. This is why it is so important to separate policy from mechanism.

mentation when using deferred coalescing, which also occurred with a FIFO ordered quick-list, was 21.03%. This compares to 8.04% fragmentation with the immediate coalescing version of this allocator, and is a further indication of the instability of the next-fit policy. However, because of the small number of programs studied, none of these three differences is statistically significant at the 85% confidence level.

Because programs tend to allocate many objects of exactly the same size (see Section 2.12), this is an important result suggesting that coalescing costs need not be much of a concern, and that deferred coalescing can provide a substantial benefit with little cost in terms of fragmentation. However, the low statistical significance of these differences is an indication that more programs must be studied to determine the true cost of deferred coalescing.

A fourth interesting result is that simple segregated storage 2^N & $3*2^N$ significantly outperforms simple segregated storage 2^N, even though simple segregated storage 2^N & $3*2^N$ has twice as many size classes as simple segregated storage 2^N.[28] Also notice that the binary-buddy allocator suffers from much more fragmentation than the double-buddy allocators. Again, the double-buddy allocators have size classes which are twice as precise as the binary-buddy allocators. We believe that this is evidence that very coarse size classes generally lose more memory to internal fragmentation than they save in external fragmentation.

2.10 A *Strategy* That Works

Up until this point, we have been talking about the importance of separating *policy* from *mechanism*. There is yet a third consideration that is important to separate: *strategy*. In Section 2.9.2, we saw that there are several policies that result in low

[28] Recall that in Section 2.8.3 we said that neither of the simple segregated storage allocators coalesce memory, and that the simple segregated storage 2^N & $3*2^N$ allocator has twice as many size classes as the simple segregated storage 2^N allocator. Thus, the simple segregated storage 2^N & $3*2^N$ allocator will over-estimate the total amount of memory used by about twice as much as the simple segregated storage 2^N allocator.

fragmentation. The question is: "are these policies in some way related?" In other words, is there some underlying strategy to allocating memory that will lead to policies that usually provide low fragmentation? We believe that there is such a strategy, and that when this strategy is understood, it will lead to new policies that expose even more efficient implementations.

All of the policies that performed well in our studies share two common traits: they all immediately coalesce memory, and they all preferentially reallocate objects that have died recently over those that died further in the past.[29] In other words, they all give some objects more time to coalesce with their neighbors, yielding larger and larger contiguous free blocks of memory. These in turn can be used in many ways to satisfy future requests for memory that might otherwise result in high fragmentation. In the following paragraphs, we will analyze each memory allocation policy that performs well to show how it fits into this strategy.

The best-fit policy tries to preferentially use small free blocks over large free blocks. This characteristic gives the neighbors of the large free blocks more time to die and be merged into yet larger free blocks, which, in turn, makes them even less likely that best fit will allocate something out of these larger free blocks. The cycle continues until there are only a few very large areas of contiguous free memory out of which to allocate free blocks. When one of these free blocks is used for memory allocation, a small piece is split out of it, making it somewhat smaller, which will make it more likely that that same free block will be used for subsequent memory requests, saving the other larger free areas for later needs.

Using address-ordered free lists, which worked so well for first fit and next fit, can be viewed as a variation on this same theme. Blocks at one end of memory

[29] An important exception is the first-fit FIFO free list allocator. This allocator performed remarkably well, and does not preferentially reallocate objects that have died recently over those that died further in the past. We do not know if this indicates that there is a different effective strategy at work, or if this is evidence that our suggestion of a good strategy is not correct. Clearly, more study is needed on this allocator.

Program name	90%	99%	99.9%	Total allocation time
GCC	1K	2,409K	17,807	18,404K
Espresso	1K	8K	57K	106,893K
Ghostscript	1K	40,091K	48,593K	50,170K
Grobner	2K	3,311K	3,939K	4,082K
Hyper	2K	12K	18K	7,556K
P2C	11K	3,823K	4,494K	4,753K
Perl	1K	11K	184K	33,834K
LRUsim	1K	1K	1K	1,431K
Average	2.5K	6,208K	9,387K	28,390K

Table 2.7: Time before given % of free *objects* have both temporal neighbors free

are used preferentially over blocks at the other end. This gives objects at the end of memory from which new blocks are *not* being allocated more time to die and merge with their neighbors. Note, however, that this theme is much stronger with first fit address ordered than with next fit address ordered. We believe this is why first fit address ordered performs much better than next fit address ordered.

In both best fit and first fit address ordered, objects allocated at about the same time tend to be allocated from contiguous memory. In the case of best fit, this is because once a block is split, its remainder is smaller, making it a better fit for the next request. In the case of first fit address ordered, this is because blocks tend to be allocated out of memory at one end of the heap.

2.11 Objects Allocated at the Same Time Tend to Die at the Same Time

The tendency of best fit and first fit address ordered to place blocks allocated at about the same time in contiguous memory may interact favorably with another observation about our test programs: objects allocated at about the same time tend to die at about the same time.

Table 2.7 shows the amount of time (in terms of bytes allocated: see Section

Program name	90%	99%	99.9%	Total allocation time
GCC	223K	2,355K	17,805K	18,404K
Espresso	1K	62K	9,552K	106,893K
Ghostscript	14K	44,876K	48,752K	50,170K
Grobner	2K	2,464K	3,836K	4,082K
Hyper	1K	11K	16K	7,556K
P2C	16K	4,142K	4,614K	4,753K
Perl	1K	13K	7,153K	33,834K
LRUsim	1K	1K	8K	1,431K
Average	32K	6,740K	11,467K	28,390K

Table 2.8: Time before given % of free *bytes* have both temporal neighbors free

2.8.1) before 90%, 99%, and 99.9% of all *objects* have both of their temporal neighbors free (those objects allocated just before and just after the given object). On average, after just 2.5K of allocation 90% of all objects have both of their temporal neighbors free. Thus, if we allocate blocks from contiguous memory regions, waiting just a short time after an object becomes free before allocating the memory again, then most of the time its neighbors will also be free and can be coalesced into a larger free block.

Table 2.8 shows the same information as Table 2.7, except weighted by the *size* of the objects becoming free. Thus, the table shows how long (in allocation time) before 90%, 99%, and 99.9% of the *bytes* allocated can be coalesced with neighboring memory. Here, we see that if we wait for just 32K of allocation, 90% of all memory allocated can be coalesced with its neighboring memory.

Thus, whether we measure in bytes or objects, the vast majority of all objects allocated at around the same time also die at around the same time.

2.12 Programs Tend to Allocate Only a Few Sizes

For most programs, the vast majority of objects allocated are of only a few sizes. Table 2.9 shows the number of object sizes represented by 90%, 99%, 99.9%, and

Program	90%	99%	99.9%	100%	Total Objects
GCC	5	12	254	641	721,353
Espresso	9	95	308	758	1,672,889
Ghostscript	7	85	344	589	566,542
Grobner	12	55	100	139	163,310
Hyper	1	2	2	6	108,720
LRUsim	1	1	5	21	39,103
P2C	4	26	58	92	194,997
Perl	10	27	60	99	1,600,560
Average	6	38	141	293	628,551

Table 2.9: Number of object sizes representing given percent of all object sizes

100% of all objects allocated. The last column is the total number of objects allocated by that program. On average, 90% of all objects allocated are of just 6.12 different sizes, 99% of all objects are of 37.9 sizes, and 99.9% of all objects are of 141 sizes.

The reason that most objects allocated are of so few object sizes is that, for most programs, the majority of dynamic objects are of just a few types. These types often make up the nodes of large or common data structures upon which the program operates. The remaining object sizes are accounted for by strings, buffers, and single-use objects.

A good allocator should try to take advantage of the fact that, for most programs, the majority of all objects allocated are of only a few sizes. We believe that this is part of the reason that the buddy systems and simple segregated storage policies have so much fragmentation. These policies increase internal fragmentation to try to reduce external fragmentation. As we can see from Table 2.9, this is unnecessary. The vast majority of dynamic memory requests are for objects of *exactly* the same size as recently freed objects, and there is no need to worry about the next memory request being for a block that is just a little larger than any free region.

2.13 Small Policy Variations Can Lead to Large Fragmentation Variations

A result of particular importance to anyone presenting research in memory allocation algorithms is that seemingly small variations in policy can lead to large variations in fragmentation. In Table 2.4 we saw that the difference in fragmentation between next fit address ordered and next fit LIFO is 478%. The difference between first fit address ordered with memory requested from the operating system in 8K chunks and first fit LIFO is a staggering 4,706%. It is therefore very important, when presenting memory allocation research results, to carefully describe the algorithm being studied.

2.14 A View of the Heap

To further validate the idea that best fit and first fit address ordered work well because they allow large contiguous areas in the heap to become free, we wrote a program that generates an image of the heap over time. In the pictures that follow, the X-axis is allocation time, and the Y-axis is the heap (going from low heap addresses to high heap addresses). For any given pixel on the graph, the darkness represents the percentage of that portion of the heap at that interval in time which is allocated. So, a black pixel is 100% allocated, and a white pixel is 100% free. A gray pixel is somewhere in between, depending on its darkness.

In what follows, we will show and discuss allocation graphs for a subset of the eight test programs, and nine selected allocators (binary buddy, best fit LIFO, first fit address ordered, first fit LIFO, half fit, Lea's 2.6.1, next fit LIFO, simple segregated storage 2^N, and simple segregated storage 2^N & $3 * 2^N$ — we do not present allocation graphs for double buddy 5K because our implementation of this allocator made interpreting these graphs difficult). In addition, we present graphs

of the memory usage of a special allocator that we call a "linear allocator." This allocator allocates all of its memory sequentially, and never reuses freed memory. The allocation graphs of this allocator give us an indication of the natural fragmentation inherent in the trace. By comparing graphs for this allocator to those of the other allocators, we can get a better idea of how the different allocation policies interact with each trace.

These pictures correspond to the actual fragmentation numbers from Table 2.4. In other words, all header, footer, minimum object size, and alignment costs have been removed. Thus, these are graphs of memory use of the *policies*, and not the allocator *implementations*. The complete set of allocation graphs for the eight programs and nine selected allocators can be found in Appendix C.

2.14.1 GCC Allocation Graphs

The first ten pictures (Figures 2.2 to 2.11) are of the gnu C compiler, compiling the file combine.c (part of the GCC distribution). As can be seen in the plots, this program exhibits very strong phase behavior, with two particularly large data structures freed at allocation time 4 and 7 megabytes. The horizontal lines running across the plot are objects that remain live after the data structures are freed (presumably, the results of some computation involving the data structure). Figure 2.2 is the plot of the linear allocator. In this plot, the strong phase behavior of the GCC compiler is shown as triangular features.

As can clearly be seen in Figures 2.5, 2.7, 2.9, 2.10, and 2.11, the reuse of memory after the first data structure becomes free (at around allocation time 4 megabytes) critically influences later fragmentation. In Figures 2.4, 2.5, and 2.8 memory in the lower address range is aggressively reused, allowing for very large free areas in the upper address range. Thus, at later times (particularly for the large data structure allocated between times 5.5 and 7 megabytes), this memory can be