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.

[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.

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 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 allocation 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 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]

[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.

[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

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.

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:

---

this paper also found it surprising that no one had done work in this area before.

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 multi-media) 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 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.

[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.

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

[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.

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 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.

[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.

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

[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

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.

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]

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.

with real data.)

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

[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.

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 (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

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

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

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

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

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

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 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.

[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].

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 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.")

- *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

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

[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).

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.

[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.

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 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.

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.

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

- *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 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.

---

[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.

[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).

[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.

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]

- *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]

[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.

[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.

which grows monotonically. A new entry 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] 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

[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.)

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

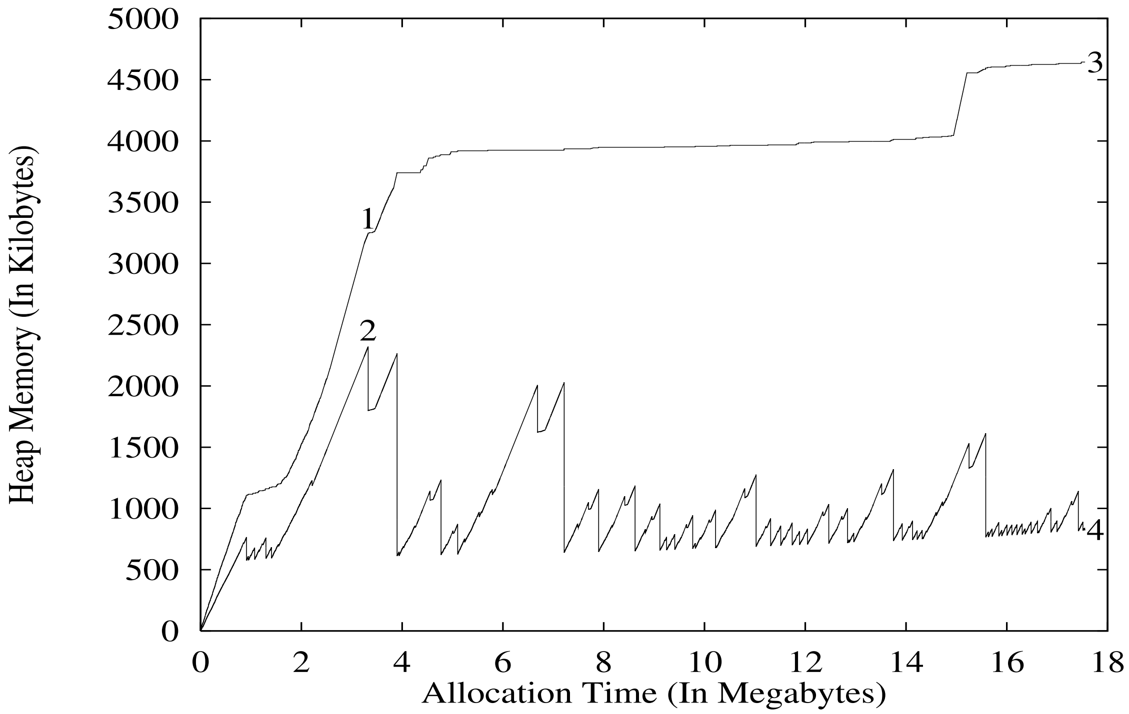

Figure 2.1: Measurements of fragmentation for GCC using simple segregated 2^N (top line: memory used by allocator; bottom line: memory requested by allocator)

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

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

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.

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

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

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 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. 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

[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.

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

	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

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 fragmentation 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.

[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.

[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.

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 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 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.

[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

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

2.11 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 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

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

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 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 easily reused. We believe that this is an indication of our strategy at work.

Figures 2.10 and 2.11 show an interesting side effect of simple segregated storage's basic policy: that of not coalescing memory. Objects allocated at 15 megabytes cause the policy to request more memory from the operating system even though there are large free areas available. This is because these regions of memory are already dedicated to objects of a different size than the one being requested. A variation on this policy that reuses entirely free pages of memory would not suffer from this particular problem, but would still perform poorly for the data structure created between 2 and 4 megabytes.

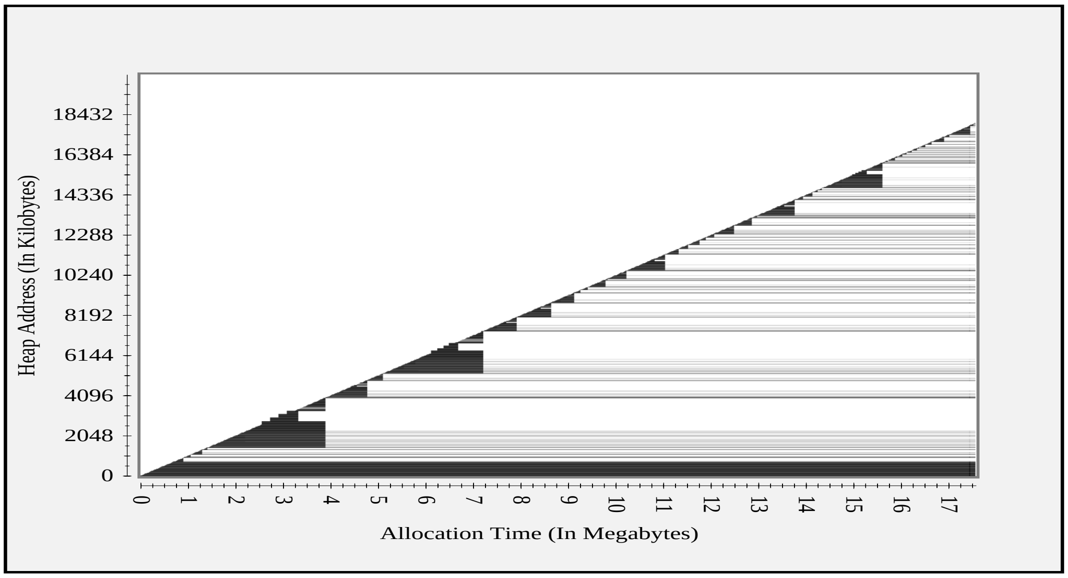

Figure 2.2: Fragmentation plot for GCC using the linear allocator

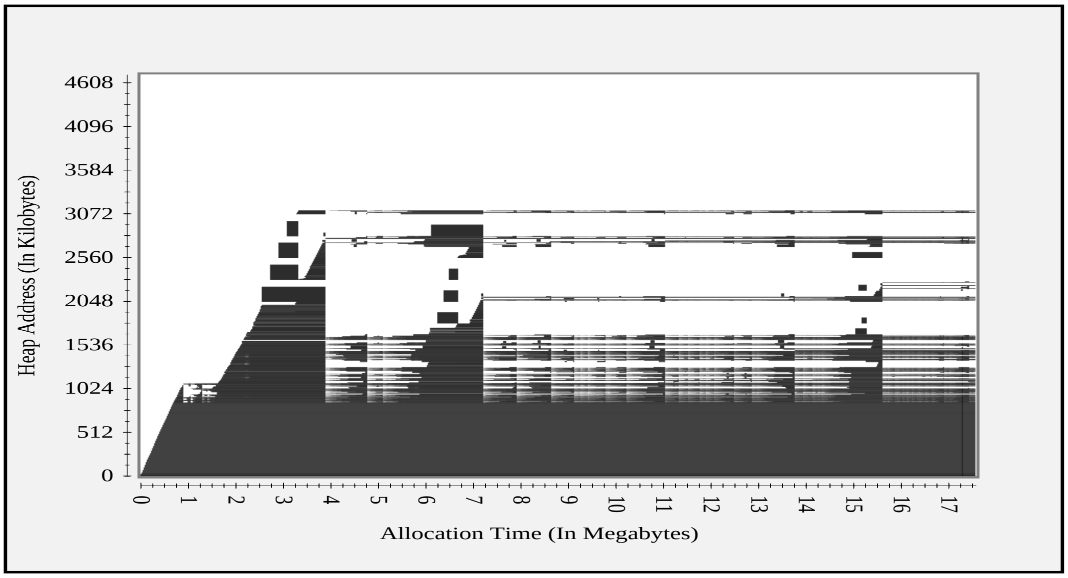

Figure 2.3: Fragmentation plot for GCC using the binary-buddy policy (accounting for all overheads)

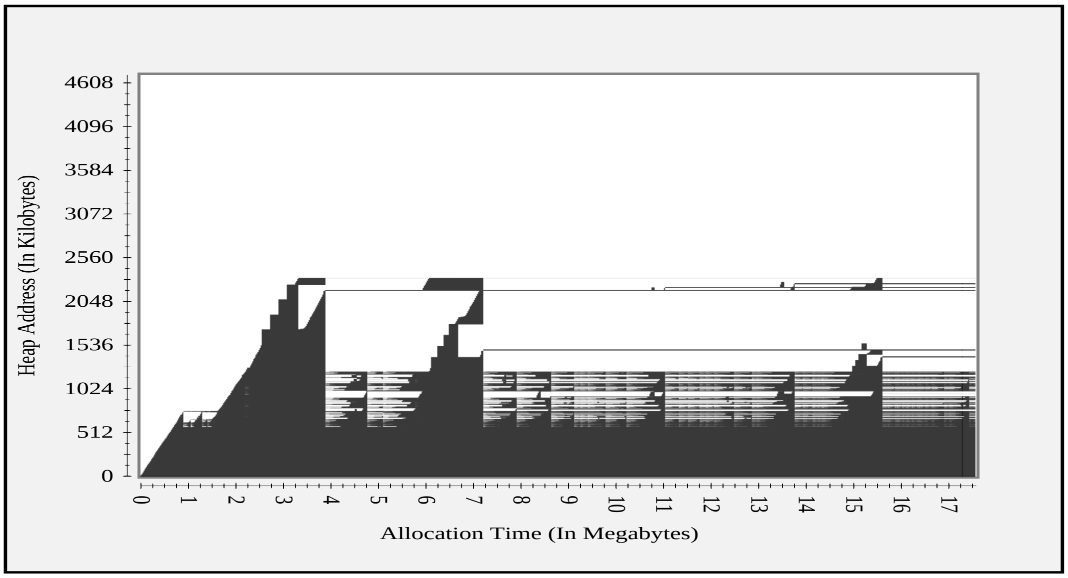

Figure 2.4: Fragmentation plot for GCC using the best-fit LIFO no footer policy (accounting for all overheads)

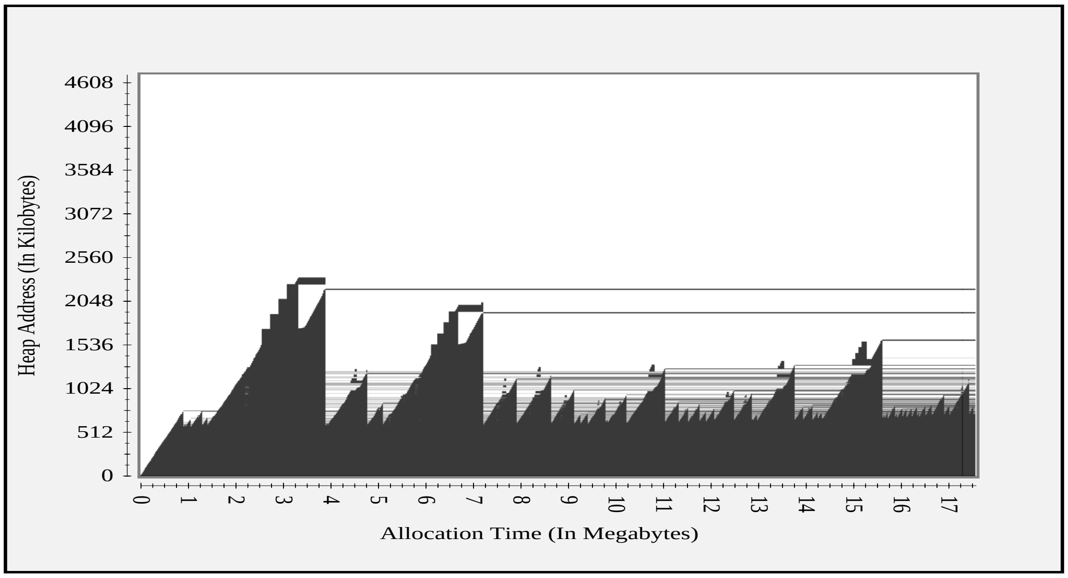

Figure 2.5: Fragmentation plot for GCC using the first-fit address-ordered no footer policy (accounting for all overheads)

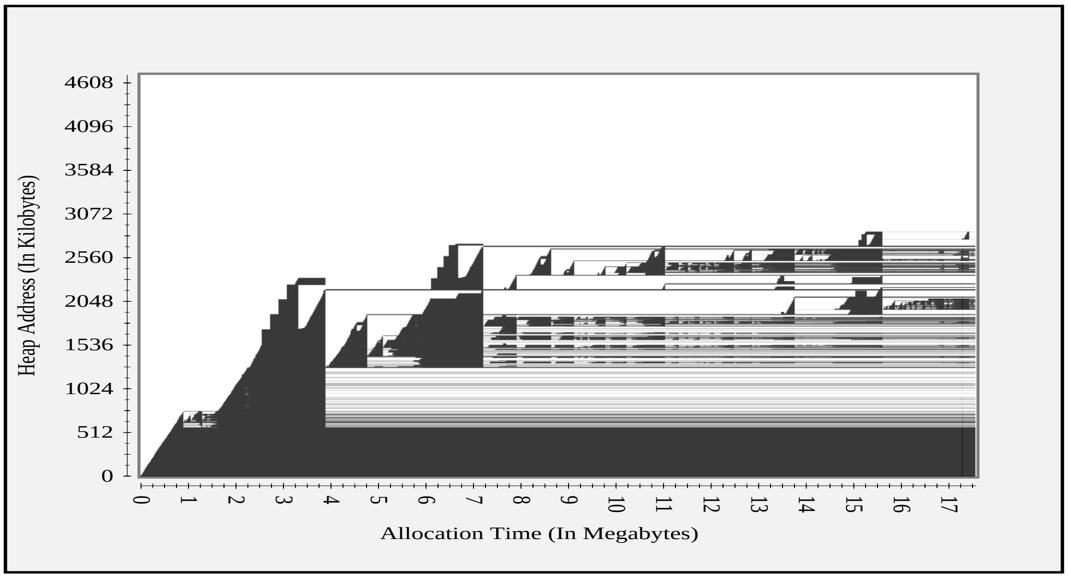

Figure 2.6: Fragmentation plot for GCC using the first-fit LIFO no footer policy (accounting for all overheads)

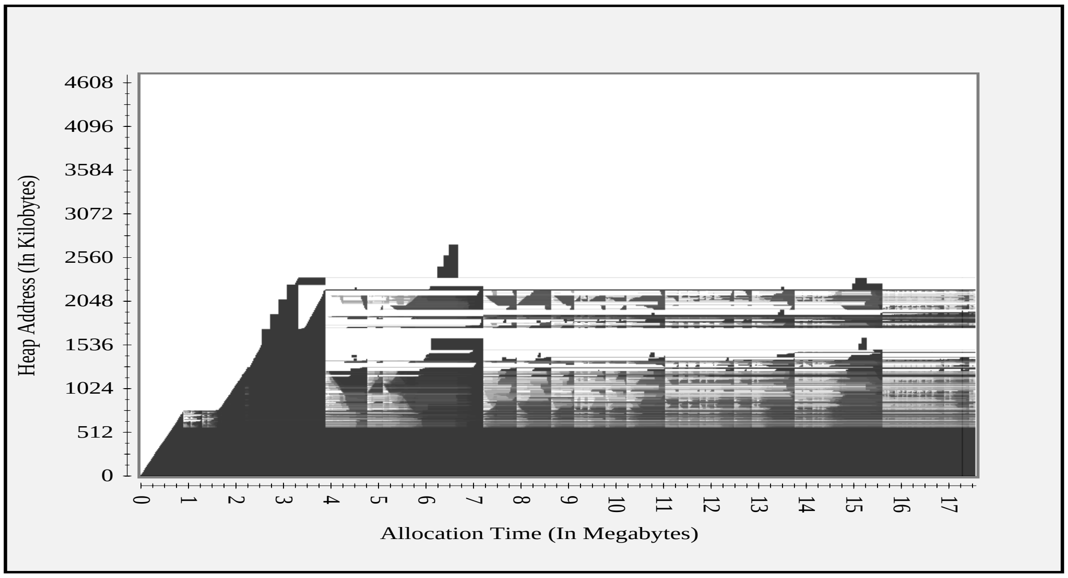

Figure 2.7: Fragmentation plot for GCC using the half-fit policy (accounting for all overheads)

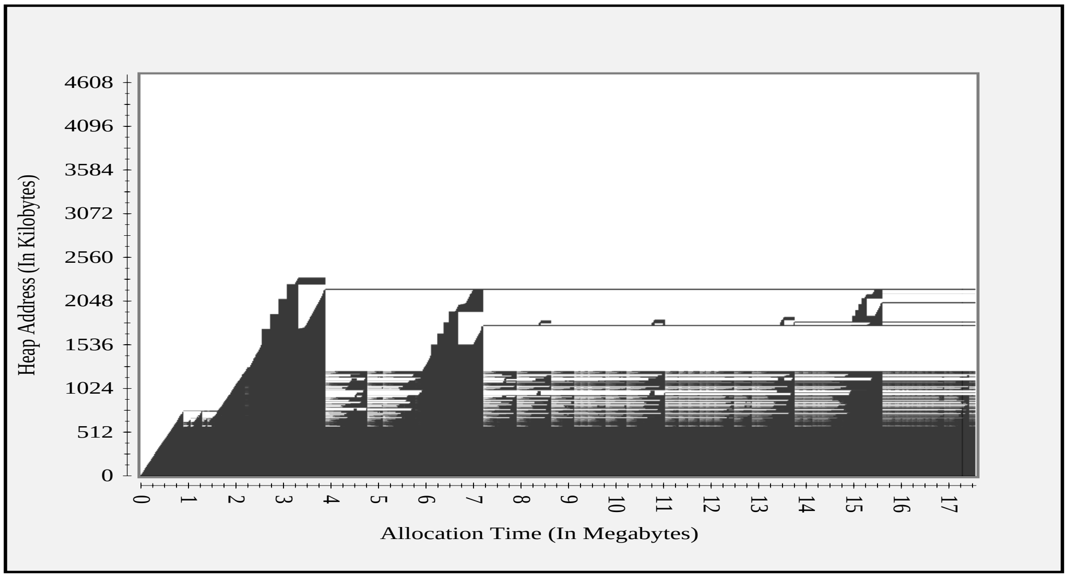

Figure 2.8: Fragmentation plot for GCC using Lea's 2.6.1 policy (accounting for all overheads)

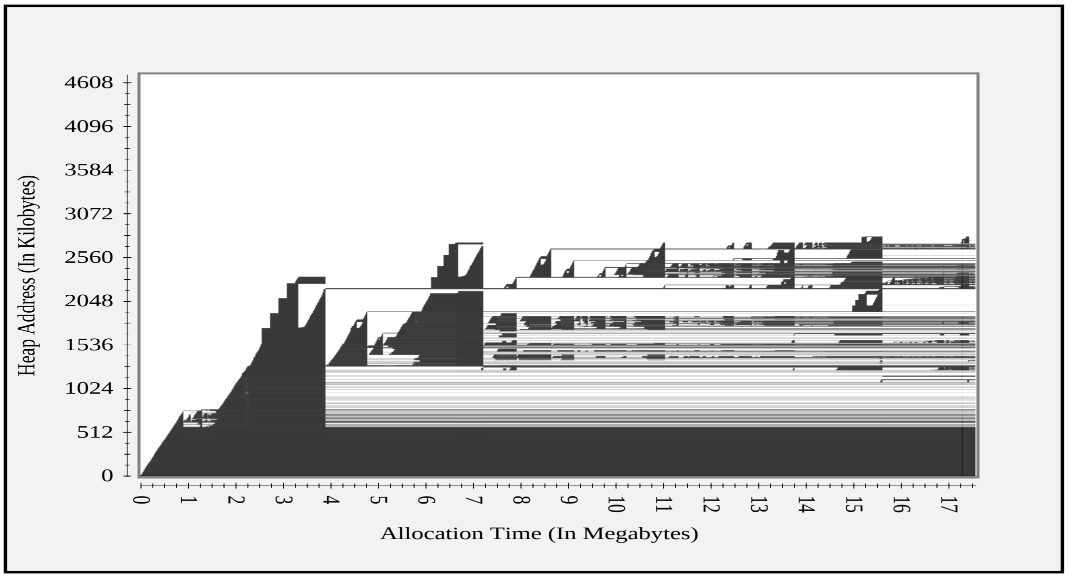

Figure 2.9: Fragmentation plot for GCC using the next-fit LIFO no footer policy (accounting for all overheads)

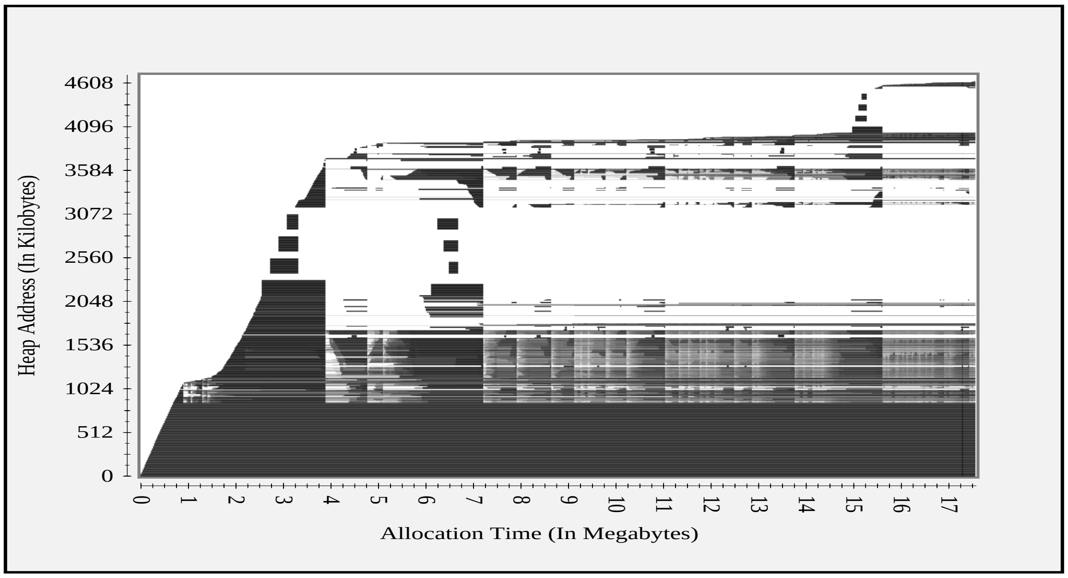

Figure 2.10: Fragmentation plot for GCC using the simple segregated storage 2^N policy (accounting for all overheads)

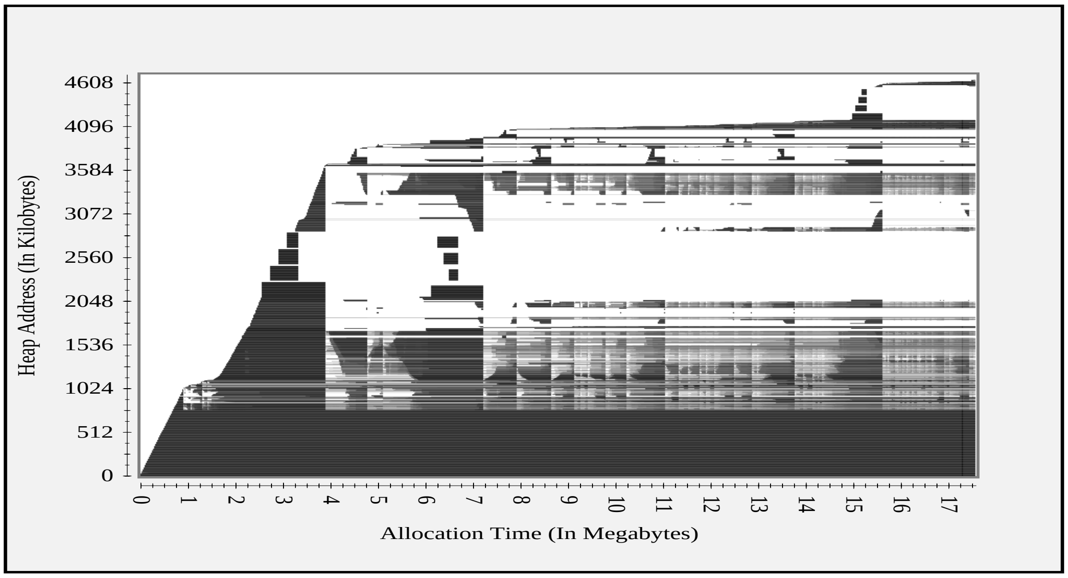

Figure 2.11: Fragmentation plot for GCC using the simple segregated storage 2^N & $3 * 2^N$ policy (accounting for all overheads)

2.14.2 Espresso Allocation Graphs

The memory usage of Espresso (Figures 2.12 to 2.21) is very different from that of the gnu C compiler. In particular, Espresso allocates three large objects that remain live for most of the run of the program. This can be clearly seen in Figure 2.13, starting at around 10 Megabytes. As can be seen from the linear allocator plot (Figure 2.12), the vast majority of the memory allocated for Espresso is in the form of small, short-lived objects. If we contrast the linear allocator plot to that of the other policies, we see that most of these short-lived objects are allocated in small clusters, showing up as spikes at around 16 megabytes of allocation, and again between 70 and 85 megabytes of allocation.

It is also particularly interesting to see how the lack of coalescing in the segregated storage system results in tremendous fragmentation (Figures 2.20 and 2.21). Even though there are very large areas of free memory available between heap addresses 128K and 200K, the objects allocated starting at around 10 Megabytes cannot be placed there, and cause the policy to request more memory from the operating system.

We believe that the allocation of the large object at about time 58 Megabytes is evidence of our strategy (Section 2.10) at work. Here, the three policies, best fit LIFO (Figure 2.14), first fit address-ordered (Figure 2.15), and Lea's 2.6.1 (Figure 2.18) all have sufficiently large free areas to service this request without requesting more memory from the operating system, while all of the other policies (Figures 2.13, 2.16, 2.17, 2.19, 2.20, and 2.21) must request more memory, resulting in higher fragmentation.

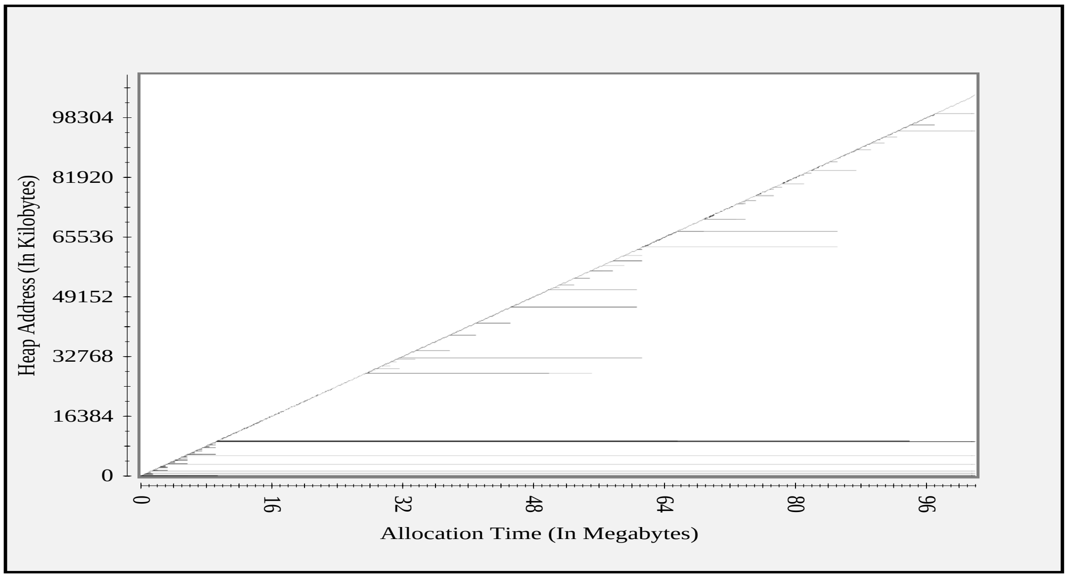

Figure 2.12: Fragmentation plot for Espresso using the linear allocator

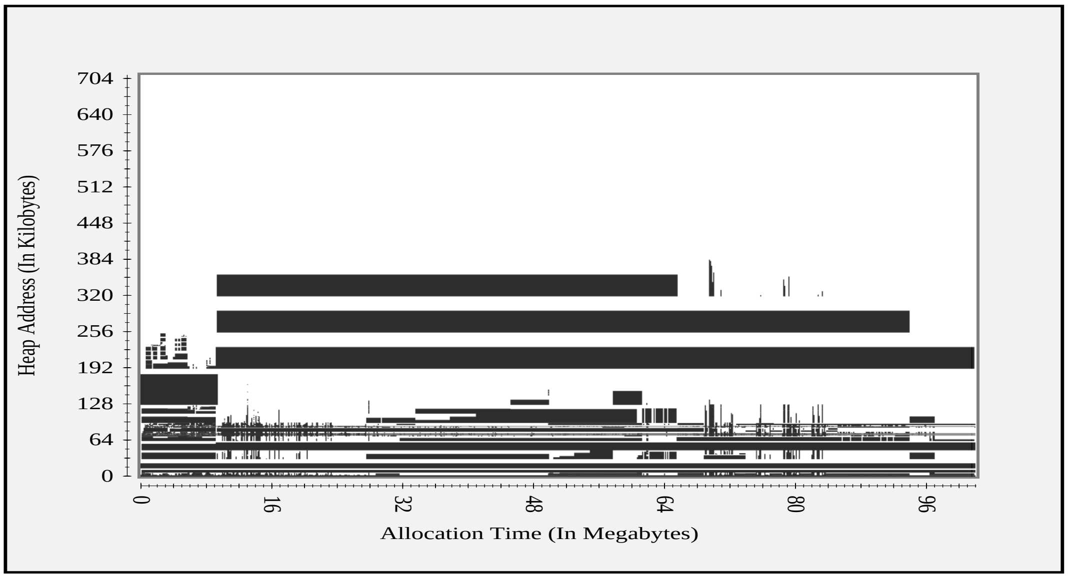

Figure 2.13: Fragmentation plot for Espresso using the binary-buddy policy (accounting for all overheads)

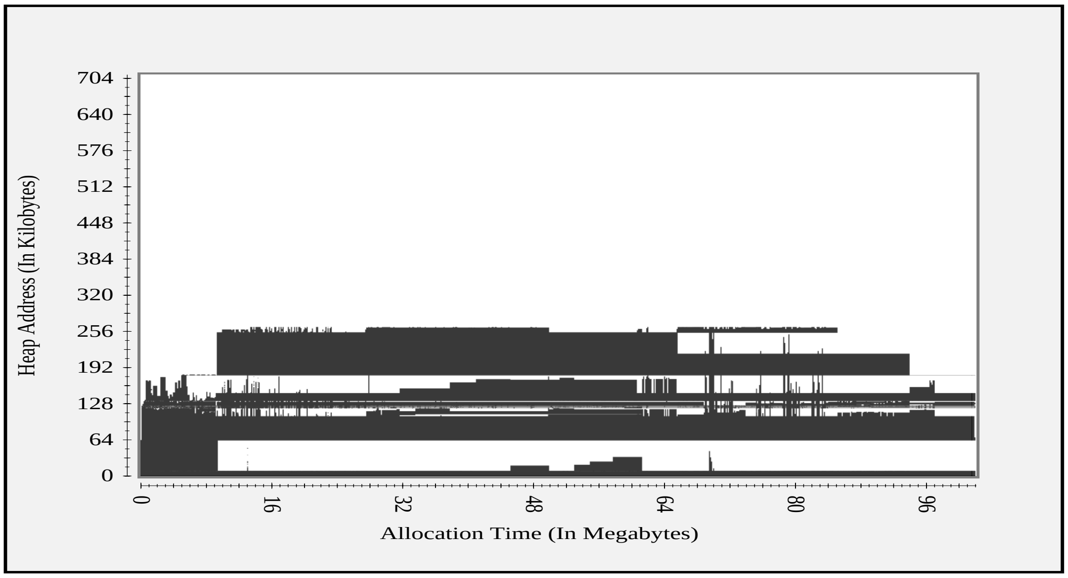

Figure 2.14: Fragmentation plot for Espresso using the best-fit LIFO no footer policy (accounting for all overheads)

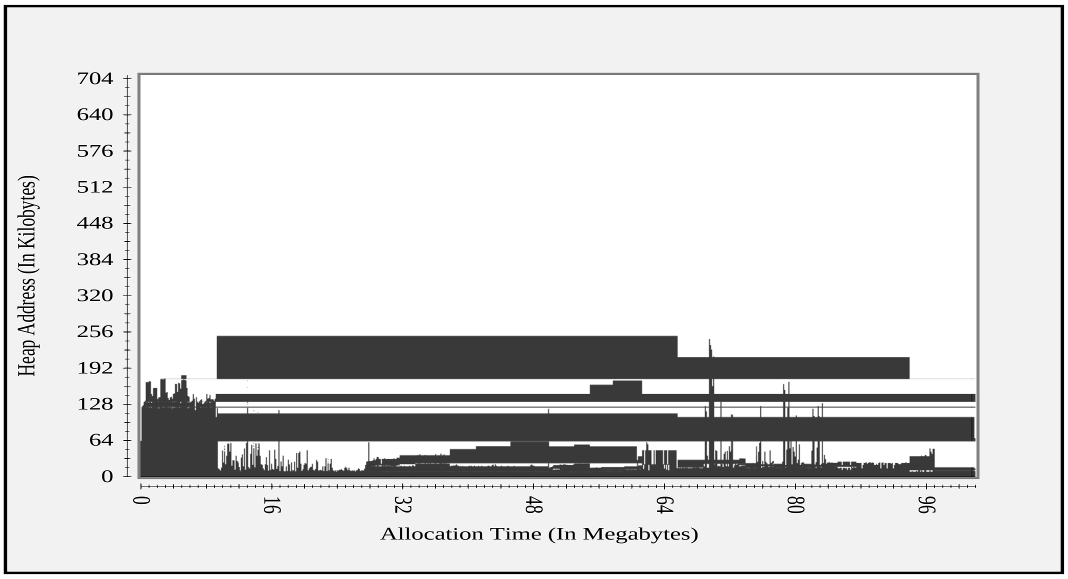

Figure 2.15: Fragmentation plot for Espresso using the first-fit address-ordered no footer policy (accounting for all overheads)

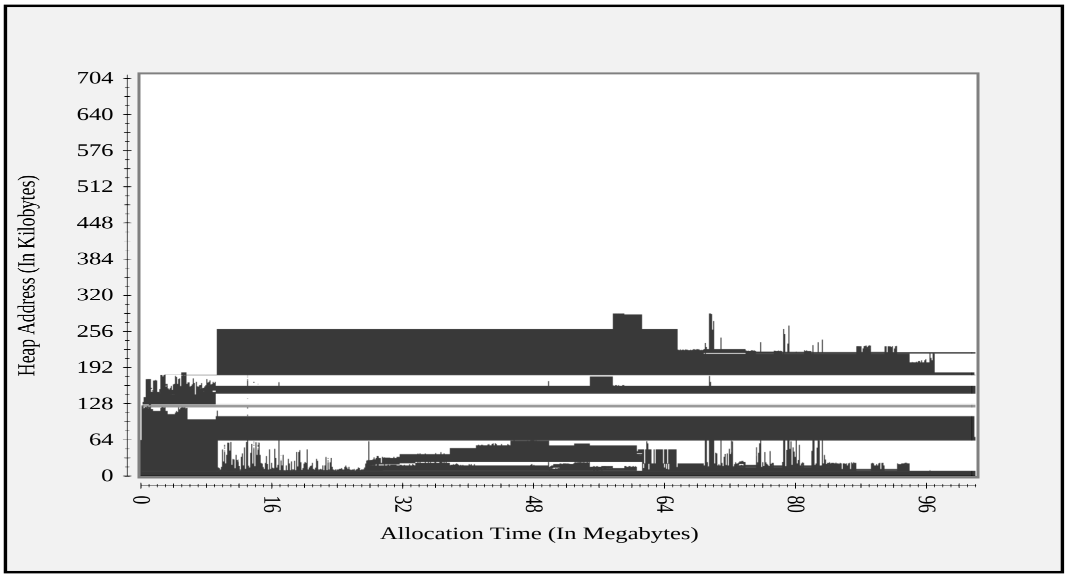

Figure 2.16: Fragmentation plot for Espresso using the first-fit LIFO no footer policy (accounting for all overheads)

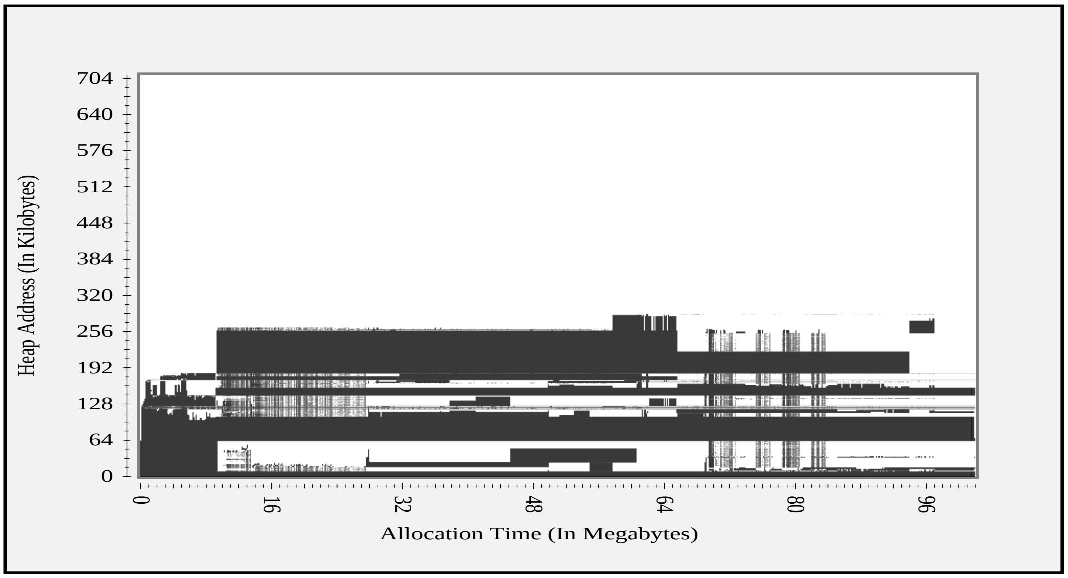

Figure 2.17: Fragmentation plot for Espresso using the half-fit policy (accounting for all overheads)

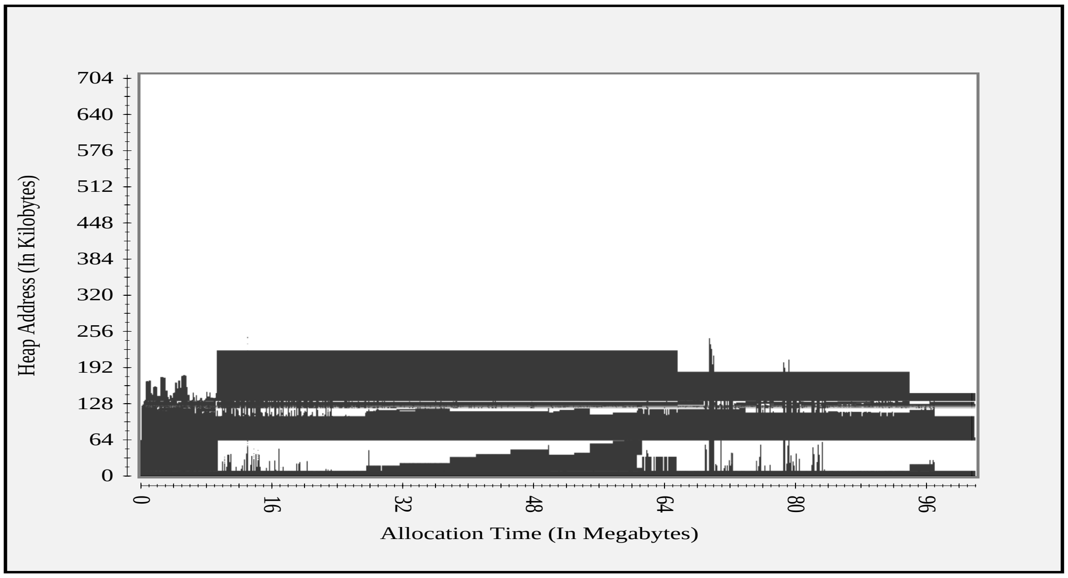

Figure 2.18: Fragmentation plot for Espresso using Lea's 2.6.1 policy (accounting for all overheads)

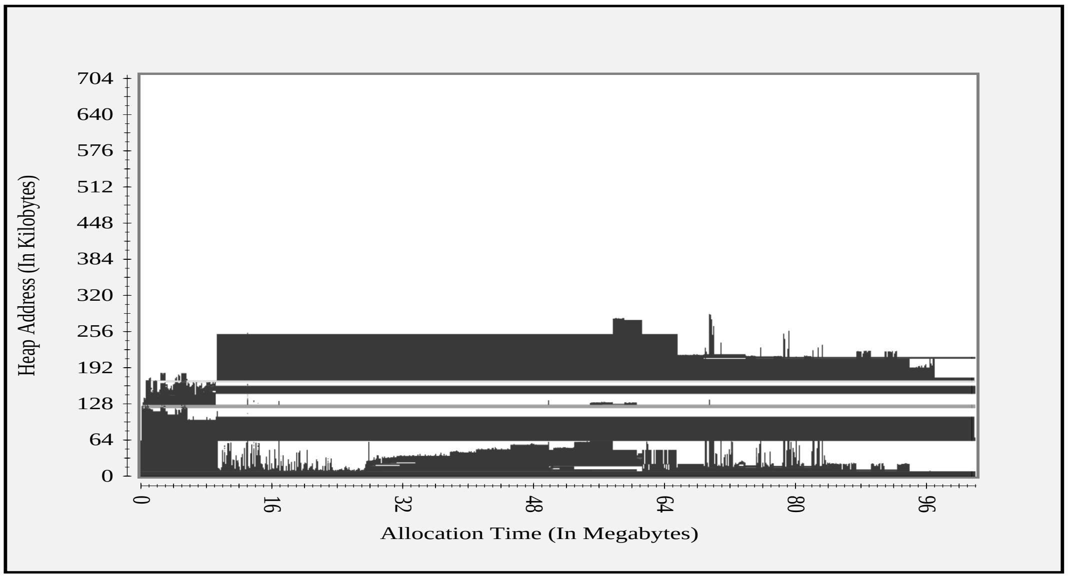

Figure 2.19: Fragmentation plot for Espresso using the next-fit LIFO no footer policy (accounting for all overheads)

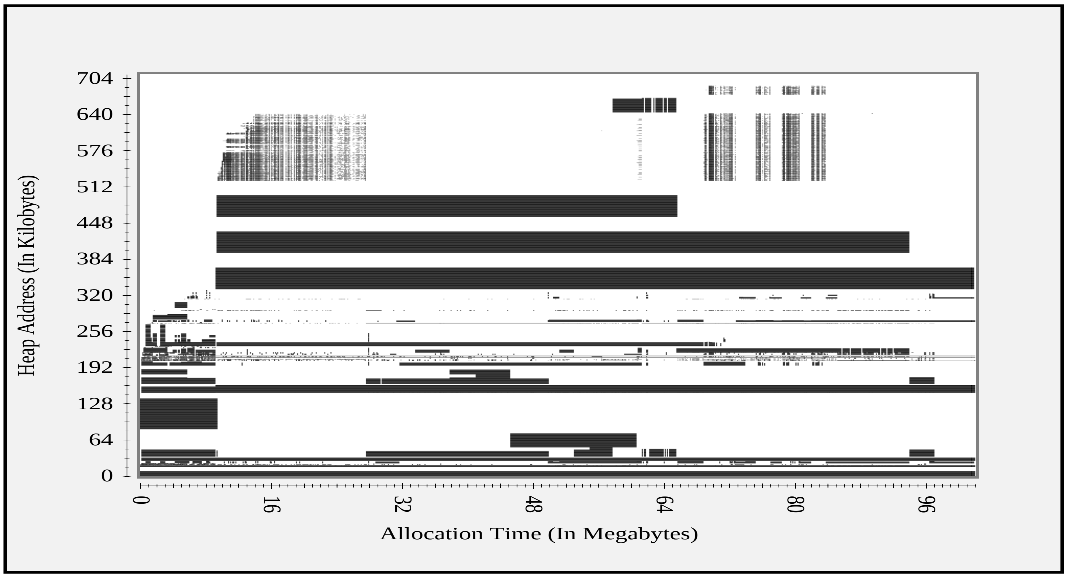

Figure 2.20: Fragmentation plot for Espresso using the simple segregated storage 2^N policy (accounting for all overheads)

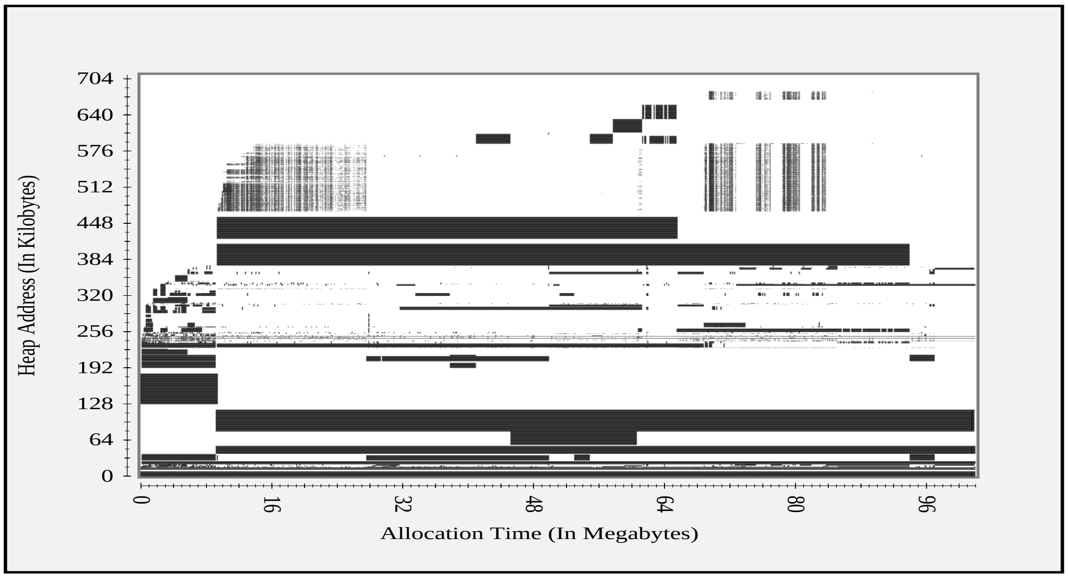

Figure 2.21: Fragmentation plot for Espresso using the simple segregated storage 2^N & $3*2^N$ policy (accounting for all overheads)

2.14.3 Ghostscript & Grobner Allocation Graphs

The Ghostscript program (figures 2.22 to 2.30) and the Grobner program (Figures 2.31 to 2.39) exhibit a third memory allocation pattern. These programs allocate memory slightly faster than they free it, causing a gradually increasing heap. The linear allocator plots (Figures 2.22 and 2.31), appear to show that these programs keep a large portion of their memory live for the entire run of the program. However, this effect is due to the scale at which the plot is presented (The Y-axis covers 50 megabytes of the heap). In Tables 2.7 and 2.8 (Section 2.11) we showed that on average, for 90% of the objects allocated by these two programs, only 2K of allocation time need pass after the deallocation of each object before both of its temporal neighbors are also deallocated. Thus, for these two programs, it is necessary that a policy be able to reuse small free areas for subsequent allocation requests.

For this pattern of memory allocation, first fit LIFO (Figures 2.26 and 2.35) and next fit LIFO (Figures 2.28 and 2.37) show considerably more fragmentation than the other policies. Interestingly, for the Ghostscript program, the segregated-storage policies (Figures 2.29 and 2.30) both generate relatively little fragmentation, while for the Grobner program, these two policies have considerable fragmentation (Figures 2.38 and 2.39). This is most likely due to the Ghostscript program allocating objects in fewer size classes than the Grobner program.

Our strategy is less obvious in these plots. However, we believe it is what allows good allocation policies, such as best fit LIFO (Figures 2.24 and 2.33) and first fit address ordered (Figures 2.25 and 2.34) to pack memory more tightly than first fit LIFO (Figures 2.26 and 2.35) and next fit LIFO (Figures 2.28 and 2.37) which do not follow this strategy.

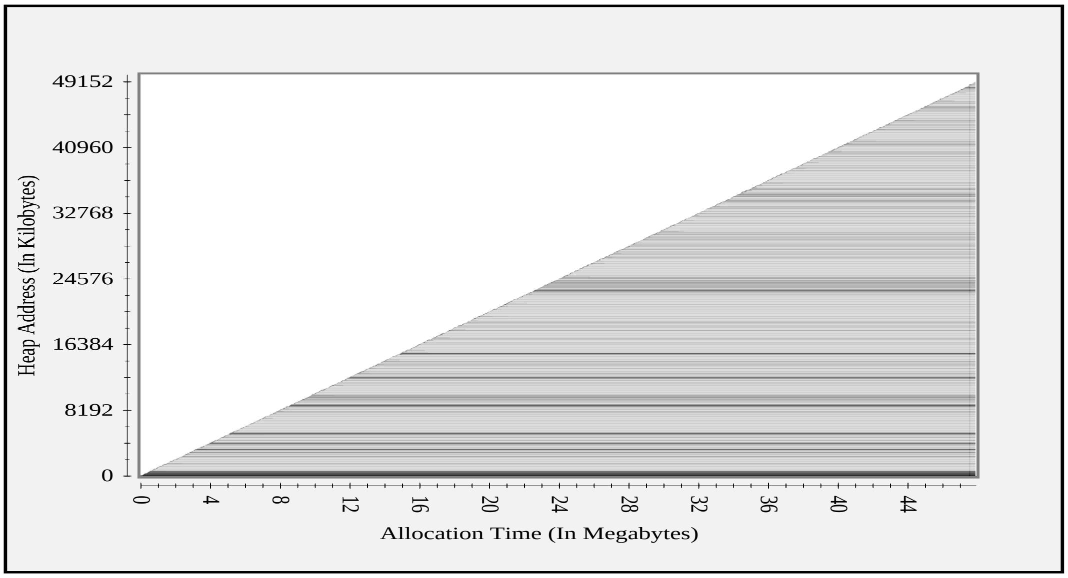

Figure 2.22: Fragmentation plot for Ghostscript using the linear allocator

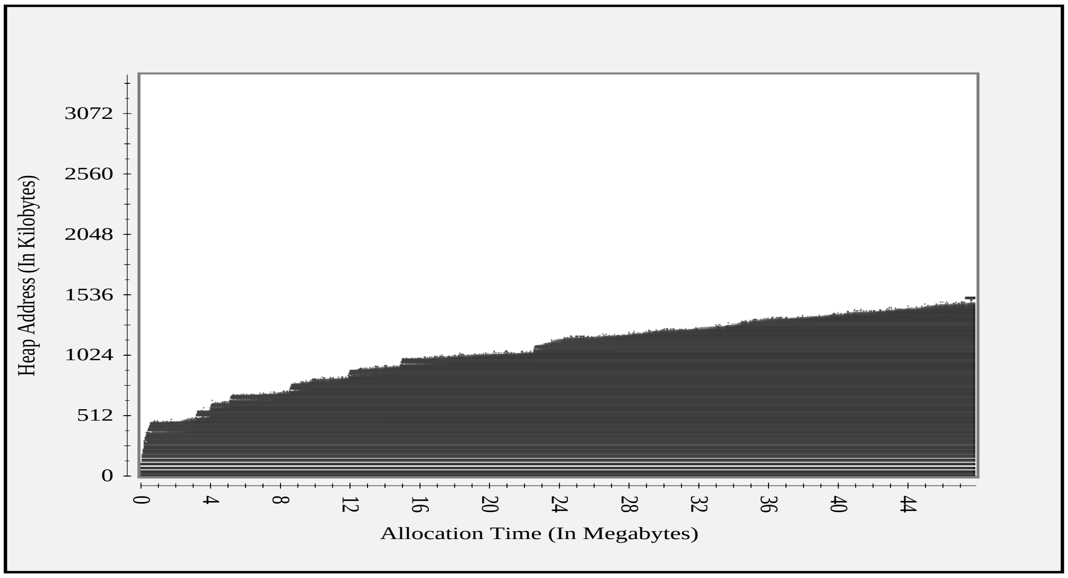

Figure 2.23: Fragmentation plot for Ghostscript using the binary-buddy policy (accounting for all overheads)

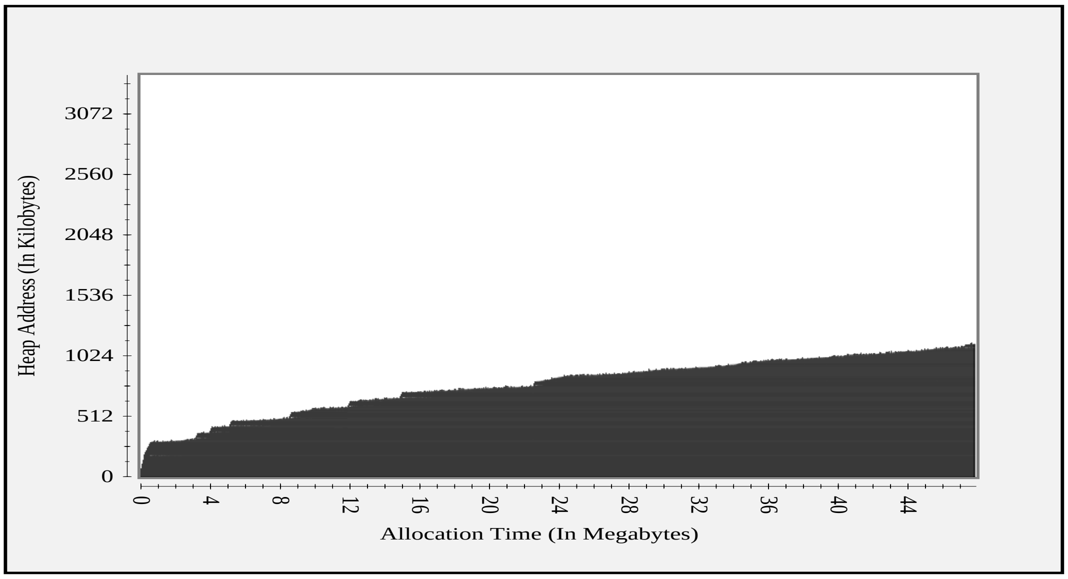

Figure 2.24: Fragmentation plot for Ghostscript using the best-fit LIFO no footer policy (accounting for all overheads)

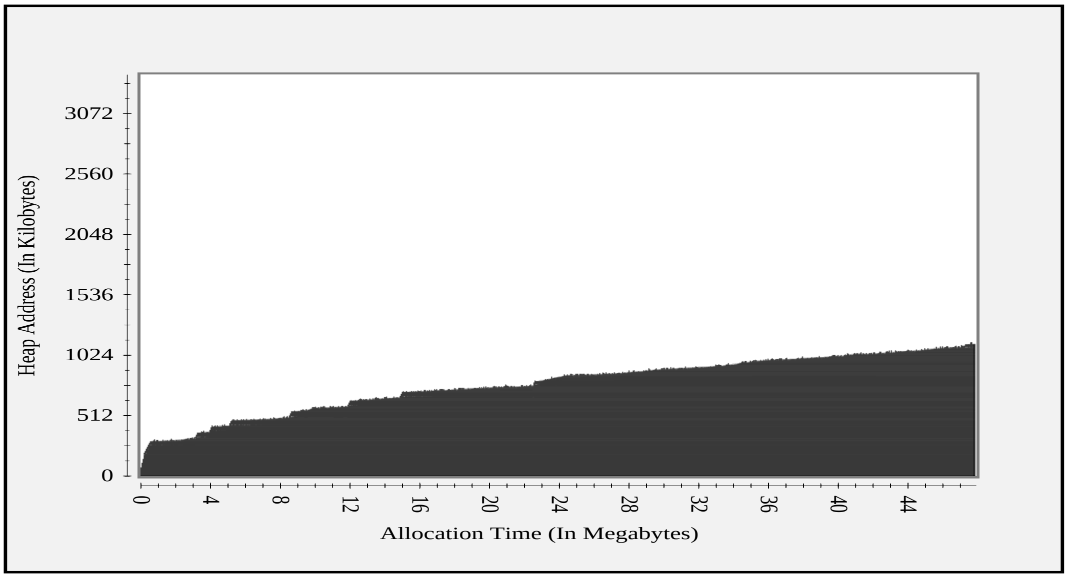

Figure 2.25: Fragmentation plot for Ghostscript using the first-fit address-ordered no footer policy (accounting for all overheads)

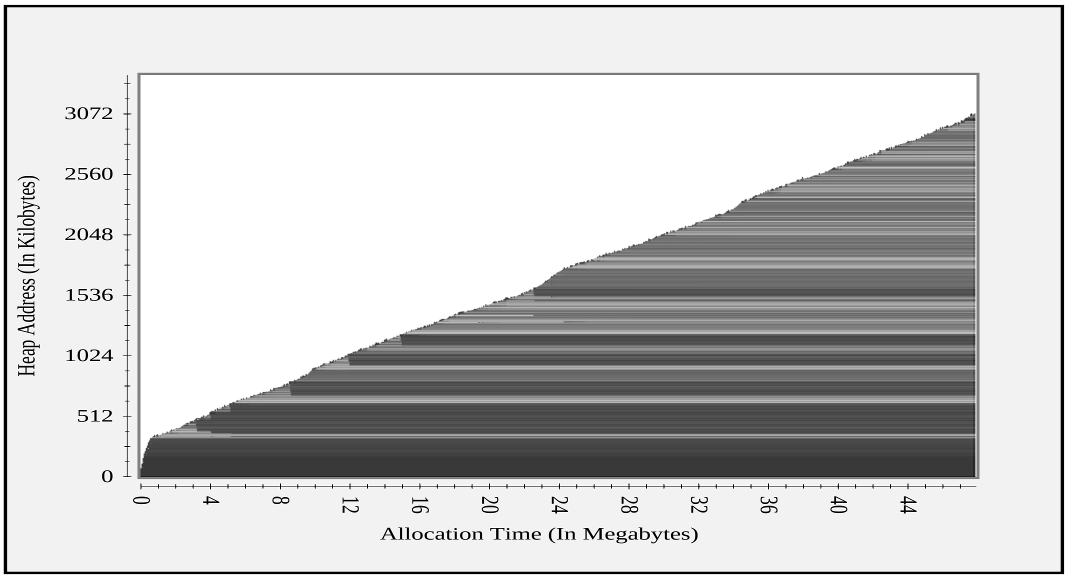

Figure 2.26: Fragmentation plot for Ghostscript using the first-fit LIFO no footer policy (accounting for all overheads)

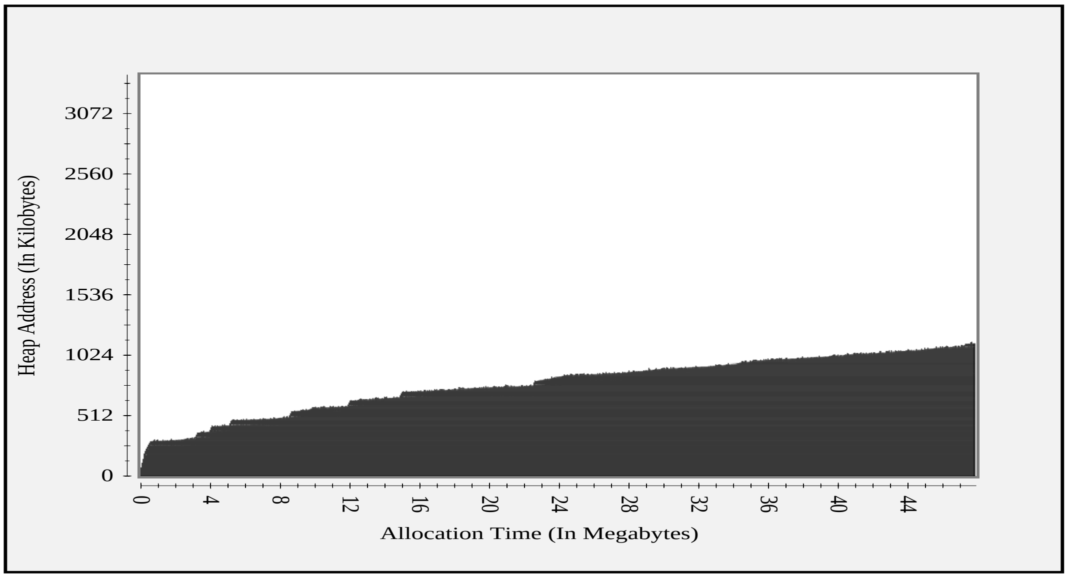

Figure 2.27: Fragmentation plot for Ghostscript using Lea's 2.6.1 policy (accounting for all overheads)

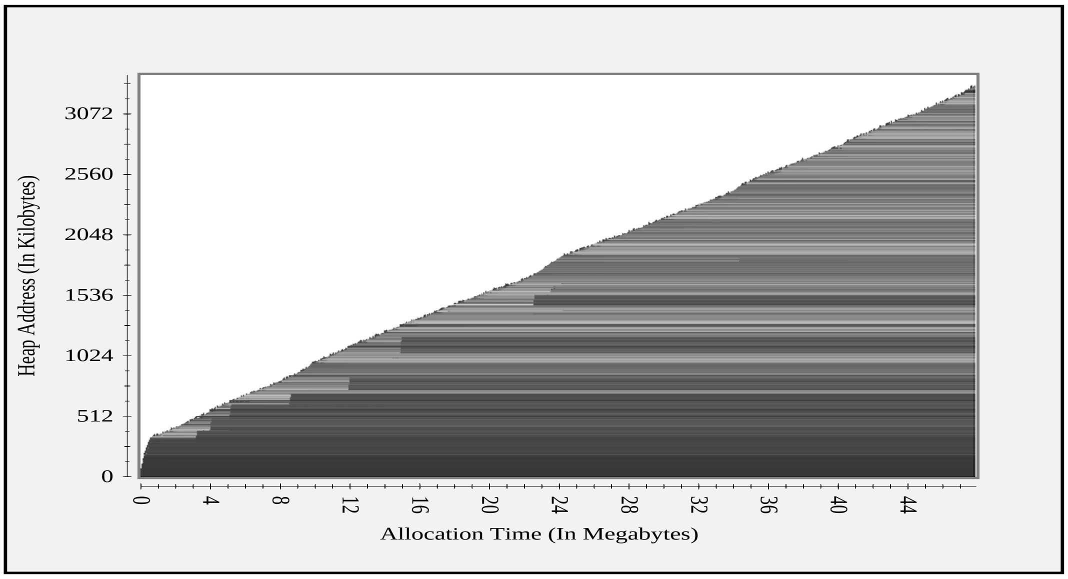

Figure 2.28: Fragmentation plot for Ghostscript using the next-fit LIFO no footer policy (accounting for all overheads)

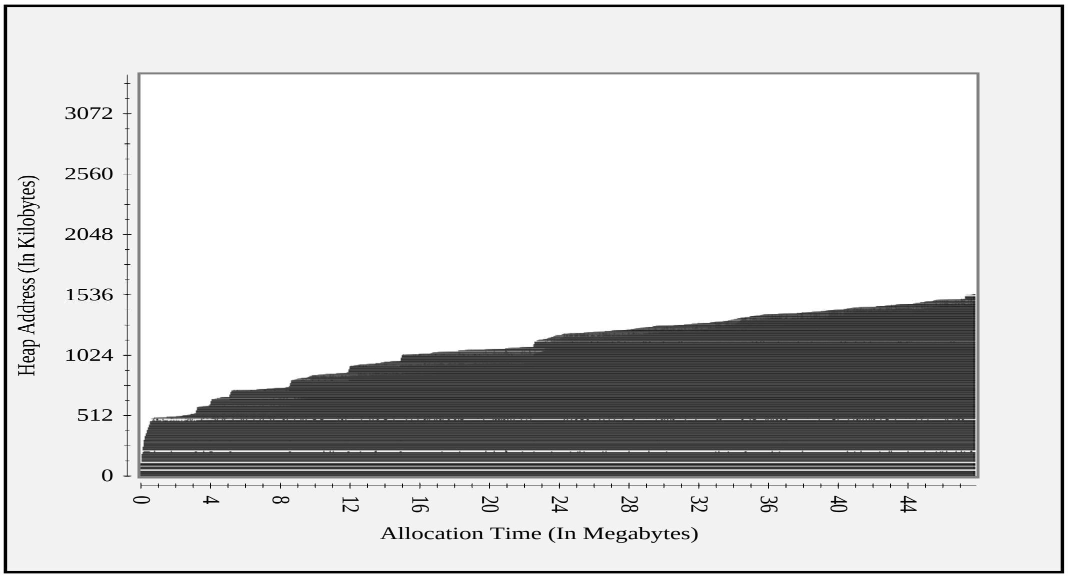

Figure 2.29: Fragmentation plot for Ghostscript using the simple segregated storage 2^N policy (accounting for all overheads)

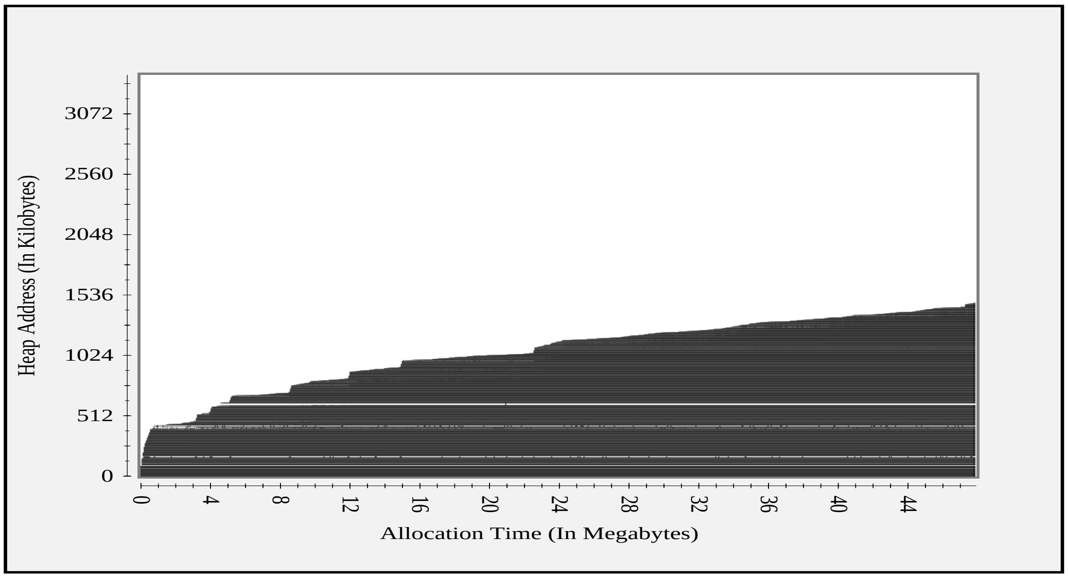

Figure 2.30: Fragmentation plot for Ghostscript using the simple segregated storage 2^N & $3 * 2^N$ policy (accounting for all overheads)

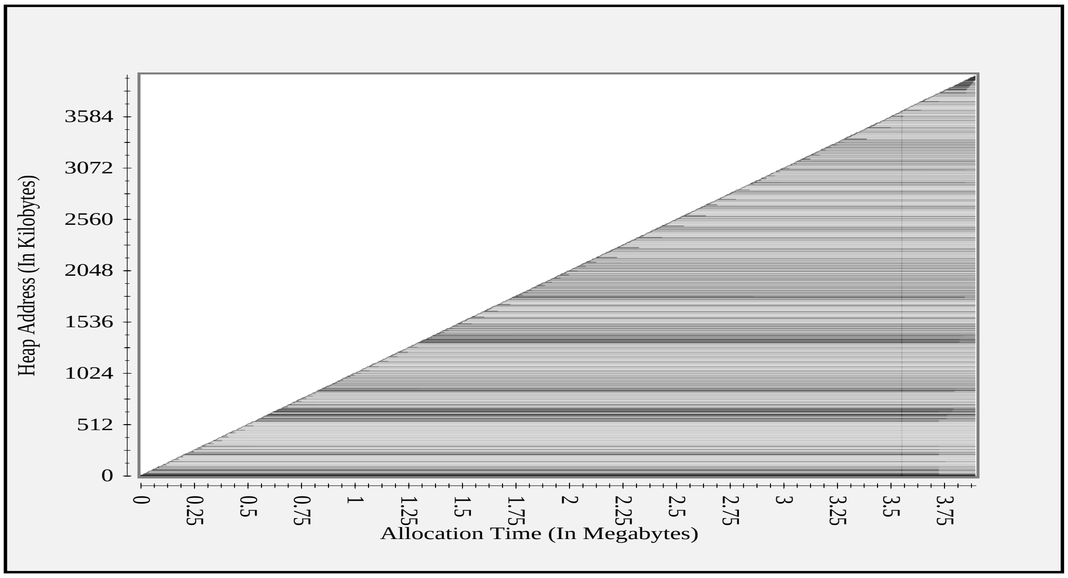

Figure 2.31: Fragmentation plot for Grobner using the linear allocator

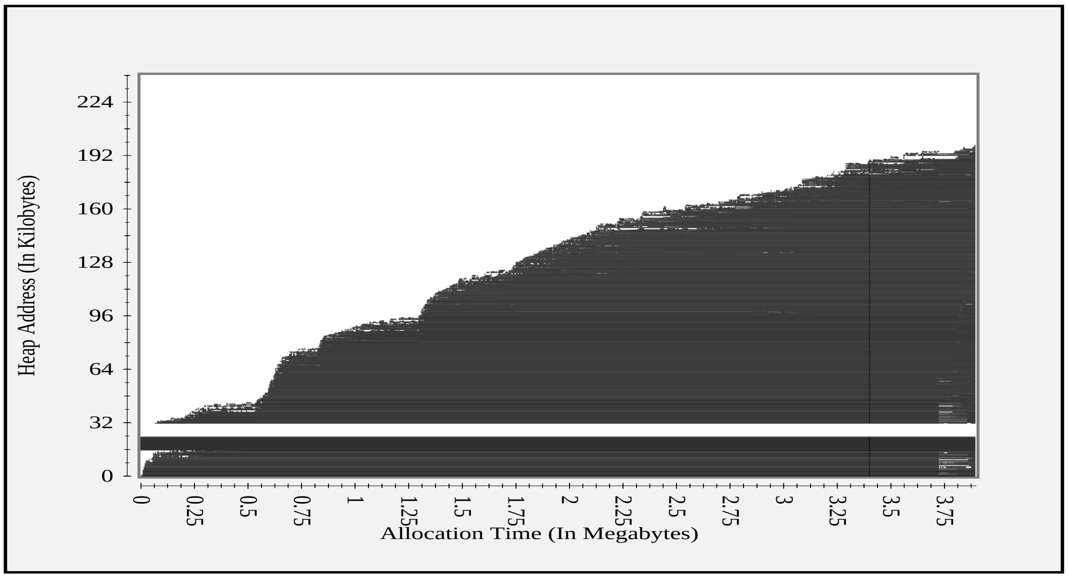

Figure 2.32: Fragmentation plot for Grobner using the binary-buddy policy (accounting for all overheads)

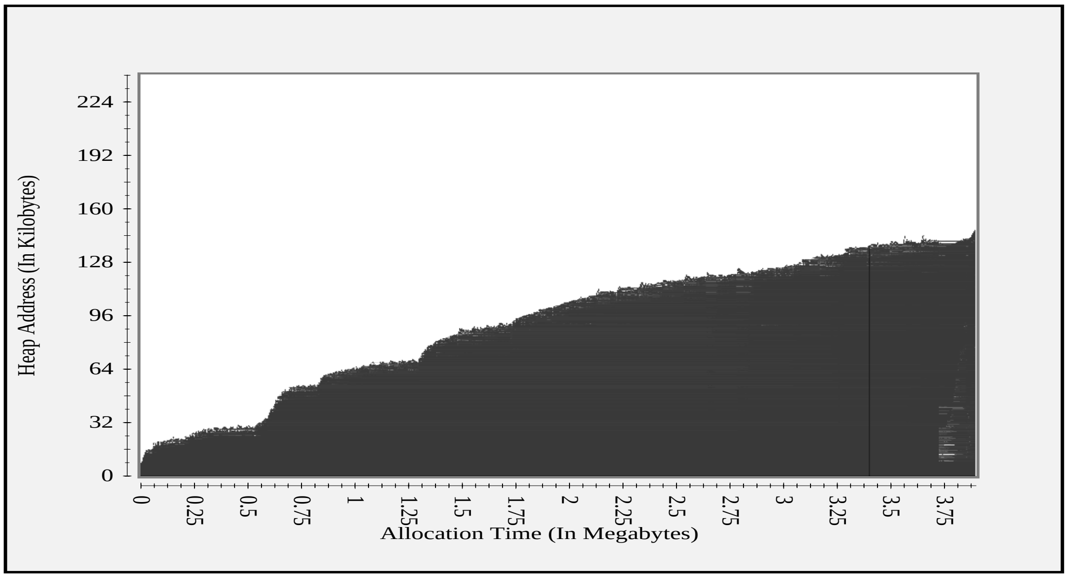

Figure 2.33: Fragmentation plot for Grobner using the best-fit LIFO no footer policy (accounting for all overheads)

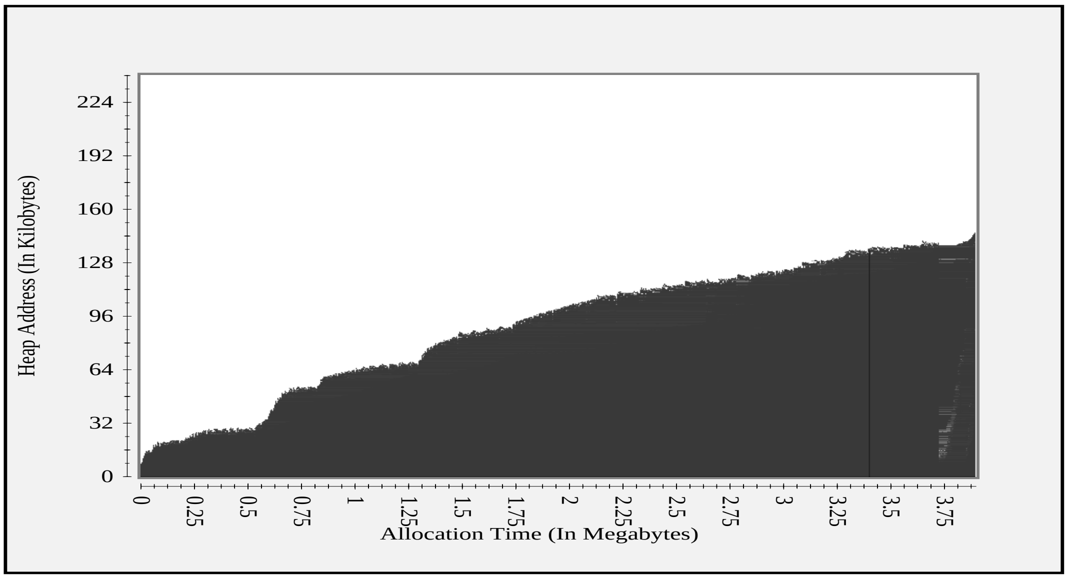

Figure 2.34: Fragmentation plot for Grobner using the first-fit address-ordered no footer policy (accounting for all overheads)

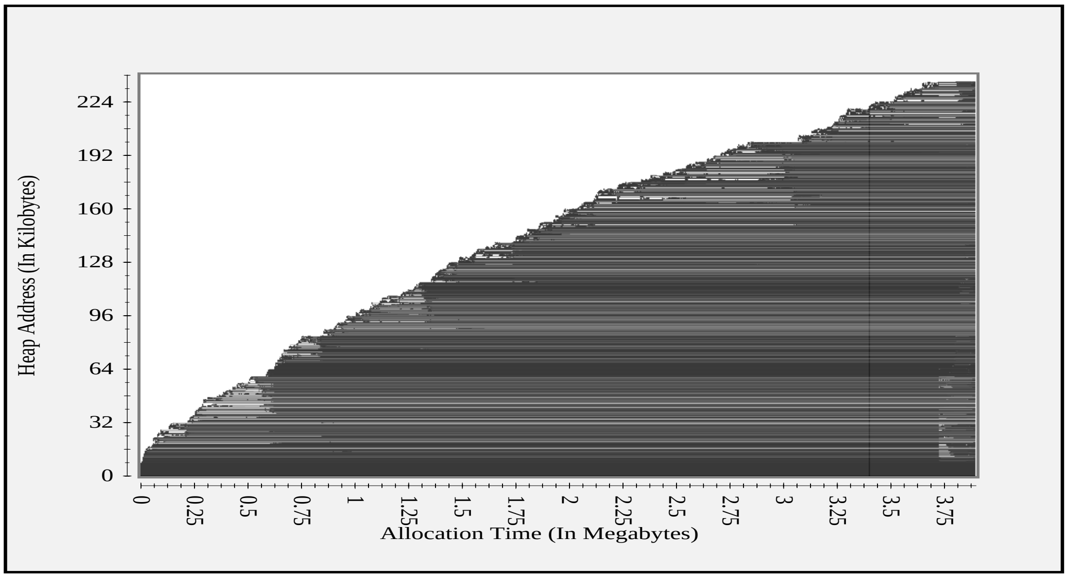

Figure 2.35: Fragmentation plot for Grobner using the first-fit LIFO no footer policy (accounting for all overheads)

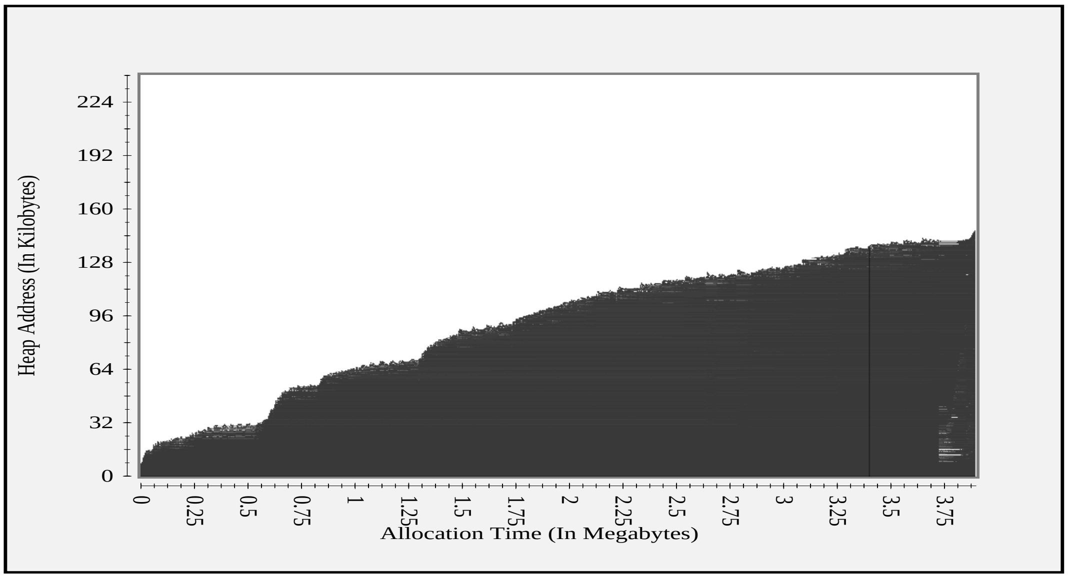

Figure 2.36: Fragmentation plot for Grobner using Lea's 2.6.1 policy (accounting for all overheads)

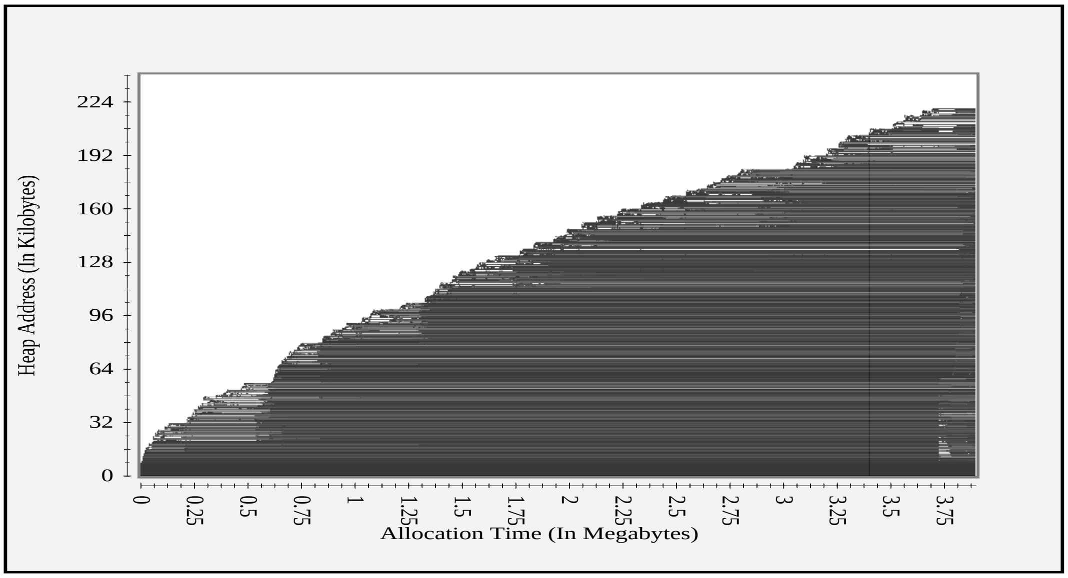

Figure 2.37: Fragmentation plot for Grobner using the next-fit LIFO no footer policy (accounting for all overheads)

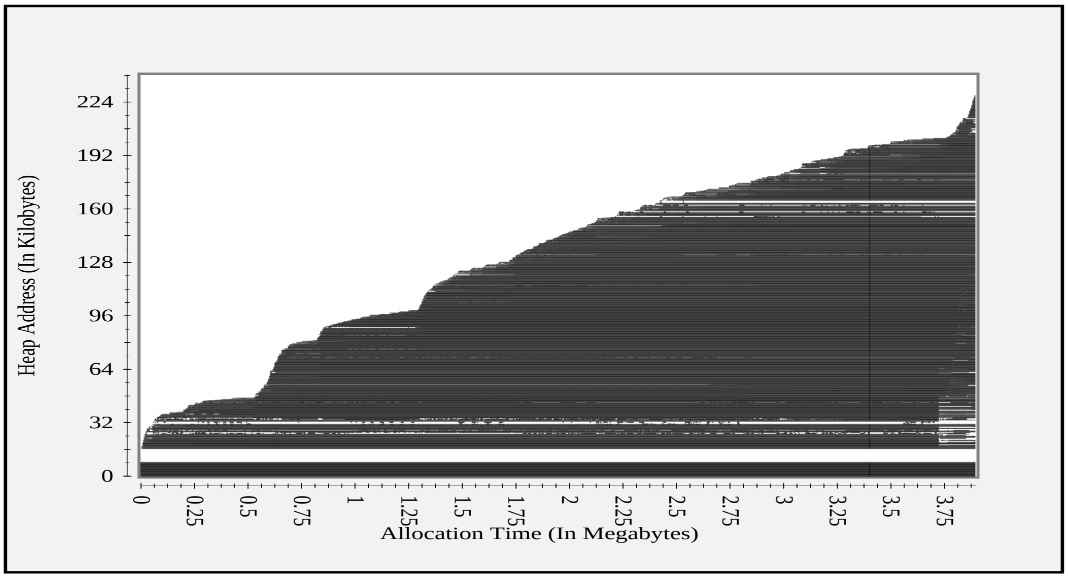

Figure 2.38: Fragmentation plot for Grobner using the simple segregated storage 2^N policy (accounting for all overheads)

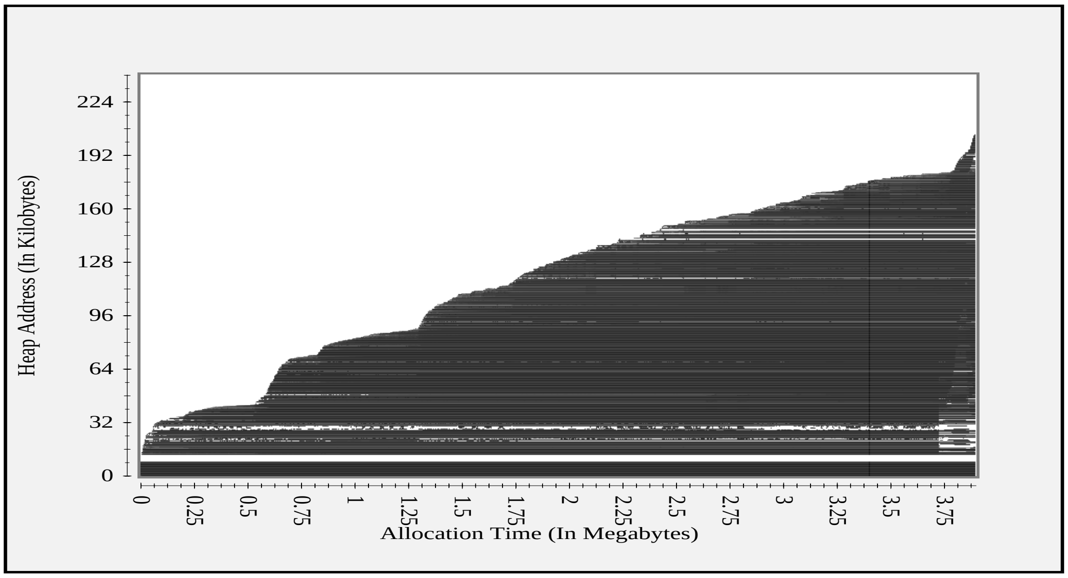

Figure 2.39: Fragmentation plot for Grobner using the simple segregated storage 2^N & $3 * 2^N$ policy (accounting for all overheads)

2.14.4 Hyper Allocation Graphs

The memory allocation behavior of the Hyper program (Figures 2.40 to 2.42) is very simple. Two large blocks are allocated and stay live for the entire lifetime of the program. The remainder of the memory allocation requests consist of many small objects being allocated and quickly deallocated (this can be seen as the diagonal line in the linear allocator plot (Figure 2.40)). Most of our policies performed well on this program. In Figure 2.42 we can see that the best-fit LIFO policy has essentially no fragmentation. Binary buddy (Figure 2.41), on the other hand, performs particularly poorly compared to the other policies. This is because the buddy-system policy requires that a block of a particular size be paired with a block of the same size. Since the smaller of the two large objects was allocated first, external fragmentation was required to bring the first region of memory up to the size of the second requested block.

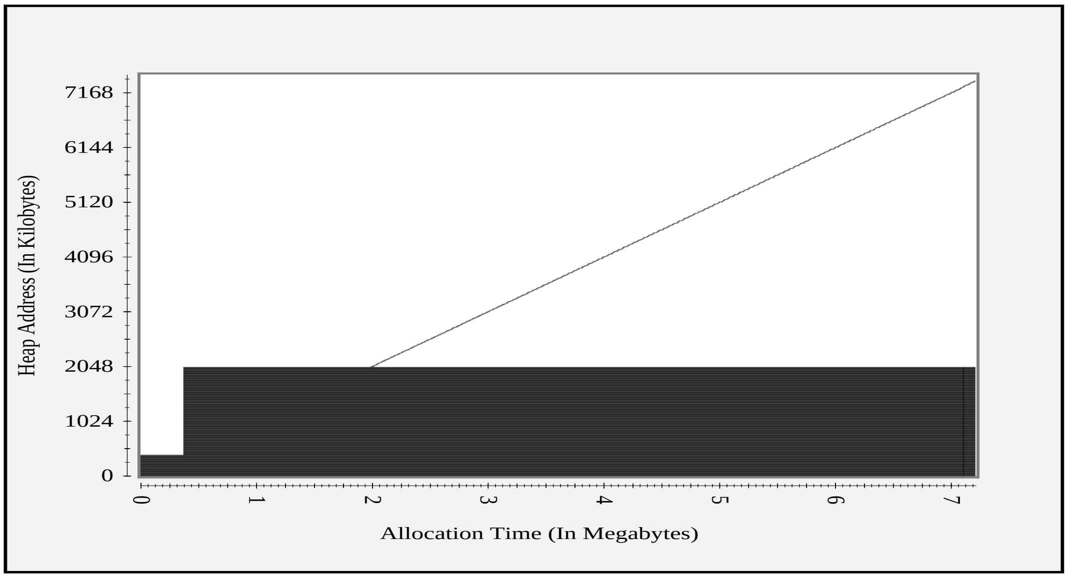

Figure 2.40: Fragmentation plot for Hyper using the linear allocator

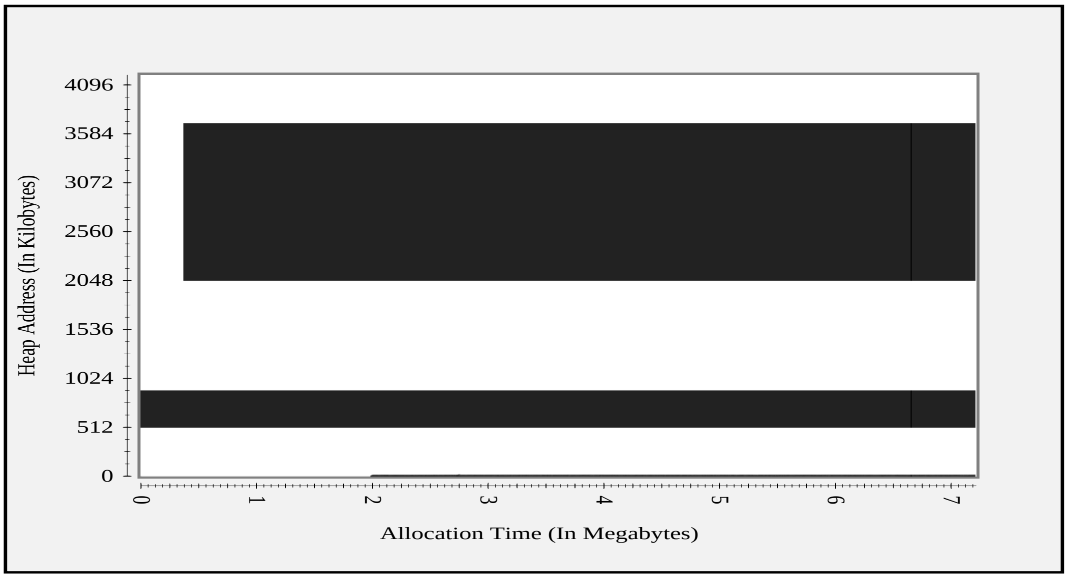

Figure 2.41: Fragmentation plot for Hyper using the binary-buddy policy (accounting for all overheads)

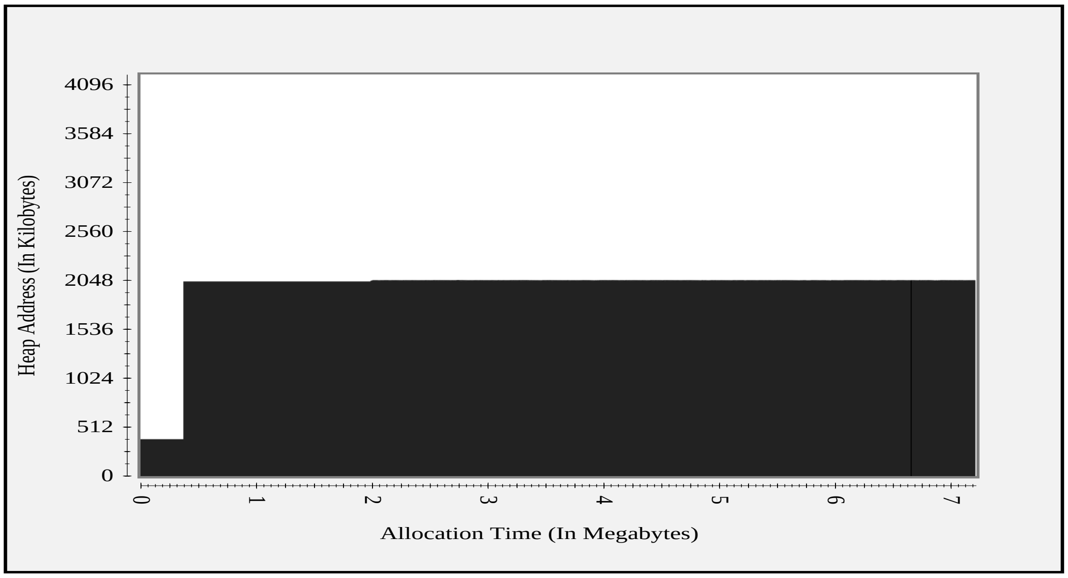

Figure 2.42: Fragmentation plot for Hyper using the best-fit LIFO no footer policy (accounting for all overheads)

2.14.5 P2C Allocation Graphs

The P2C program (Figures 2.43 to 2.48) is another program with strong phase behavior. Of particular note for this program is the increased fragmentation exhibited by the binary-buddy policy (Figure 2.44) and the two segregated-storage policies (Figures 2.47 and 2.48) after the large data structure is freed at allocation time 3 Megabytes, compared to the fragmentation exhibited by the best-fit LIFO (Figure 2.45) and first-fit address-ordered (Figure 2.46) policies.

Note the considerable fragmentation, showing up as empty streaks across both plots (Figures 2.47 and 2.48), caused by the simple segregated storage policies. The P2C program allocates 90% of its objects in just 4 sizes (Table 2.9), and just 14K of allocation time need pass after the deallocation of 90% of all objects before both of their temporal neighbors are deallocated (Table 2.7), so it is somewhat surprising that there is any fragmentation at all for this program. These streaks are evidence that the simple segregated storage policies increase internal fragmentation in a futile attempt to reduce external fragmentation.

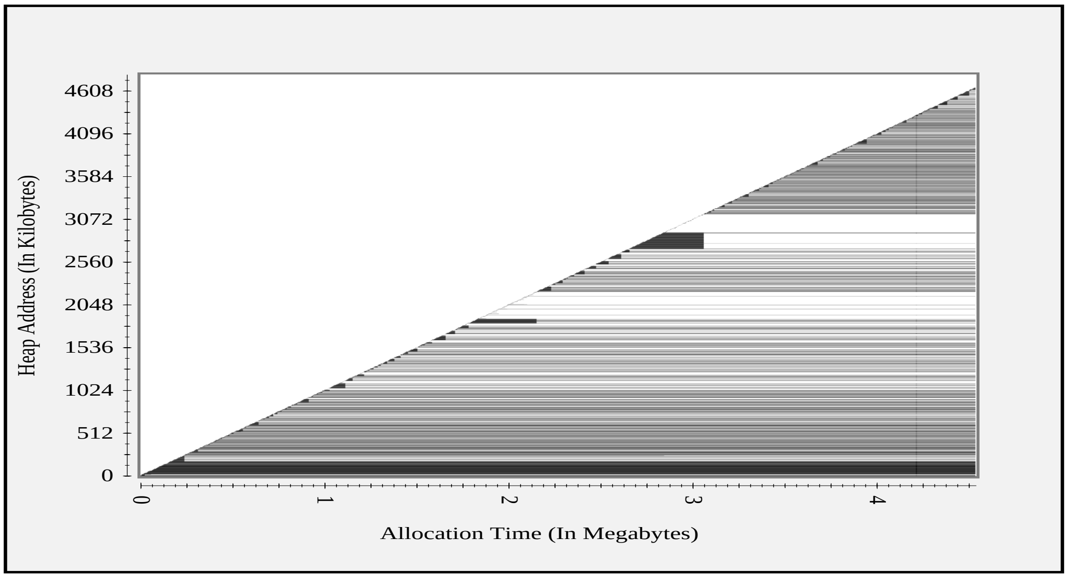

Figure 2.43: Fragmentation plot for P2C using the linear allocator

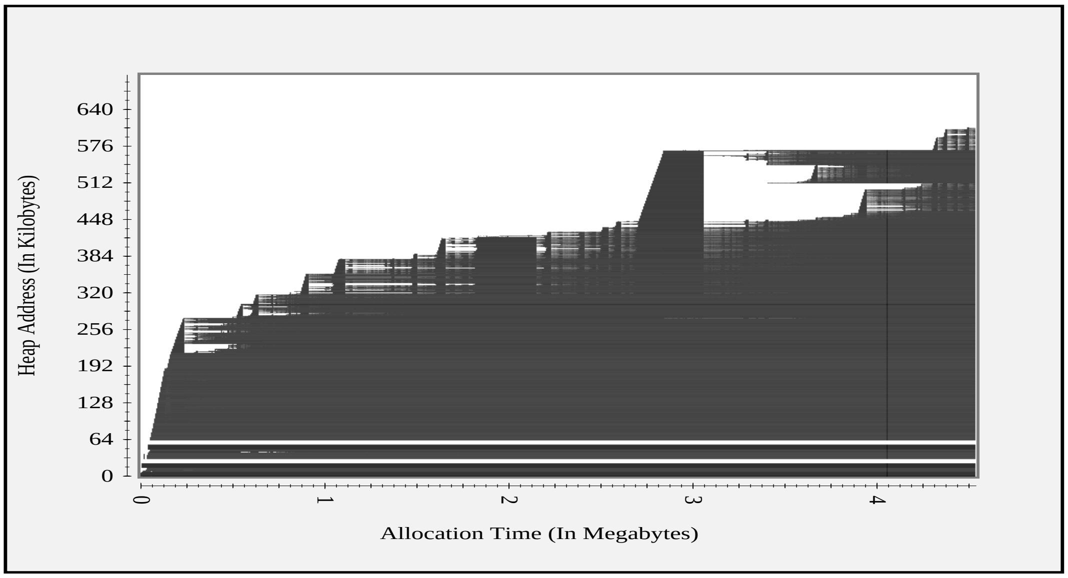

Figure 2.44: Fragmentation plot for P2C using the binary-buddy policy (accounting for all overheads)

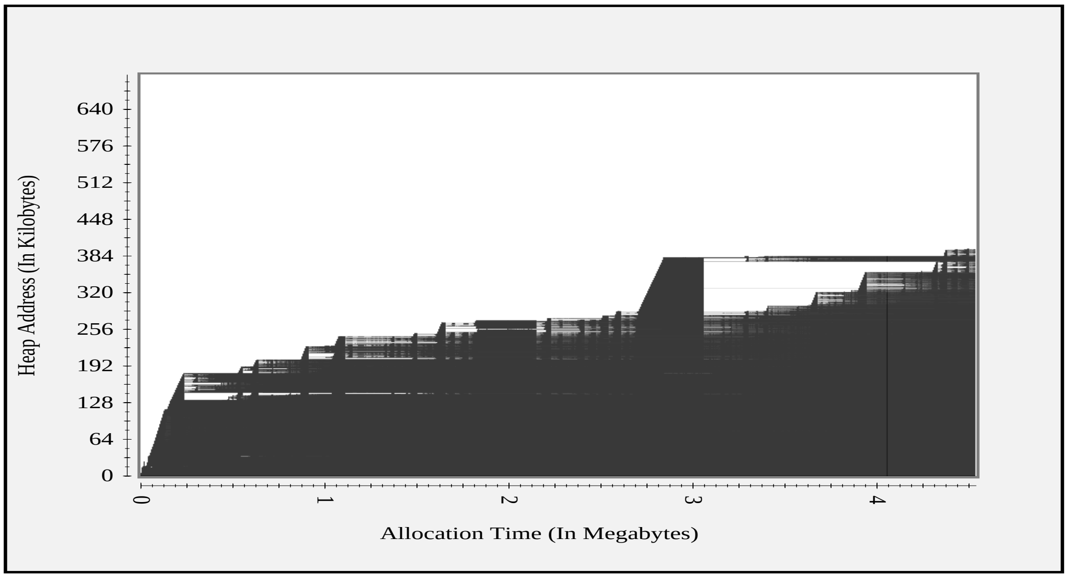

Figure 2.45: Fragmentation plot for P2C using the best-fit LIFO no footer policy (accounting for all overheads)

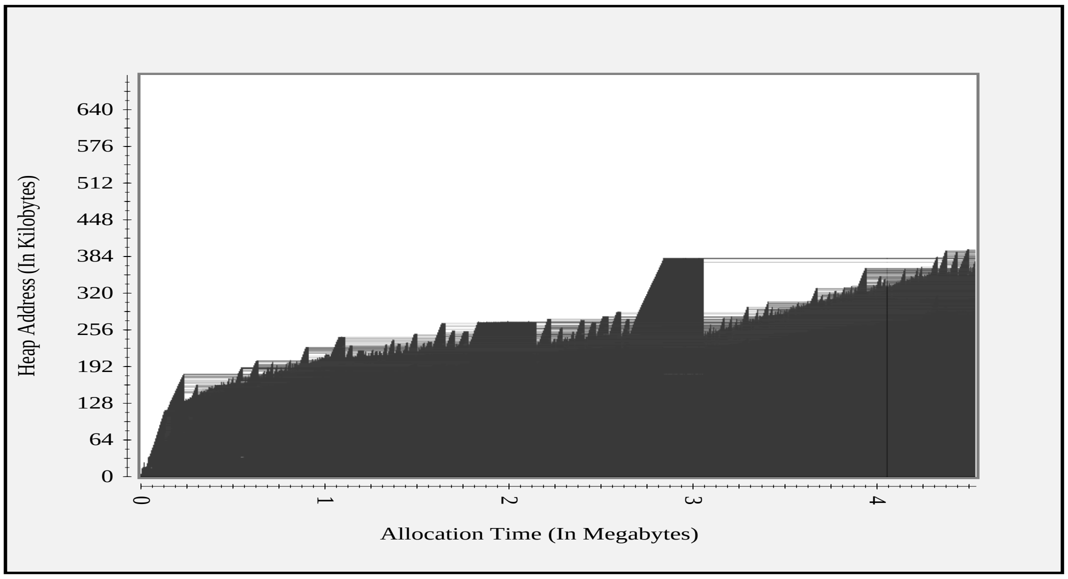

Figure 2.46: Fragmentation plot for P2C using the first-fit address-ordered no footer policy (accounting for all overheads)

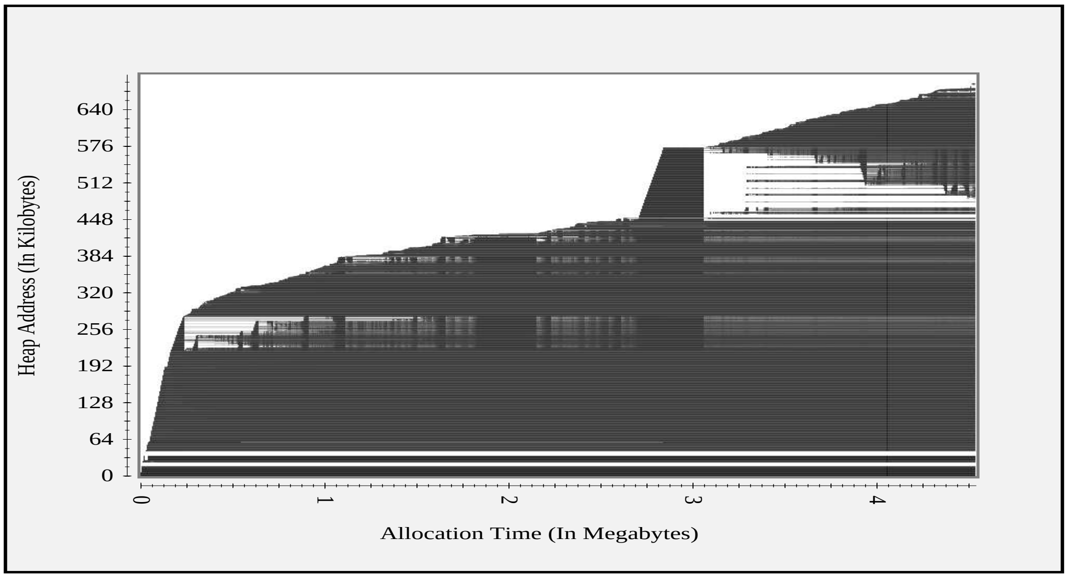

Figure 2.47: Fragmentation plot for P2C using the simple segregated storage 2^N policy (accounting for all overheads)

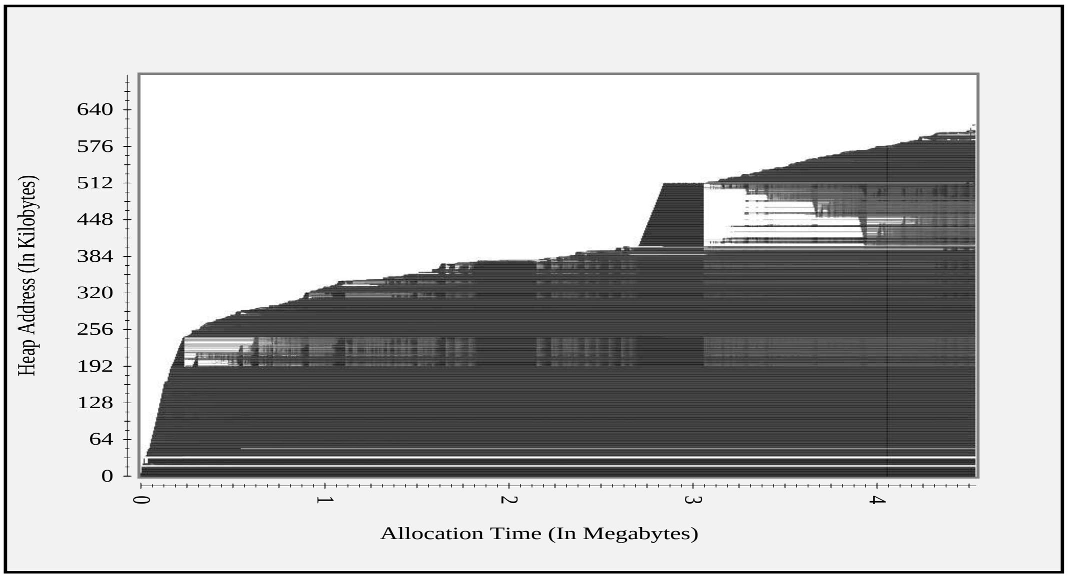

Figure 2.48: Fragmentation plot for P2C using the simple segregated storage 2^N & $3 * 2^N$ policy (accounting for all overheads)

2.14.6 Perl Allocation Graphs

The plots for the Perl program (Figures 2.49 to 2.53) show that it quickly reaches steady-state memory usage. Here again we can see the increased fragmentation due to the segregated-storage policy (Figures 2.52 and 2.53) compared to that of the best-fit LIFO (Figure 2.51) and first-fit address-ordered (Figure 2.50) policies. The linear allocator plot (Figure 2.49) shows that most objects only live a very short time, and thus the large black bands in the best-fit LIFO (Figure 2.50) and first-fit addressed-ordered (Figure 2.51) plots show these two policies aggressively reusing memory.

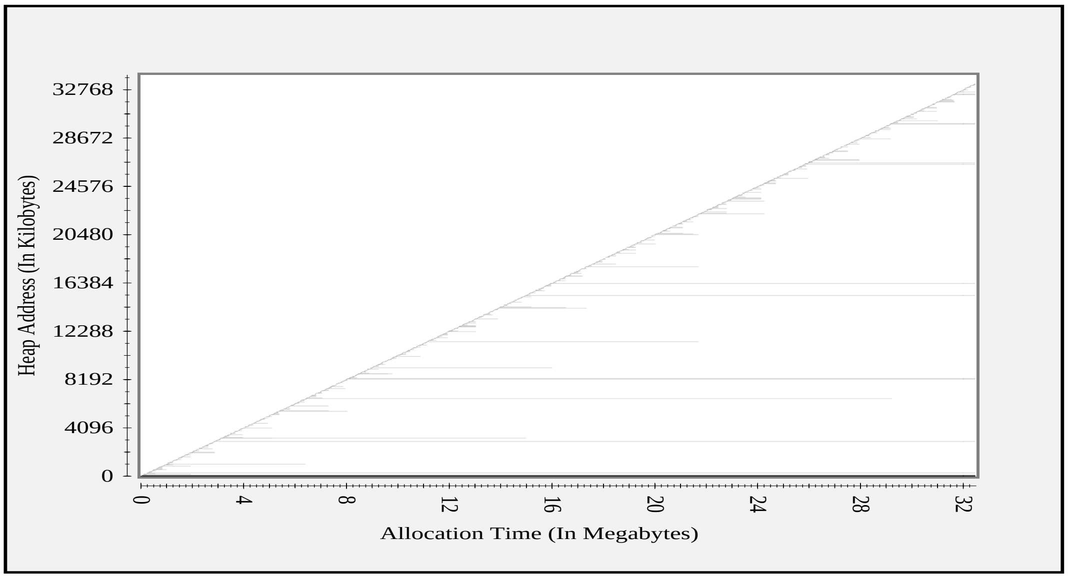

Figure 2.49: Fragmentation plot for Perl using the linear allocator

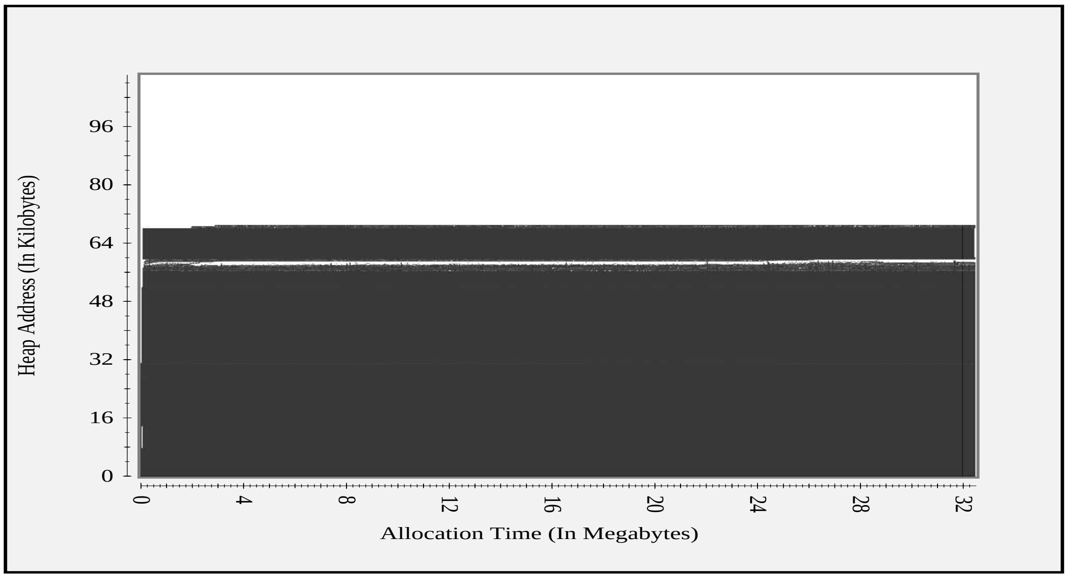

Figure 2.50: Fragmentation plot for Perl using the best-fit LIFO no footer policy (accounting for all overheads)

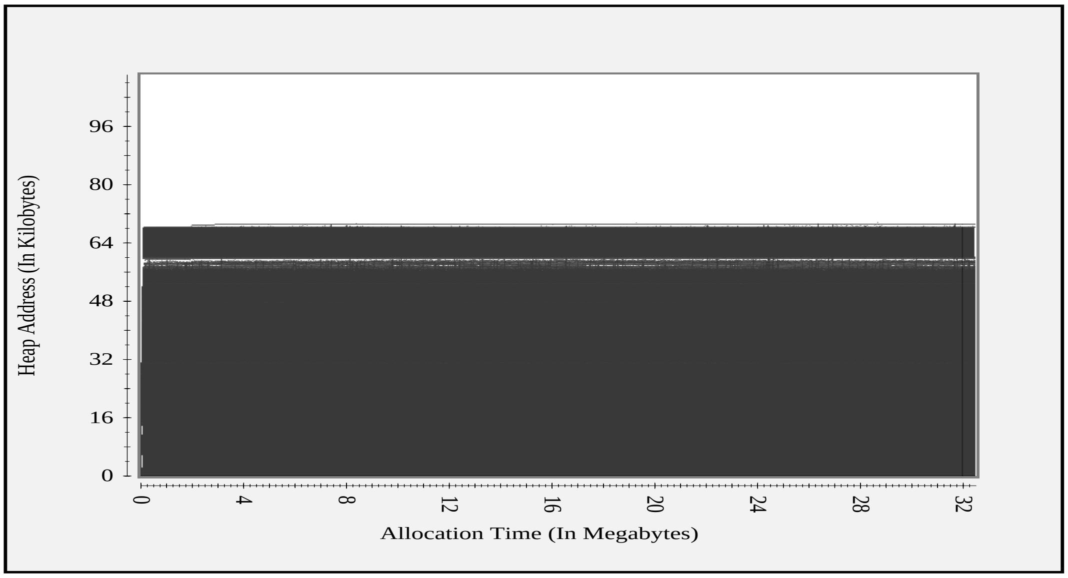

Figure 2.51: Fragmentation plot for Perl using the first-fit address ordered no footer policy (accounting for all overheads)

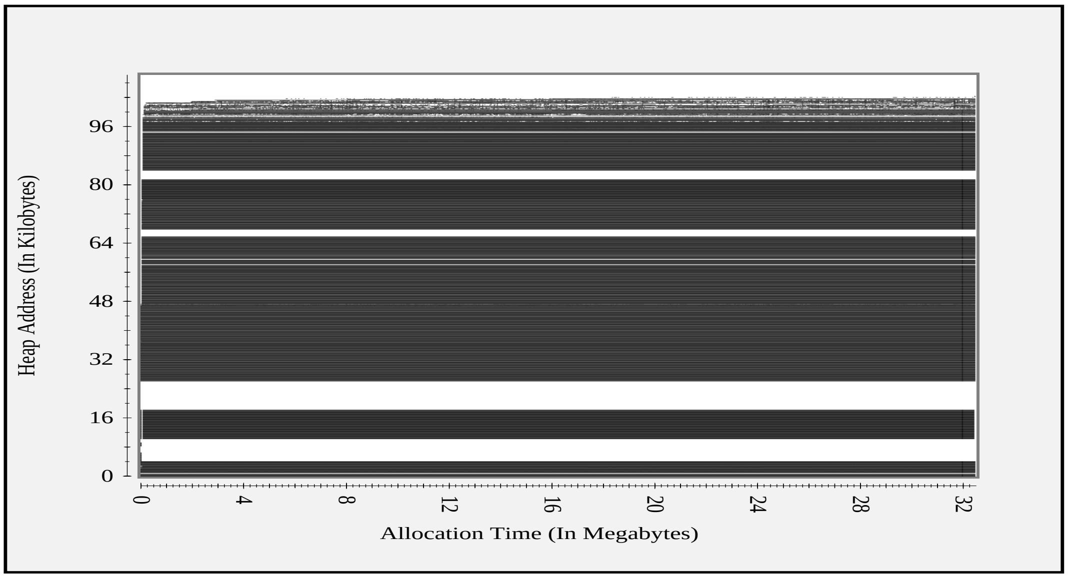

Figure 2.52: Fragmentation plot for Perl using the simple segregated storage 2^N policy (accounting for all overheads)

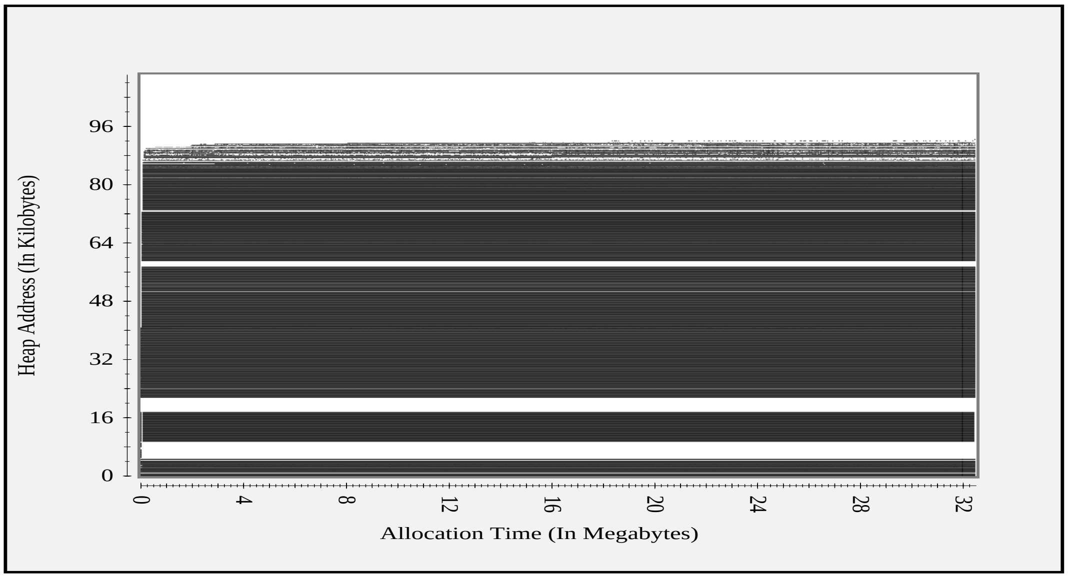

Figure 2.53: Fragmentation plot for Perl using the simple segregated storage 2^N & $3 * 2^N$ policy (accounting for all overheads)

2.14.7 LRUsim Allocation Graphs

The LRUsim program (Figures 2.54 to 2.59) continually allocates objects and finally frees them all at the end of the program. As with Ghostscript, the binary-buddy (Figure 2.55) and segregated storage (Figures 2.58 and 2.59) policies show the increased fragmentation caused by increasing internal fragmentation in a futile attempt to reduce external fragmentation, compared to the best-fit LIFO (Figure 2.56) and first-fit address-ordered (Figure 2.57) policies.

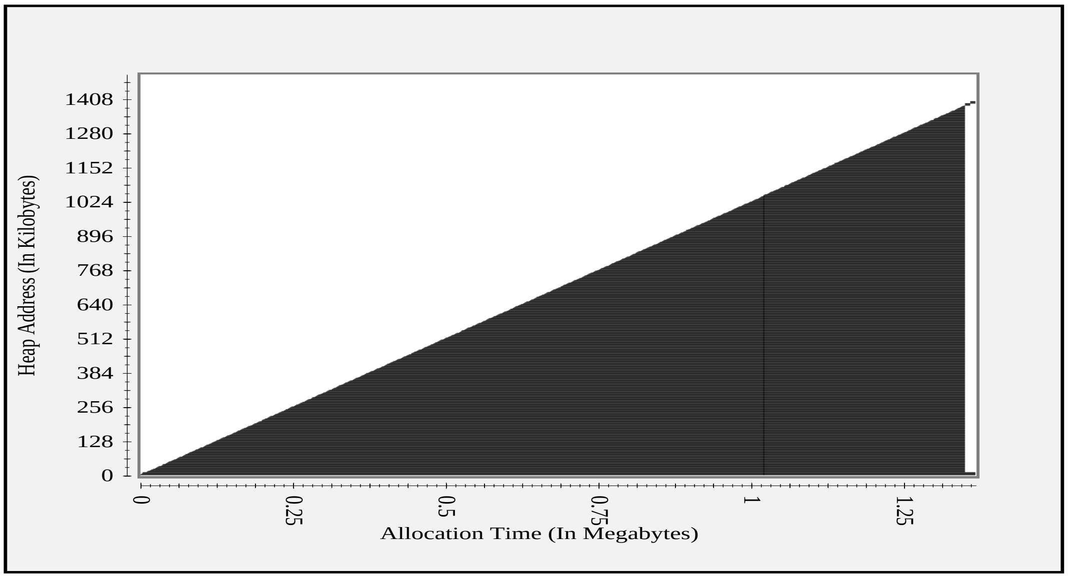

Figure 2.54: Fragmentation plot for LRUsim using the linear allocator

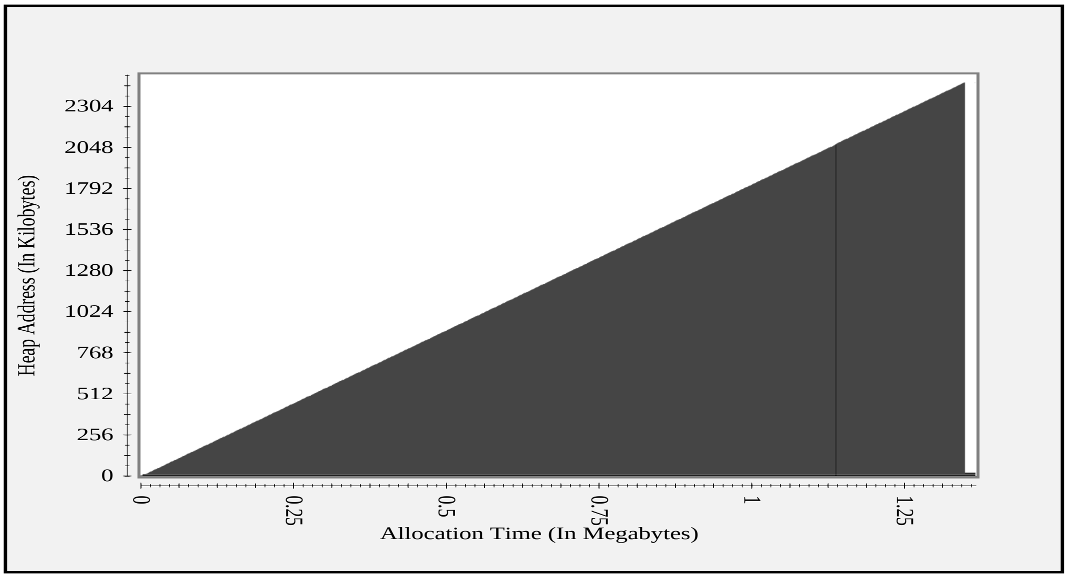

Figure 2.55: Fragmentation plot for LRUsim using the binary-buddy policy (accounting for all overheads)

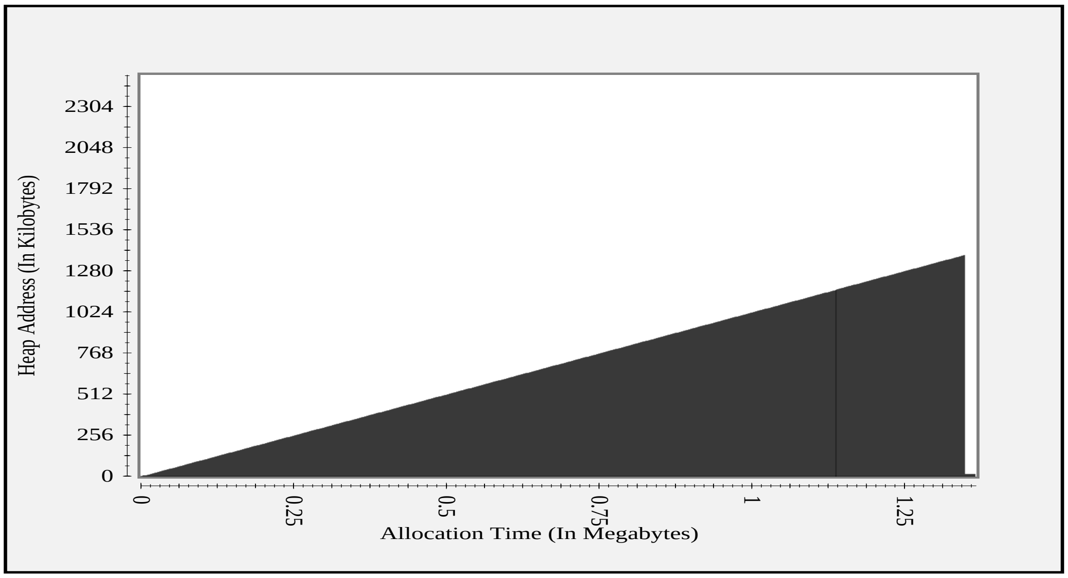

Figure 2.56: Fragmentation plot for LRUsim using the best-fit LIFO no footer policy (accounting for all overheads)

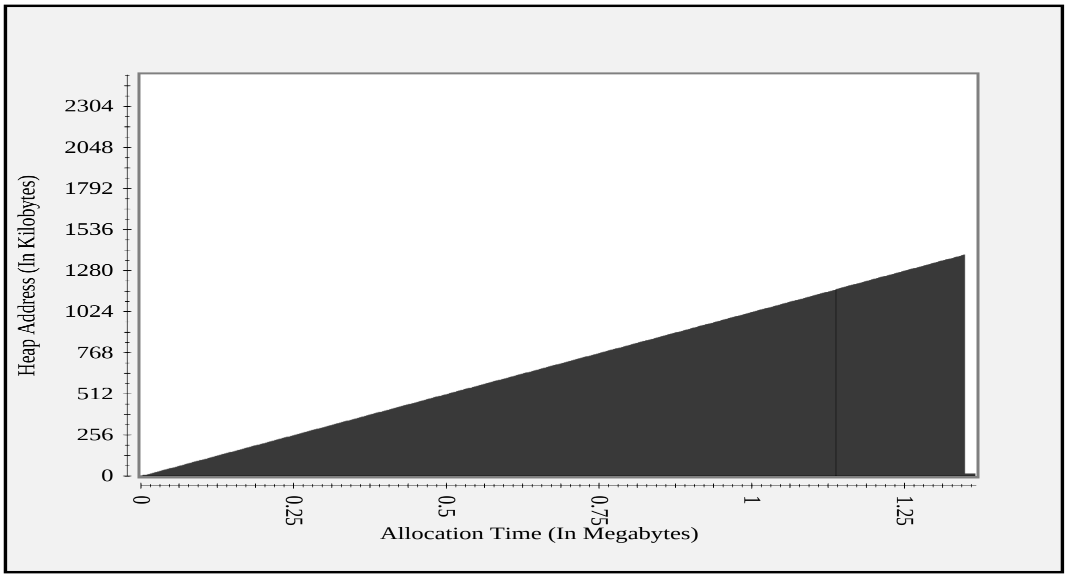

Figure 2.57: Fragmentation plot for LRUsim using the first-fit address-ordered no footer policy (accounting for all overheads)

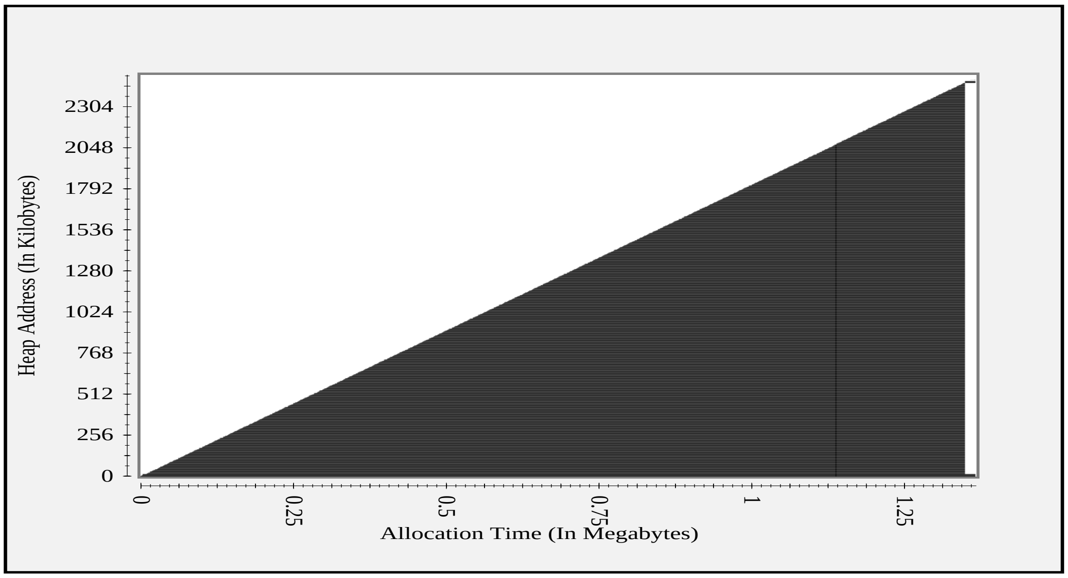

Figure 2.58: Fragmentation plot for LRUsim using the simple segregated storage 2^N policy (accounting for all overheads)

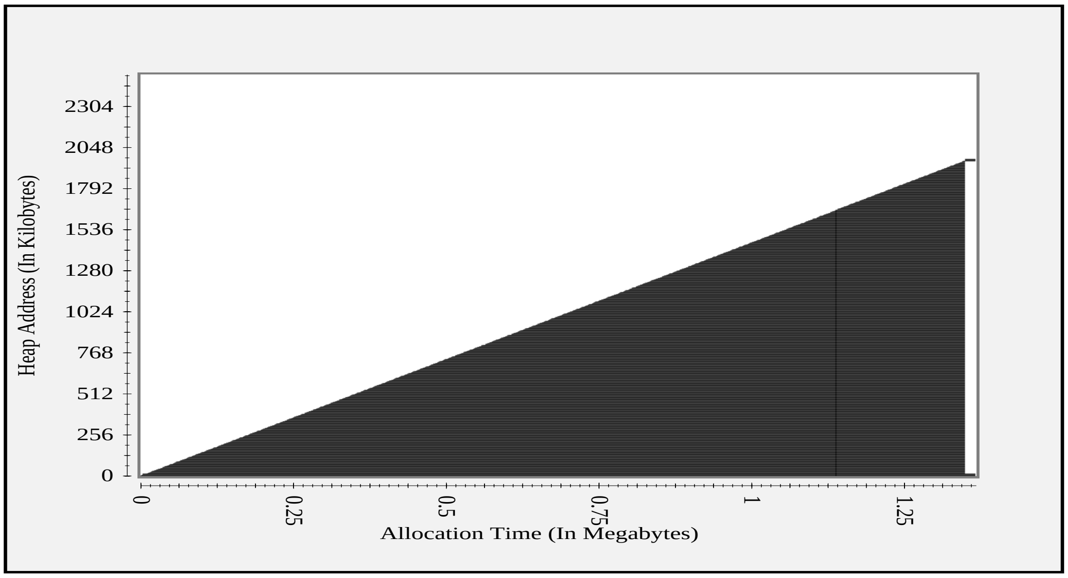

Figure 2.59: Fragmentation plot for LRUsim using the simple segregated storage 2^N & $3*2^N$ policy (accounting for all overheads)

2.15 Randomized Traces

In Sections 2.9 and 2.14 we showed that some algorithms that have been known for decades work extremely well. What is particularly surprising is that no one seems to have discovered how well they work. We have two hypotheses as to why these results have gone unnoticed for so long.

First, researchers have not been careful about separating policy from mechanism. As we saw in Table 2.2, the obvious implementations of our two best policies had considerable fragmentation (around 34%). However, when the overheads of the *implementation* were removed (Table 2.4), the allocator *policies* fared much better (around 0.80%).

Second, researchers have tended to use an experimental methodology that is fundamentally flawed. 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 advantage that it is easy to implement, and allows experiments to avoid idiosyncratic behavior specific to only a few programs. Often the people doing these studies went to great lengths to ensure that their traces had similar statistical properties to real programs. However, none of these studies showed the validity of using a randomly generated trace (no matter how well the random trace statistically modeled the original program trace) to predict performance on real programs. 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 published an extensive review of the relevant literature in [WJNB95], and describe at length the traditional methodology and why it is unsound. The interested reader is encouraged to see this paper for more details.

There are three basic reasons why randomly generated traces fail to represent real program traces. First, real programs tend to have strong phase behavior: programs tend to exhibit phases of computation where they build large data structures and finally discard all but a small amount of memory at the end of the phase. It is at these points that efficient use of memory by the allocator becomes most critical. However, in a synthetic trace, all of these phases have been smoothed out by the random sequence of allocation and deallocation requests.

The second reason that synthetic traces are not representative is that, with the exception of strings, real programs tend to allocate many objects of exactly the same size or of just a very few discrete sizes (see Section 2.12), while most synthetic trace generators allocate a large number of randomly chosen sizes which center around some mean.

The third and final reason that randomly generated traces are not representative of real programs is that in a randomly generated trace the relative proportions of different sized objects are relatively stable, whereas in a real trace the relative proportions will change dramatically based on the program's phase.

These three simple changes can have a major effect on how well a particular allocator performs.

Simply stated, an allocator's job is to *predict* the sizes and death times of objects that will be requested in the future. Using information from past memory requests, it chooses which free block of memory to allocate for the current request, such that future requests can be satisfied with a minimal amount of fragmentation. But it is the very information that a well-written allocator will use to predict these requests that is thrown out by a randomly generated trace.

One might say that a synthetic trace generator could be written to preserve the phase behavior, exact object size distributions, and exact proportions of real programs. But what should those phases look like? What sizes should be used? How do we vary the proportions? Until we have a much better understanding of allocation behavior, this information can only come from real traces.

To invalidate this methodology once and for all, we designed an experiment where we took actual memory allocation traces and changed just the order of the allocation and deallocation requests. The resulting trace is a random trace with the *exact* object size and lifetime distributions of the original trace. These traces, we claim, are much closer to a real trace than any of the randomly generated traces from other work. We then ran these traces through our fragmentation experiments (as described in Section 2.8).

The results for randomized traces (Table 2.10) differ in important ways from the results for real traces (Table 2.5). Most importantly, the rank ordering of allocators is different in several places, and the average fragmentation for most allocators is also too high—by over 11%. This is almost *thirteen* times the actual fragmentation of our best allocators. In addition, our best allocators (best fit LIFO and first fit address ordered) show over *six* times too much fragmentation.

The rank ordering is crucial, since the major purpose of simulation studies is usually to find the best policy. However, the expected amount of memory lost to fragmentation is also important because expected fragmentation is a major factor in deciding whether to use heap allocation or static allocation, or some more restrictive allocation scheme which exploits stereotyped usage patterns (such as GNU "obstacks").

Allocator	GCC	Espresso	Ghost	Grobner	Hyper	Perl	P2C	LRUsim	Avg	Std Dev
Lea 2.6.1	11.4%	12.9%	6.5%	6.9%	0.1%	9.7%	6.4%	0.9%	6.9%	4.6%
best fit LIFO NF	11.4%	16.0%	6.1%	6.9%	0.1%	9.7%	6.4%	0.9%	7.2%	5.3%
first fit AO NF	14.3%	11.3%	6.9%	6.9%	0.1%	9.4%	7.7%	0.9%	7.2%	4.9%
half fit	11.8%	17.5%	6.1%	6.9%	0.1%	15.0%	7.7%	1.3%	8.3%	6.1%
double bdy 5K	33.1%	100.3%	28.4%	16.8%	50.2%	28.1%	28.8%	35.7%	40.2%	26.1%
first fit LIFO NF	54.1%	70.1%	217.7%	194.0%	0.1%	30.6%	33.0%	1.3%	81.9%	98.3%
next fit LIFO NF	84.1%	59.3%	302.9%	254.2%	0.1%	35.9%	47.6%	1.3%	98.2%	115.5%
binary buddy	63.0%	48.4%	47.1%	41.2%	100.2%	67.3%	64.9%	79.7%	64.0%	19.4%
simp seg $3 * 2^N$	65.1%	117.1%	39.5%	32.4%	26.0%	37.5%	44.4%	43.1%	50.6%	29.2%
simp seg 2^N	76.3%	88.5%	46.5%	45.8%	26.0%	56.1%	62.9%	78.7%	60.1%	20.7%
Average	42.5%	54.2%	76.2%	61.2%	20.3%	30.0%	31.0%	24.4%		
Std. Dev.	29.2%	39.4%	112.8%	88.3%	33.0%	20.1%	23.4%	32.9%		

Table 2.10: Percentage actual fragmentation for selected allocators for all shuffled traces

The correlation between real and shuffled results for total memory waste (the memory used by our allocator *implementations*) across our selected allocators is .68; for actual fragmentation (the memory used by our allocation *policies*) it is .73.[30]

These correlations show that the results for shuffled runs are *suggestive* of the results for real programs. On the standard interpretation of correlation, however, the amount of variation in one set which is accounted for by regularities in the other set is proportional to the *square* of the correlation. This means that only 46% of the variation in wasted memory in the actual traces is accounted for by variation in the shuffled traces. Similarly, only 54% of the variation in actual fragmentation is accounted for by variation in the shuffled traces.[31] That is, for fragmentation, *the results of randomized experiments fail to account for half the observed variation due to real program behavior.* For both raw memory waste and fragmentation, simply randomizing the order of object creation (and keeping the size and lifetimes fixed) discards a large majority of the important information in a trace.

The most systematic effect of shuffling seems to be that simple segregated storage looks unrealistically good when compared to more sophisticated schemes. Next fit and first fit fare worst by an uncomfortable margin, while in reality they are better than binary-buddy systems and both simple segregated storage systems. This is a confirmation of our hypothesis that the random-walk nature of randomized traces tends to stabilize the proportions of objects of different sizes, which reduces external fragmentation by making it less important for a policy to adapt to the application's shifting needs for objects of different size classes.

Randomization not only changes the rank ordering, but it changes some allocators' rank dramatically—first fit LIFO and next fit move down 3 out of 10 places, and the simple segregated storage allocators move up 3 positions. Interestingly, the three best allocators keep their positions at the head of the pack, suggesting that the advantages of the best-fit and address ordered first-fit policies are unexpectedly robust across both patterned and randomized inputs.

The low correlation between real and shuffled results shows that the random methodology is inaccurate for the average performance of the nine allocators across our eight test traces. Examination of the results for individual allocators and individual traces show extreme errors in some cases. For example, for the best-fit LIFO no-footer allocator, the fragmentation results for Espresso are *sixty one times higher* for the shuffled trace (about 15.96%) as for real traces (about 0.26%).

---

[30] We use population correlation, rather than straight correlation, because we are sampling the space of possible programs and program executions.

[31] The correlation between real traces and shuffled traces for total memory waste across all of our allocators is .83, which means that about 69% of the variation in wasted memory in the actual traces is accounted for by variation in the shuffled traces. Similarly, the correlation between the real and shuffled traces for actual fragmentation across all of our allocators is .80, which means that about 63% of the variation in actual fragmentation is accounted for by variation in the shuffled traces. It should be noted, however, that our complete set of allocators has many more variants of the sequential-fit allocators than the other policies. This will tend to increase the correlation over a completely random sample. We therefore assert that the selected allocator results are more representative than the results for all allocators.

2.15.1 Another View of The Heap (Real Vs. Shuffled)

Using plots such as those from Section 2.14 we can show visually the difference between an allocator's performance with a real trace compared to a randomized trace. Here, we show the best-fit LIFO allocation policy using a normal (Figure 2.60) and a shuffled (Figure 2.61) trace of the GCC compiler, compiling the file combine.c. For the real trace, there is strong phase behavior, with large data structures coming free all at once. These features are completely absent from the shuffled trace. We also show the best-fit LIFO allocation policy for the Ghostscript program (Figures 2.62 and 2.63). Even though Ghostscript does not show strong phase behavior, the shuffled trace looks significantly different from the real trace. A full set of shuffled plots from the GCC compiler and the Ghostscript program for the nine selected allocators can be found in Appendix D.

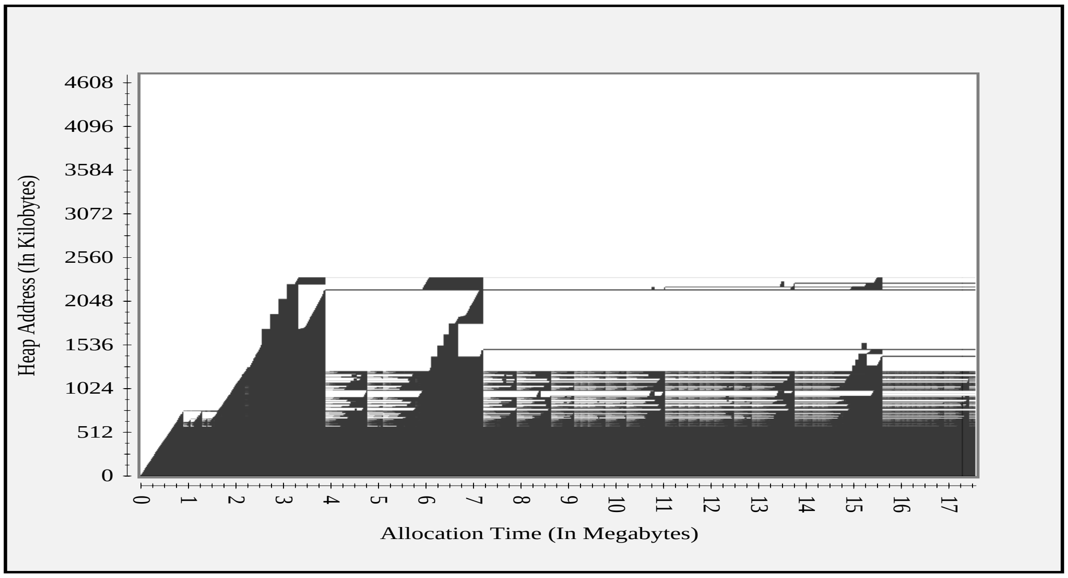

Figure 2.60: Actual fragmentation for GCC using the best-fit LIFO no-footer allocator (accounting for all overheads)

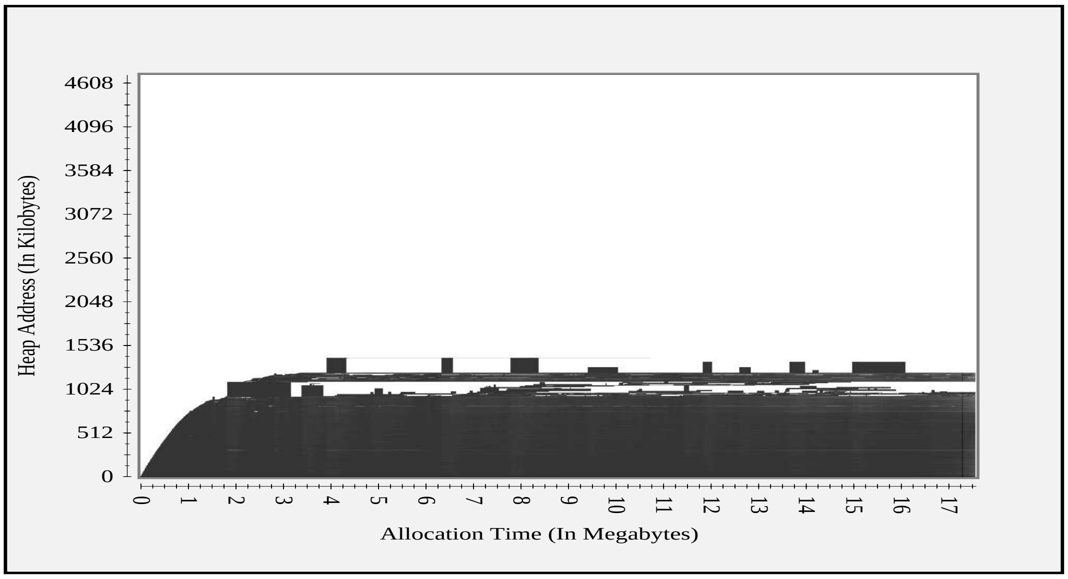

Figure 2.61: Actual fragmentation for GCC *shuffled* using the best-fit LIFO no-footer allocator (accounting for all overheads)

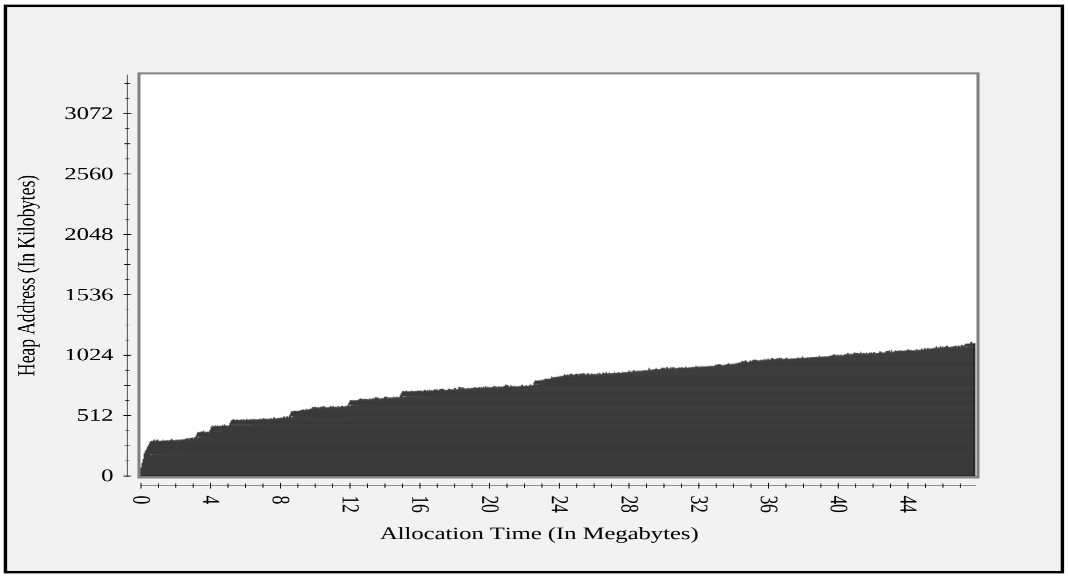

Figure 2.62: Actual fragmentation for Ghostscript using the best-fit LIFO no-footer allocator (accounting for all overheads)

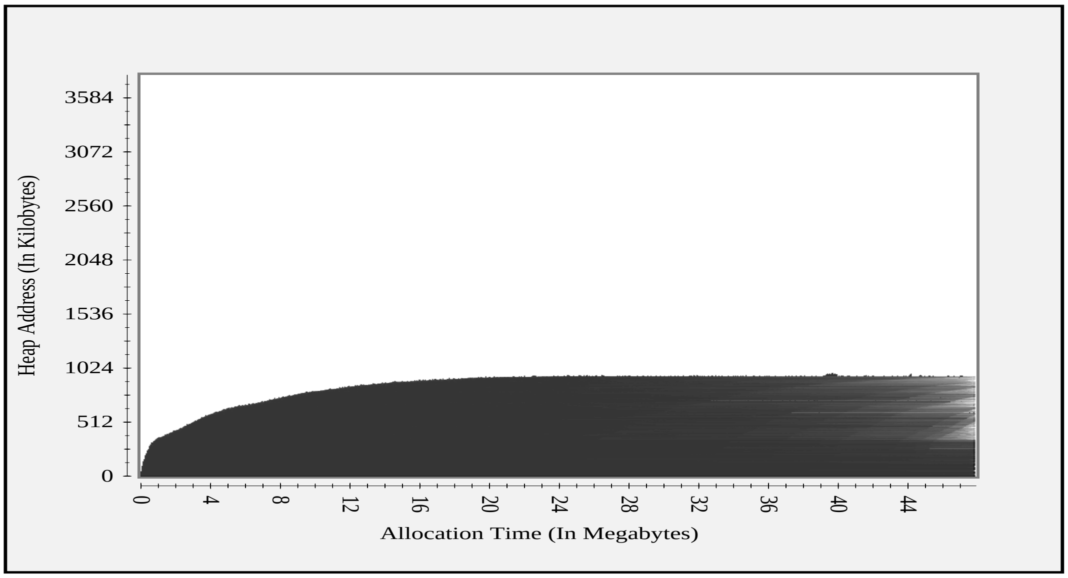

Figure 2.63: Actual fragmentation for Ghostscript *shuffled* using the best-fit LIFO no-footer allocator (accounting for all overheads)

2.16 Extrapolating These Results to Programs with Larger Footprints

The test programs that we selected for this research were constrained in a number of ways (see Section 2.6.1): the programs needed to be allocation-intensive, representative of a wide variety of problems, mostly leak free, and publicly available. We were able to identify eight programs that meet our criteria. Unfortunately, none of these eight programs had total heap sizes that were very large compared to the main memory size of modern computers.

We believe that the allocators that performed well did so because they use a sound strategy: they attempt to reuse memory in one part of the heap before reusing memory in other areas of the heap, giving objects in the latter areas longer to die and be coalesced with their neighbors. These strategies take advantage of the fact that objects allocated at about the same time tend to die at about the same time (Section 2.11), by placing these objects near each other in memory. The results for our allocators should therefore continue to hold for larger programs as long as this property of programs continues to hold.

2.17 Summary

In Section 2.9, we showed that the two best allocation policies, first fit address ordered free list with 8K allocation, and best fit address 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.

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. 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 much, and should only be considered if they make the implementation more efficient. Only three versions of next fit had less than 10% actual fragmentation, and all of those versions used an address-ordered free list.

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 applications, we believe this is one of our most significant findings.

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 tradeoff as a part of their basic strategy. However, the worst of our allocators, those that had over 50% actual fragmentation, tried to trade increased internal fragmentation to reduce external fragmentation, showing that this is not a good policy decision.

For good allocation policies, deferred coalescing appears not to cost much in terms of fragmentation. 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. In addition, we can not conclude that this difference is statistically significant at the 95% confidence level.

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, and neither policy reuses memory from one size class for objects in another size class. We believe that this is evidence that very coarse size classes generally lose more memory to internal fragmentation than they save in external fragmentation.

In Section 2.11 we showed that on average 90% of all objects have both of their temporal neighbors free after just 2.5K of allocation. Thus, if we allocate blocks from contiguous memory regions, and wait 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.

In Section 2.12 we showed that for most programs, the vast majority of objects allocated are of only a few sizes. 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.

In Section 2.13 we showed that small variations in policy can lead to large variations in fragmentation. For example, 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%. This shows that it is very important to carefully specify allocation policy when comparing allocator performance.

Korn and Vo suggest special treatment of the block of memory most recently obtained from the operating system. They call this a "wilderness preservation heuristic," and report that it is helpful for some allocators [KV85]. However, our results (in Section 2.9) show that for the best allocation policies (best fit and first fit address ordered), special treatment of the wilderness block is unnecessary.

In Section 2.15 we showed that the results of our experiments with randomized traces fail to account for half the observed variation due to real program behavior. For both raw memory waste and fragmentation, simply randomizing the order of object creation (and keeping the sizes and lifetimes fixed) discards a large majority of the important information in a trace.

The low correlation between real and shuffled results shows that the random methodology is inaccurate for the average performance of the nine allocators across our eight test traces. Examination of the results for individual allocators and individual traces show extreme errors in some cases. For example, for the best-fit LIFO no-footer allocator, the fragmentation results for Espresso are *sixty one times higher* for the shuffled trace (about 15.96%) than for the real trace (about 0.26%).

We believe that our experiments have shown definitively that the traditional methodology of allocator evaluation is unreliable. We have shown that useful results can be obtained by the more reliable method of realistic trace-driven simulation. Sadly, we believe this should have been obvious for decades, and that there was never a compelling technological reason for accepting the potential errors—serial storage for real traces has never really been that expensive, given that the cost is fairly stable relative to run times of real programs. Tracing real program allocation behavior is easy: all that is necessary is to use a modified allocator that records events. Simulation is likewise easy: it only requires being willing to let simple programs run many times with different allocators, using otherwise idle CPU time.

We have shown that for a large class of programs, the fragmentation "problem" is really a problem of poor allocator *implementations*, and that for these programs well-known policies suffer from almost no true fragmentation. In addition, very good implementations of the best policies are already known. 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]. Most importantly, an excellent allocator implementation that runs on many platforms was written by Douglas Lea and is freely available [Lea]. This allocator was improved partly due to the results in our original survey [WJNB95], and is now a very close approximation of best fit.

If these results hold up to further study with additional programs, we arrive at the conclusion that the fragmentation problem, which computer scientists have been trying to solve for over 30 years, is not a problem of finding good allocation *policies*, but is actually a problem of *recognizing necessary overheads in existing implementations.*

Chapter 3

Locality

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, 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. A good deal of research on cache design has been done to improve the locality of reference of programs, particularly those in the SPEC benchmark suites. 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.[1]

The only paper on this topic we were able to find was [GZH93]. This paper's key contribution was showing that the choice of memory allocator can affect the locality of the application. Unfortunately, its authors failed to separate the locality effects of the allocation *policy* from those of their *implementation*. Thus, for the memory allocation policies that fared worst, they could not be sure if the poor locality was because the policy itself has inherently poor locality, or because their implementation of the policy had poor locality. This paper was also limited in its breadth, in that the authors only studied a small number of allocators. We remedy this weakness by accounting for all the locality effects of the memory allocator *implementation*, and then carefully varying the policy decisions so that we can measure the individual effects of these policy decisions on the locality of reference of the application.

In Chapter 2 we showed that, for a large class of programs, fragmentation is not a problem, and developed a strategy (Section 2.10) for allocating memory that subsumes the two best policies (best fit and first fit address ordered). Fragmentation alone, however, is not the main concern of most programs: programs are often far more constrained by their locality of reference. It would often be desirable to sacrifice some fragmentation if that would increase locality. In this chapter, we show that it is unnecessary to sacrifice fragmentation to improve locality of reference. The best policies in terms of fragmentation are also the best policies in terms of locality.

[1]This is possibly because the SPECINT95 benchmark suite only contains two programs that have any dynamic memory allocation at all.

We begin this chapter with an introduction to memory hierarchy, focusing on both the cache and virtual memory levels. We then discuss several factors that can affect locality, and present a method of measuring locality of reference. We follow this with a discussion of our methodology and experimental design. We then present our results, and develop a method of comparing locality of reference to fragmentation. Next, we present a graphical representation of the heap to show how different allocator placement policies affect locality. Finally, we show how these results relate to garbage collection, present our conclusions, and outline some future work.

3.1 Background

This section presents background material on locality and memory hierarchy design. The reader already familiar with this material can safely skip ahead to Section 3.2.

3.1.1 Memory Hierarchy

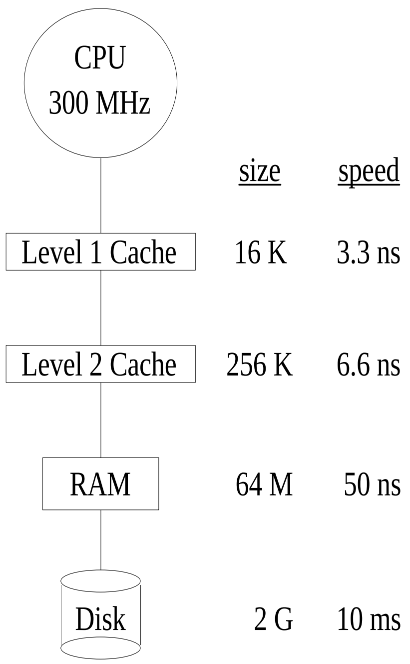

Figure 3.1: The levels in a typical memory hierarchy

In an ideal computer system, memory would be infinitely large and uniformly fast. Unfortunately, fast memory is very expensive. To keep costs reasonable, modern computers are built using several kinds of memory: on-processor fast caches, slower and larger off-chip static RAM caches, still slower and larger dynamic RAM main memories, and finally very slow and much larger disk-based memories. Each memory area is a strict subset of the next memory area, so that if a memory request is not available at one level, an attempt is made to find it in the next level.

In Figure 3.1 we show an example of a memory hierarchy. The first attempt to load a word of memory is made in the on-chip primary, or level 1, cache. In this example, the size of the level 1 cache is just 16 kilobytes. If the word is not present in the level 1 cache, then the processor looks in the off-chip secondary, or level 2, cache. In this example, the size of the level 2 cache is 256 kilobytes, and accessing this cache takes twice as long as accessing the level 1 cache. If the word is not present in the level 2 cache, then main memory is searched. In this example, the size of main memory is 64 Megabytes, and accessing main memory takes 8 times as long as accessing the level 2 cache, (16 times as long as accessing the level 1 cache). Finally, if the word is not present in the main memory, the processor looks on disk. In this example, the size of the disk-based memory is one gigabyte, and accessing a word on disk takes two hundred thousand times as long as accessing the main memory, and 3.3 million times as long as accessing the level 1 cache.

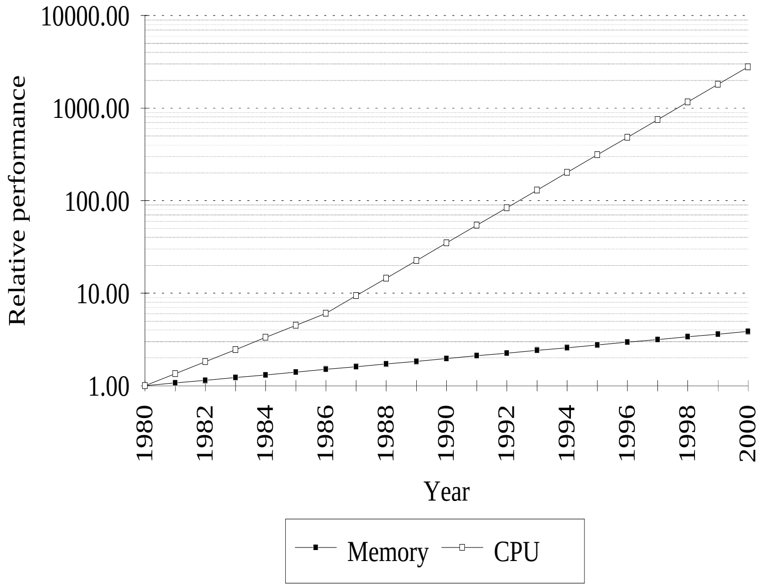

Figure 3.2: Relative performance of memory and CPUs

Figure 3.2, from [PH96], shows the performance of CPUs relative to main memory over time. The numbers for 1997-2000 are projections. As can clearly be seen, processors are increasing in speed much faster than main memory. This increasing disparity will continue to increase the performance penalty for accessing memory that is not in the primary cache. It is therefore desirable to attempt to place memory words that will be used frequently into the primary cache whenever possible. Most programs exhibit a property called *locality of reference*, which we will describe in the next section. It is this property that allows us to build computers with a memory hierarchy and still achieve good utilization of the CPU.

3.1.2 Locality of Reference

Programs tend to reuse data and instructions they have used recently; this property is called *locality of reference.* 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 is the property that programs tend to access data and instructions that have been accessed in the recent past. The purpose of a memory hierarchy is to make temporal locality look more like spatial locality to the processor. In other words, the purpose of a memory hierarchy is to place the data and instructions that have been accessed in the recent past into a fast subset of the memory so that they can be accessed more quickly.

Cache Memory

A *cache* memory is a small, fast memory located close to the CPU that holds the most recently accessed code or data [PH96]. While there are many possible implementations of a cache, they all share some common properties: they hold a small subset of the total memory, they are very fast to access, and they attempt to exploit locality of reference. The design of a cache must answer four basic questions [PH96]:

1. Where can a block be placed?
2. How is a block found if it is in the cache?
3. Which block should be replaced if a new block is brought into the cache?
4. What happens when data is written to the cache?

All cache designs operate on blocks of memory of some fixed size, usually four or eight contiguous words, called *cache lines.* Since memory access times are governed both by the memory access speed, or latency, of the memory and by the bandwidth of the memory bus, it is faster to load some number of contiguous words than to load the same number of random words. The principle of spatial locality states that if one of the words in a line is accessed, it is likely that the rest of the words in the line will be accessed soon. This probability decreases as the line size increases. Thus there is a tradeoff between the time savings of loading a large line, the bandwidth cost of using the memory bus, and the space cost of filling up more of the cache with words that are less likely to be accessed soon. The choice of an appropriate line size is largely determined by the bandwidth and latency of the memory system, and is fixed early in the design process of a computer.

Accessing a word of memory that is found in the cache is known as a *cache hit.* A *cache miss* occurs when a word of memory that is not in the cache is accessed. If this is the case, then the block containing the accessed word is loaded into a line in the cache. There are three different kinds of cache misses: *compulsory* misses, *capacity* misses, and *conflict* misses. Compulsory misses occur when a word of memory is accessed for the first time. Capacity misses occur when a word of memory was in the cache at one time, but was *evicted* because the cache was too small and room was needed for another word. Conflict misses occur when a cache attempts to place two words of memory in the same part of the cache. One of the words is evicted to make room for the other, *even if there are other unused cache lines.*

Next, we will describe and compare several cache designs. Perhaps the simplest cache design is called a *direct-mapped* cache. In this design, when a word of memory is loaded into the cache, it can go into only one place, determined by the middle bits of its address. The words of memory that previously occupied this line are *evicted* to the next lower level in the memory hierarchy. This type of cache is fast, very easy to build, and works fairly well. It is, however, susceptible to *conflict* misses. In this design, a conflict miss occurs when two words of memory in different blocks happen to map to the same cache line.

To combat the problem of conflict misses, some caches (particularly level 1 caches) are designed to be *set-associative.* A set-associative cache allows a set of cache lines (usually two, four, or eight) to map to the same cache location at the same time. The number of cache lines in a set determines its associativity. Thus, a cache with two lines per set is said to be 2-way set associative. If a program repeatedly accesses two memory words that would cause a conflict miss in a direct-mapped cache, the two blocks can co-exist in a 2-way set-associative cache, each in its own cache line. As a rule of thumb, a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size N/2 [PH96]. Unfortunately, as set size increases, access time also increases. Thus, there is a fundamental tradeoff between the probability of a cache hit and the time it takes for that hit to occur. Most modern computers use a 4-way set-associative cache, with advanced designs moving towards 8-way set-associative caches.

Another cache design, which is rarely implemented, is a *fully-associative* cache.[2] In this design, a block of memory can occupy any cache line. Thus, there are never any conflict misses in a fully-associative cache. This cache design is most useful for cache simulations. By simulating a theoretical fully-associative cache and comparing it to a set-associative cache, one can determine what percentage of misses are conflict misses and what percentage are capacity misses. This information can then be used to optimize the set-associativity of the cache.

Another important issue in cache design is deciding which line should be evicted when room must be made for a new block. There are several possibilities: LIFO, FIFO, a random block (random), Least Frequently Used (LFU), or Least Recently Used (LRU). In practice, most caches implement *pseudo*-LRU, which is essentially LRU with a small amount of randomness.

[2] A fully-associative cache is often used as part of the branch predict unit of the processor, and some advanced designs are using a very small fully associative cache as a level 0 cache.

The final issue in cache design is what to do for a write. There are two basic policies when writing causes a cache hit: *write through* and *write back*. In a write-through cache, the information is written to both the block in the cache and to the block in the lower-level memory. In a write-back cache, the information is written only to the block in the cache. On a cache miss, there are also two basic policies: *write allocate* and *no-write allocate*. For a write-allocate cache, the block is loaded on a write miss, followed by one of the write-hit actions above. For a no-write-allocate cache, the block is modified in the lower level memory, but not loaded into the cache. In this research, we only studied write-allocate caches.

Virtual Memory

Although most people think of a computer's RAM as being its main memory, in most modern computers, the RAM is just a cache for the larger disk-based *virtual memory* [Den70]. Thus, many of the same locality issues exist at the virtual memory level as do at the cache level of the memory hierarchy. In particular, because disk storage is around 1,000,000 times slower than RAM, achieving good locality of reference is very important. As we will see in Section 3.6, a dynamic memory allocator's placement choices can have a dramatic effect on locality. We will also see that the placement policies that improve cache locality are sometimes at odds with the placement policies that improve virtual memory locality. Fortunately, the best policies in terms of fragmentation (best fit and first fit address ordered; see Section 2.9) are among the best policies in terms of locality of reference at both the cache and virtual-memory levels of the memory hierarchy.

Virtual memory is implemented by special hardware called the *memory management unit* (MMU). The MMU maintains a mapping in the *page table* between the *physical address* of the data and instructions in RAM and the *virtual address* of the data and instructions as seen by the processor. A small cache combined with special purpose parallel hardware, called the *translation lookaside buffer* (TLB), makes this mapping fast. Locality issues also apply to the TLB, since it works as a cache for physical to virtual address mappings. We did not study issues regarding the TLB in this dissertation.

An additional role of the MMU is to cache the recently used areas of memory in RAM. This "cache" is typically fully associative with a line size (called a *page*) of 4K bytes to accommodate the dramatic difference in access time between RAM and disk. The choice of which page to replace is left to the operating system. Although several replacement algorithms have been used in the past, virtually all modern operating systems use LRU replacement on a miss.

3.2 Effects on Locality

It has often been observed that programs exhibit locality of reference. But what is not often appreciated how this locality can be affected:

- *The choice of data structures.* Different data structures have different locality characteristics. A hash-table, for example, has properties that work against good spatial locality of reference. The idea behind a hash-table is that subsequent references access unrelated areas of the table to improve performance by reducing the chance of collisions. However, this same property reduces the spatial locality of the program. On the other hand, a splay-tree [ST85] attempts to increase performance by increasing temporal locality. The tree is reorganized after every access to move recently accessed objects nearer to the top of the tree. Thus, when recently accessed nodes are again accessed, fewer tree node traversals will be required. If there is locality in the way the objects are accessed, then this will improve the temporal locality of the program.

- *The choice of algorithms.* Operations on a matrix, for example, can have very different locality characteristics if they are performed in row-major order or in column-major order. This often interacts heavily with the choice of language and compiler.

- *The language and compiler.* The language and compiler affect locality by their object and code layout policy. For example, if the language specifies that an array should be laid out in row-major order, and the program's algorithm traverses this array in column-major order, then the program will have poor spatial locality. Likewise, if the compiler places commonly called routines at very different addresses, particularly if these addresses contend for the same cache line, then the program will have very poor locality of reference.

- *The memory allocator's placement of dynamic objects.* The memory allocator has complete freedom to place dynamic objects in any unallocated area of the heap. If objects that are referenced at about the same time are placed in adjacent memory locations, then the program will exhibit good spatial locality. If, on the other hand, objects that are referenced at about the same time are placed in such a manner that they contend for the same cache line, then the program will exhibit poor spatial locality.

It is this last effect on locality that we studied for this research. Surprisingly, [GZH93] was the only work we could find that studied the effect of non-moving memory algorithms on the locality of reference of programs. The authors of [GZH93] also found it surprising that this seems to be an entirely unexplored research area.

Having settled the issue of how different allocation policies affect fragmentation, we wanted to study how these same policy decisions affect locality of reference. There are times when it might be desirable to trade small amounts of increased fragmentation for substantially better locality of reference. This research will show that this is an unnecessary compromise.

3.3 Measuring Locality

As was the case with fragmentation (Section 2.8.2), it is difficult to attach a number to locality of reference. Locality can be measured in many different ways, and can include the effects of part or all of the memory hierarchy. Locality can be measured for only heap accesses, all data accesses, or all accesses (data and instructions). Each has its merits:

- **Heap only.** The allocator's placement decisions only affect the locality of reference of objects on the heap. The locality of instructions, globals, and the stack are unaffected by the choice of allocation policy.[3] However, no level in the memory hierarchy of any modern machine caches loads and stores to the heap alone. Thus, any measurement only consisting of touches to the heap will not be representative of the actual locality of reference on a real machine.

- **All data references.** On a machine with split instruction and data caches, we can measure the cache locality of the data references independently of instruction locality. We cannot, however, measure the data cache locality without also measuring references to the stack and globals.

- **All references.** At the virtual memory level of the memory hierarchy, all references become important when measuring locality. Even though references to the heap, stack, globals, and code all occur on different pages of memory, they cannot be measured independently at the virtual memory level. At the virtual memory level the performance of a program is a complex interaction of these four kinds of references, and any measurement not including all four would be meaningless.

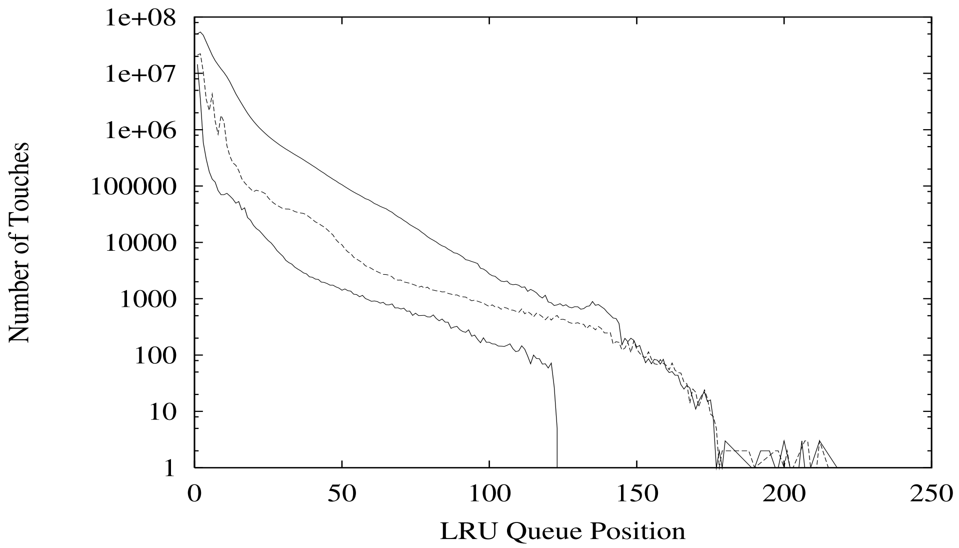

Figure 3.3: A histogram of touches to each position in the virtual memory's LRU queue for Espresso using Lea's 2.6.1 allocator

Figures 3.3 and 3.4 illustrate these differences. Figure 3.3 is a histogram of the touches to each queue position in the virtual memory's LRU queue for the Espresso program using Lea' 2.6.1 allocator. The bottom (solid) line is a measurement of only references to the heap, the middle (dashed) line is a measurement of all data references, and the top (solid) line is a measurement of all references. This plot can be transformed into a plot of miss rate versus memory size by summing the area under the curves. Figure 3.4 is the miss rate equivalent of Figure 3.3. As can be seen from this figure, the miss rate changes dramatically depending

[3]The *implementation* of the allocator will affect the locality of instruction access. However, as we will see in Section 3.5, we are able to factor this effect out of our measurements.

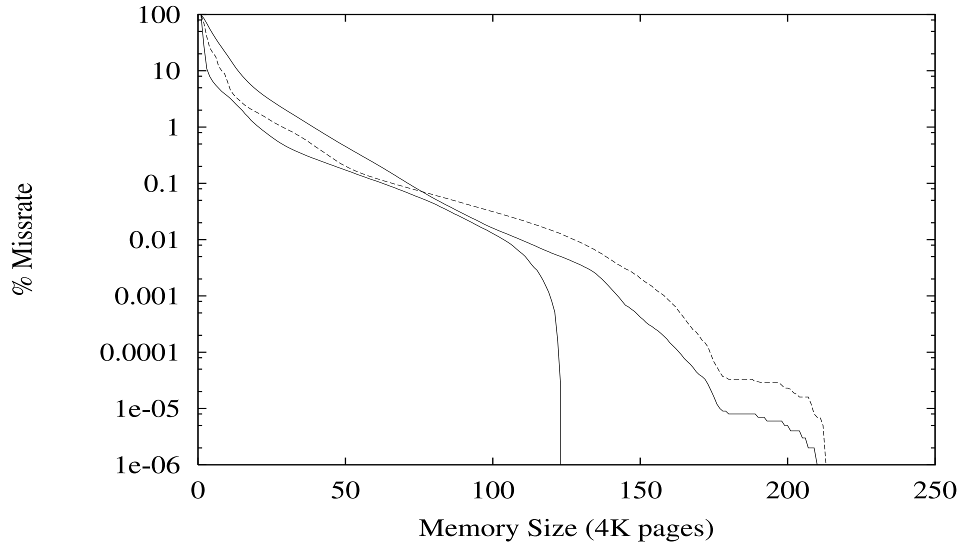

Figure 3.4: The miss rate of Espresso using Lea's 2.6.1 allocator

on where it is measured. In particular, the miss rate for all data references (dashed line) is actually *higher* than the miss rate for all references (higher solid line) for memory sizes above about 60 pages.

In this research, we measured locality at two levels in the memory hierarchy: the cache level and the virtual memory level. For the locality of reference to the cache, we measured the miss rate for three sizes of 8-way set-associative and fully-associative write-allocate data caches. These measurements include all data references, and exclude all instruction references. We report miss rates relative to the number of data accesses to the cache. Thus, a miss rate of 1% means that 1 out of 100 data references was not in the cache being simulated. This is a reasonable measure of locality because a program that exhibits better locality of reference will miss the cache less often. We measure locality for a high-associativity cache to attempt to avoid having our results skewed by conflict misses caused by the application. Of course, these results will be more applicable to machines that implement an 8-way set-associative write-allocate cache than to other machines. We do, however, believe that these results will be representative of the allocators' performance on other machines.

For locality of reference at the virtual memory level, we measure the amount of memory needed to maintain a particular CPU utilization. In other words, we measure how much memory we would need for a given program in order to keep the miss rate low enough that the CPU can maintain a given utilization. For these measurements, we include all memory references. Thus, if a program requires 100 pages to achieve 90% CPU utilization when linked with a particular memory allocator, and requires 110 pages when linked with a different allocator, then we can conclude that the second allocator requires 10% more memory than the first allocator. This is a reasonable measure of locality because a program that exhibits better locality of reference will need less memory to have the same CPU utilization.

We measure locality in different ways for the cache level and virtual memory level of the memory hierarchies for an important reason: while programs are often written to fit into a fixed size of memory, they are not generally written to accommodate a particular cache size. This is because program performance generally degrades gracefully with respect to the size of the cache, but can degrade dramatically if the amount of available RAM is exceeded. Thus, we wanted a way to measure locality at the virtual memory level that we could confidently say was not interacting in an unusual way with the design of the test programs. This measure also enables us to perform some interesting comparisons between locality and fragmentation that would not have been possible if we just measured miss rate.

There is an interesting relationship between fragmentation and locality at the virtual memory level. A very loose definition of fragmentation is: the amount of additional memory needed to run a program. If a program linked with a particular allocator uses 10% more memory than it requests, then we say that this allocator produces 10% fragmentation for that program. Similarly, if one allocator requires 10% more memory than another to maintain the same CPU utilization, then, using this loose definition of "fragmentation," that allocator produces 10% more "fragmentation" than the other allocator for that program.

3.4 Experimental Methodology

When studying the effect of a dynamic memory allocator's placement choices on the locality of reference of an application, it is very important to separate policy from mechanism. A given implementation of a memory allocator can dramatically affect the locality of an application that uses it. For example, a best-fit allocator that exhaustively searches a free list for the best block will have very different locality characteristics from a best-fit allocator that maintains a bitmap of free words, or one that uses a binary tree of object sizes. However, the locality effects due to the *allocation policy* remain the same. We believe that once good policies are discovered, it will be relatively simple to design good implementations of those policies.

This presents an interesting question in experimental design: how does one study the locality effects of a memory allocation policy without implementing the policy? And if one must implement the policy, how can the locality effects of policy be measured without also measuring the implementation of that policy? Our answer to this problem, which we describe in the next section, is to implement the policy, and use it with real programs, but to filter out all effects of that implementation.

3.5 Experimental Design

In our locality experiments, we used six of the eight test programs described in Section 2.6: Espresso, Ghostscript, Grobner, Hyper, P2C, and Perl. We did not use the GCC compiler, because, as we described in Section 2.6.2, the allocation behavior of this program was reconstructed using a post-processor to remove the effects of obstacks. We also did not use the LRUsim program because its run time would have been unreasonable given the hardware available to us. With the six programs we used the 53 allocators described in Section 2.5, giving us a total of 318 program-allocator pairs.

Because locality of reference is determined by the interaction of several independent factors—the properties of the program, the placement choices of the allocator, and the design of the memory hierarchy to name just a few—locality can only be reasonably studied using actual programs, allocation policies, and memory hierarchy designs. We accomplished this by using *trace-driven simulation* (Section 2.7). We gathered a trace of the instruction and data references of our test programs using the *Shade* instruction-level trace gathering tool [CK93].

Shade [CK93] is a program that generates exact instruction level traces of any program that runs on the Sparc architecture (V8 & V9). By using Shade and a small routine we wrote, we were able to generate dinero format [Hil87] instruction-level traces for each combination of test program and allocation policy. A dinero trace of a program is just an ascii trace of the loads, stores, and other instructions of that program, along with their corresponding addresses.

We were able to measure placement policy effects without measuring the effects of our implementation by slightly modifying the implementation of each memory allocation policy to store a known value into a global variable as the first and last instructions of malloc, free, and realloc. We then wrote a small filter program to look for stores to this variable in the dinero trace generated by Shade, and remove the allocator instructions. In this manner, we were able to obtain an exact instruction level trace of a program using a particular memory allocation placement policy with none of the noise of the implementation of that policy.

It is important to measure the effect on locality of allocator *policy* without measuring the effect of an *implementation* of that policy. By separating the policy effects from the implementation effects, we were able to directly study the effects of placement policy without having to worry about measuring effects which are simply due to questionable implementations of those policies.

3.5.1 Cache Simulations

We measured the effects on locality of allocator placement policies at two levels in the memory hierarchy: the cache level and the virtual memory level. To measure locality at the cache level, we used the tycho cache simulator. Tycho [Hil87] is a trace-driven cache simulator that can simulate many alternative direct-mapped, set-associative, and fully-associative caches with one pass through a dinero format address trace to produce a table of miss ratios. In our experiments, we configured tycho to generate cache miss ratios for 16K, 64K, and 256K 8-way set-associative and fully-associative caches. We processed filtered dinero traces for each program-allocator pair with the tycho cache simulator, and measured the cache miss rate for each of our programs linked with each of our allocators. By comparing the miss rates of the different policies as measured across all of our programs, we were able to identify many of the locality effects of allocator placement policy. We present these results in Section 3.6.

3.5.2 Virtual Memory Simulations

To measure locality of reference at the virtual memory level, we wrote a simple virtual memory simulator which we called VM-simulator. VM-simulator simulates LRU page replacement using 4K pages in a virtual memory system. VM-simulator takes as input a filtered dinero-format trace and produces a histogram of the number of hits to each queue position in the LRU queue used for page replacement. From this information, we can compute the memory size needed to achieve a particular CPU utilization. The computation is quite simple. Assuming a 100 MIP processor, and a 10 millisecond hard-disk, any memory reference that is not in RAM takes approximately 1,000,000 instructions to access. In order to achieve 50% CPU utilization, for every disk access we must execute 1,000,000 instructions. We then sum all touches in the LRU histogram, starting with the first queue position, until we exceed 1,000,000 touches. This histogram position represents the number of 4K pages that we would need for that program to achieve 50% CPU utilization. The computation is similar for 10% and 90% utilization.

Note that this measurement is similar to measuring the miss rate. From this LRU histogram, it is easy to compute the miss rate for a given memory size. The miss rate is the sum of the LRU histogram positions normalized by the number of references.

This notion of locality is particularly interesting when compared to our fragmentation results (Chapter 2). The disk subsystem is a very expensive resource in terms of its effect on computation speed. For many applications, the total amount of memory used is not as important as the number of pages of RAM that are required to prevent significant paging. Because main memory is quantized (e.g., 16 Meg or 32 Meg chunks), it is not possible to buy a few more K of memory when a program needs it. By reducing the number of pages in the active working set of a program, larger problems can be solved without having to purchase additional memory. On larger multiprocessing systems, more processes can exist simultaneously in main memory.

3.6 Results

In Tables 3.1 to 3.4 we present the results for our virtual memory experiments (sorted in alphabetical order of allocator name). Tables 3.1 and 3.2 show the average number of 4K pages that a program would need so as to achieve 10%, 50%, and 90% CPU utilization running on a 100 MIPS machine with a 10ms disk, counting compulsory misses. The average number of pages necessary to achieve 10% CPU utilization (counting compulsory misses) ranges from 83.67 (Lea 2.6.1) to 141.33 (next fit LIFO with the wilderness preservation heuristic), a difference of 69%. For 50% utilization, the average number of pages needed ranges from 153.83 (Lea 2.6.1) to 272.17 (next fit LIFO split-7), a difference of 77%. For 90% utilization, the number of pages needed ranges from 246.170 (Lea 2.6.1) to 366.50 (next fit LIFO split-14), a difference of 49%. Thus, for a system that attempts to achieve 50% CPU utilization, the choice of a bad memory allocation policy can result in a program requiring 77% more memory than the choice of a good policy.

An important optimization is possible at the virtual memory level of the memory hierarchy: the first time a page is accessed, it is not necessary to load its contents from backing store. For security reasons, the operating system must fill all pages with zeros before any access is made to those pages, thus any data read from backing store for these pages will only be overwritten with zeros. With this optimization, we can ignore compulsory misses when measuring performance. Tables 3.3 and 3.4 show the number of 4K pages that a program would need to achieve 10%, 50%, and 90% CPU utilization running on a 100 MIPS machine with a 10ms disk, *not* counting compulsory misses.

Allocator	10% CPU	50% CPU	90% CPU
binary buddy	98.50	182.17	280.50
best fit AO	115.83	162.00	253.17
best fit AO 8K	115.00	163.00	254.17
best fit AO def AO	114.33	161.67	256.00
best fit AO def FIFO	119.50	165.00	256.50
best fit AO def LIFO	118.67	164.33	254.67
best fit AO no footer	111.17	154.33	247.00
best fit FIFO	116.50	162.83	254.67
best fit FIFO no footer	111.83	155.33	247.83
best fit LIFO	117.00	162.50	254.17
best fit LIFO def AO	114.67	162.67	257.33
best fit LIFO def FIFO	120.00	164.67	254.33
best fit LIFO def LIFO	120.33	164.67	256.33
best fit LIFO no footer	111.50	154.83	247.00
best fit LIFO split-14	114.83	163.00	254.67
best fit LIFO split-7	116.33	162.83	254.33
double buddy 10K	91.50	167.17	265.67
double buddy 5K	91.67	166.67	265.50
first fit AO	114.50	162.33	253.50
first fit AO 8K	114.33	163.00	254.17
first fit AO def AO	115.50	162.17	255.33
first fit AO def FIFO	119.00	164.83	255.83
first fit AO def LIFO	119.17	164.50	254.50
first fit AO no footer	110.33	155.17	247.67
first fit FIFO	123.33	167.00	258.50
first fit FIFO no footer	116.83	158.67	250.00
first fit LIFO	117.50	260.67	354.17

Table 3.1: Number of 4K pages necessary to achieve given percentage of CPU time, averaged across all traces, counting compulsory misses (Part 1)

Allocator	10% CPU	50% CPU	90% CPU
first fit LIFO def LIFO	124.67	189.17	281.50
first fit LIFO no footer	111.17	248.33	341.17
first fit LIFO split-14	118.50	259.83	354.00
first fit LIFO split-7	115.50	260.67	353.67
half fit	119.17	164.00	254.17
Lea 2.5.1	110.83	183.17	274.83
Lea 2.5.1 no footer	105.00	171.50	263.00
Lea 2.6.1	83.67	153.83	246.17
multi-fit max	127.33	164.83	255.00
multi-fit min	126.17	164.83	255.67
next fit AO	134.50	172.33	265.00
next fit AO 8K	137.50	176.17	267.67
next fit AO def AO	123.33	166.00	259.17
next fit AO def FIFO	130.83	170.67	262.00
next fit AO def LIFO	129.33	169.17	261.00
next fit AO no footer	128.67	166.33	258.00
next fit FIFO	126.67	188.33	278.67
next fit FIFO no footer	113.33	180.67	271.83
next fit LIFO	118.83	271.50	365.83
next fit LIFO def LIFO	118.00	199.83	291.17
next fit LIFO no footer	107.00	256.00	348.83
next fit LIFO split-14	120.00	271.83	366.50
next fit LIFO split-7	119.50	272.17	365.50
next fit LIFO WPH	141.33	266.67	359.83
simple segregated 2^N	105.83	207.67	317.00
simple segregated $3 * 2^N$	100.33	187.33	304.33

Table 3.2: Number of 4K pages necessary to achieve given percentage of CPU time, averaged across all traces, counting compulsory misses (Part 2)

Allocator	10% CPU	50% CPU	90% CPU
binary buddy	97.17	152.50	176.00
best fit AO	115.67	131.17	154.67
best fit AO 8K	114.50	130.33	154.17
best fit AO def AO	114.17	129.33	153.67
best fit AO def FIFO	119.17	131.67	154.00
best fit AO def LIFO	118.33	131.83	154.50
best fit AO no footer	110.67	125.50	147.33
best fit FIFO	116.00	131.67	153.67
best fit FIFO no footer	111.33	125.50	146.83
best fit LIFO	116.50	132.17	154.50
best fit LIFO def AO	114.33	128.83	153.33
best fit LIFO def FIFO	119.50	132.50	154.00
best fit LIFO def LIFO	119.83	132.50	154.33
best fit LIFO no footer	110.83	125.83	146.50
best fit LIFO split-14	114.33	130.17	154.33
best fit LIFO split-7	116.00	131.00	154.33
double buddy 10K	90.17	133.33	160.17
double buddy 5K	90.33	133.33	160.00
first fit AO	114.00	130.00	153.83
first fit AO 8K	114.00	130.50	155.00
first fit AO def AO	115.17	130.83	153.83
first fit AO def FIFO	118.33	132.67	154.83
first fit AO def LIFO	118.67	132.67	155.50
first fit AO no footer	110.00	124.83	146.83
first fit FIFO	123.17	137.33	156.33
first fit FIFO no footer	116.67	130.17	148.67
first fit LIFO	115.17	233.00	253.00

Table 3.3: Number of 4K pages necessary to achieve given percentage of CPU time, averaged across all traces, *not* counting compulsory misses (Part 1)

Allocator	10% CPU	50% CPU	90% CPU
first fit LIFO def LIFO	124.17	156.67	180.67
first fit LIFO no footer	108.67	219.83	240.83
first fit LIFO split-14	116.17	232.17	252.17
first fit LIFO split-7	113.33	232.33	253.83
half fit	118.67	131.17	154.67
Lea 2.5.1	109.33	146.33	173.50
Lea 2.5.1 no footer	103.83	137.00	164.67
Lea 2.6.1	82.83	121.00	147.67
multi-fit max	127.00	138.33	155.67
multi-fit min	125.83	137.00	155.50
next fit AO	134.00	147.83	163.17
next fit AO 8K	137.00	150.33	166.83
next fit AO def AO	123.00	138.33	155.33
next fit AO def FIFO	130.50	144.00	159.67
next fit AO def LIFO	129.00	141.83	158.50
next fit AO no footer	128.17	139.83	157.50
next fit FIFO	126.17	156.83	179.50
next fit FIFO no footer	112.83	150.17	172.33
next fit LIFO	112.50	243.17	263.83
next fit LIFO def LIFO	116.83	170.33	190.50
next fit LIFO no footer	102.83	228.67	248.17
next fit LIFO split-14	112.50	243.50	264.17
next fit LIFO split-7	112.17	244.17	264.17
next fit LIFO WPH	129.50	241.17	259.83
simple segregated 2^N	104.17	167.83	194.50
simple segregated $3 * 2^N$	99.00	148.00	178.67

Table 3.4: Number of 4K pages necessary to achieve given percentage of CPU time, averaged across all traces, *not* counting compulsory misses (Part 2)

Allocator	10% CPU	50% CPU	90% CPU
binary buddy	100.11%	113.51%	116.99%
best fit AO	105.25%	105.74%	104.23%
best fit AO 8K	104.89%	106.43%	104.74%
best fit AO def AO	108.46%	107.04%	107.61%
best fit AO def FIFO	116.81%	111.23%	107.46%
best fit AO def LIFO	115.73%	110.12%	105.14%
best fit AO no footer	99.52%	99.59%	100.86%
best fit FIFO	105.81%	106.39%	104.96%
best fit FIFO no footer	101.19%	100.31%	100.84%
best fit LIFO	106.25%	105.91%	104.57%
best fit LIFO def AO	109.08%	107.56%	108.25%
best fit LIFO def FIFO	117.84%	111.34%	105.68%
best fit LIFO def LIFO	116.92%	110.58%	107.40%
best fit LIFO no footer	100.00%	100.00%	100.00%
best fit LIFO split-14	104.88%	106.42%	104.39%
best fit LIFO split-7	106.58%	106.48%	104.56%
double buddy 10K	96.01%	108.84%	111.49%
double buddy 5K	96.12%	107.98%	111.05%
first fit AO	103.50%	105.89%	104.28%
first fit AO 8K	103.38%	106.88%	104.82%
first fit AO def AO	107.53%	107.60%	106.64%
first fit AO def FIFO	116.34%	111.47%	106.90%
first fit AO def LIFO	115.96%	110.87%	105.30%
first fit AO no footer	98.44%	100.58%	101.06%
first fit FIFO	113.76%	110.11%	107.19%
first fit FIFO no footer	106.83%	104.41%	102.39%
first fit LIFO	112.98%	127.50%	127.82%

Table 3.5: Number of 4K pages necessary to achieve given percentage of CPU time normalized to best fit LIFO no footer (geometric mean across all traces), counting compulsory misses (Part 1)

When ignoring compulsory misses, the average number of pages necessary to achieve 10% CPU utilization ranges from 82.83 (Lea 2.6.1) to 137.00 (next fit address-ordered free list with memory requested from the operating system in 8K units), a difference of 65%. For 50% CPU utilization, the average number of pages needed ranges from 121 (Lea 2.6.1) to 244.17 (next fit LIFO free-list with a splitting threshold of 7%), a difference of 102%. For 90% CPU utilization, the average number of pages needed ranges from 146.50 (best fit LIFO free-list no footer) to 264.17 (next fit LIFO free-list with a splitting threshold of 7%), a difference of 80%.

Because the numbers presented in Tables 3.1 to 3.4 are straight averages of the number of 4K pages necessary to achieve a given CPU utilization, these results are dominated by the programs with the largest footprints. A more interesting measure of locality is to normalize the locality for each allocator to some reference allocator, and then compute the geometric mean of this value. This computation will show us the relative performance of the different

Allocator	10% CPU	50% CPU	90% CPU
first fit LIFO def LIFO	126.41%	121.27%	118.55%
first fit LIFO no footer	106.95%	119.84%	120.71%
first fit LIFO split-14	114.09%	126.71%	127.48%
first fit LIFO split-7	110.62%	126.18%	125.72%
half fit	116.86%	110.14%	104.99%
Lea 2.5.1	113.32%	116.29%	112.85%
Lea 2.5.1 no footer	106.65%	109.47%	106.28%
Lea 2.6.1	85.95%	100.63%	100.13%
multi-fit max	126.82%	110.79%	105.53%
multi-fit min	127.01%	110.96%	106.57%
next fit AO	121.92%	110.55%	110.18%
next fit AO 8K	122.92%	111.43%	109.63%
next fit AO def AO	119.33%	110.90%	109.57%
next fit AO def FIFO	130.24%	115.08%	110.77%
next fit AO def LIFO	129.30%	114.35%	110.62%
next fit AO no footer	116.99%	105.65%	104.17%
next fit FIFO	118.99%	115.78%	111.02%
next fit FIFO no footer	108.77%	109.60%	107.04%
next fit LIFO	113.91%	128.66%	129.44%
next fit LIFO def LIFO	122.58%	122.91%	118.59%
next fit LIFO no footer	104.39%	122.38%	122.22%
next fit LIFO split-14	111.72%	127.41%	128.78%
next fit LIFO split-7	112.52%	128.40%	127.89%
next fit LIFO WPH	119.99%	125.14%	125.44%
simple segregated 2^N	112.11%	135.47%	143.98%
simple segregated $3 * 2^N$	107.17%	126.53%	137.85%

Table 3.6: Number of 4K pages necessary to achieve given percentage of CPU time normalized to best fit LIFO no footer (geometric mean across all traces), counting compulsory misses (Part 2)

Allocator	10% CPU	50% CPU	90% CPU
binary buddy	99.67%	116.11%	115.19%
best fit AO	105.82%	105.50%	106.42%
best fit AO 8K	105.07%	105.06%	105.87%
best fit AO def AO	109.04%	106.23%	105.82%
best fit AO def FIFO	117.08%	109.33%	106.30%
best fit AO def LIFO	116.07%	109.28%	106.88%
best fit AO no footer	99.65%	99.87%	100.54%
best fit FIFO	106.20%	105.76%	105.57%
best fit FIFO no footer	101.34%	100.35%	100.16%
best fit LIFO	106.44%	106.29%	106.09%
best fit LIFO def AO	109.55%	105.34%	105.60%
best fit LIFO def FIFO	118.06%	110.24%	106.40%
best fit LIFO def LIFO	117.14%	109.73%	106.88%
best fit LIFO no footer	100.00%	100.00%	100.00%
best fit LIFO split-14	105.06%	105.12%	106.09%
best fit LIFO split-7	106.85%	105.99%	105.89%
double buddy 10K	95.26%	107.77%	108.46%
double buddy 5K	95.61%	107.28%	108.21%
first fit AO	103.66%	104.21%	105.95%
first fit AO 8K	103.66%	104.89%	107.14%
first fit AO def AO	108.04%	106.75%	106.02%
first fit AO def FIFO	116.07%	110.83%	107.38%
first fit AO def LIFO	116.17%	110.10%	107.73%
first fit AO no footer	98.91%	99.18%	101.09%
first fit FIFO	114.21%	111.53%	107.55%
first fit FIFO no footer	107.11%	105.20%	101.85%
first fit LIFO	112.48%	135.13%	130.17%

Table 3.7: Number of 4K pages necessary to achieve given percentage of CPU time normalized to best fit LIFO no footer (geometric mean across all traces), *not* counting compulsory misses (Part 1)

Allocator	10% CPU	50% CPU	90% CPU
first fit LIFO def LIFO	127.00%	124.86%	119.54%
first fit LIFO no footer	106.16%	125.52%	123.34%
first fit LIFO split-14	113.59%	134.35%	129.47%
first fit LIFO split-7	110.17%	132.39%	129.57%
half fit	116.95%	108.83%	106.89%
Lea 2.5.1	113.00%	117.78%	113.73%
Lea 2.5.1 no footer	106.69%	110.55%	108.07%
Lea 2.6.1	85.69%	98.86%	100.76%
multi-fit max	127.27%	116.86%	108.27%
multi-fit min	127.44%	116.05%	107.94%
next fit AO	122.28%	117.72%	110.37%
next fit AO 8K	123.29%	118.00%	112.02%
next fit AO def AO	119.91%	115.73%	107.99%
next fit AO def FIFO	130.82%	121.57%	112.16%
next fit AO def LIFO	129.94%	120.12%	111.01%
next fit AO no footer	117.30%	110.69%	106.22%
next fit FIFO	119.51%	121.03%	115.55%
next fit FIFO no footer	109.22%	114.15%	109.31%
next fit LIFO	111.48%	136.32%	132.03%
next fit LIFO def LIFO	122.58%	129.51%	121.17%
next fit LIFO no footer	102.90%	130.27%	124.95%
next fit LIFO split-14	108.95%	134.62%	131.62%
next fit LIFO split-7	109.70%	135.79%	131.74%
next fit LIFO WPH	116.88%	134.60%	129.32%
simple segregated 2^N	110.97%	131.52%	131.22%
simple segregated $3 * 2^N$	106.58%	120.49%	123.69%

Table 3.8: Number of 4K pages necessary to achieve given percentage of CPU time normalized to best fit LIFO no footer (geometric mean across all traces), *not* counting compulsory misses (Part 2)

policies, equally weighting each program in the results. In Tables 3.5 to 3.8 we present our virtual memory locality results normalized to best fit LIFO no footer. Each value is the geometric mean of the performance across all six programs. The reference is 100%. Thus, a result of 110% is 10% worse than the reference allocator, and a result of 90% is 10% better than the reference allocator.

Using this measure of locality, the variation in policy performance for 10% CPU utilization counting compulsory misses ranges from 85.95% (14.05% better than best fit LIFO no footer) for Lea 2.6.1 to 130.24% (30.24% worse than best fit LIFO no footer) for next fit AO def FIFO, a difference of 51.53%. For 50% CPU utilization, the locality results range from 99.59% for best fit AO no footer to 135.47% for simple segregated storage 2^N, a difference of 36.03%. For 90% CPU utilization, the locality results range from 100.00% for best fit LIFO no footer to 143.98% for simple segregated storage 2^N, a difference of 43.98%.

When ignoring compulsory misses (Tables 3.7 and 3.8), for 10% CPU utilization, the locality results range from 85.69% for Lea 2.6.1 to 130.82% for next fit AO def FIFO, a difference of 52.67%. For 50% CPU utilization, the locality results range from 98.86% for Lea 2.6.1 to 136.32% for next fit LIFO, a difference of 37.89%. Finally, for 90% CPU utilization, the locality results range from 100.00% for best fit LIFO no footer to 132.03% for next fit LIFO, a difference of 32.03%.

These results show that the choice of a good allocator placement policy over a bad one can result in a difference of between 32% and 53% in the size of the program's active working set. Clearly, from these results, more attention needs to be paid to the locality costs of allocator placement choices at the virtual memory level of the memory hierarchy when designing systems.

In Tables 3.9 to 3.12 we present the results for our cache experiments. Tables 3.9 and 3.10 show the average miss rate for our six test programs when simulating 16K, 64K, and 256K 8-way set-associative caches. The miss rate for a 16K cache ranges from 0.899% (first fit address ordered no footer) to 1.532% (first fit FIFO), a difference of 70%. For a 64K cache, the miss rate ranges from 0.0205% (first fit address ordered no footer) to 0.0247% (simple segregated 2^N), a difference of 20%. The miss rate for a 256K cache ranges from 0.00862% (Lea 2.6.1) to 0.01748% (simple segregated 2^N), a difference of 103%. The two best policies in terms of fragmentation, best fit and first fit address ordered, fall within 15% of the best policy for a 16K cache. Deferred coalescing for these two policies appears to increase miss rate a small amount.

When comparing the results for fully-associative caches to those for 8-way set-associative caches, the most striking difference is binary buddy for a 16K cache. On average, this allocator has a 32% higher miss rate for an 8-way set-associative cache than for a fully-associative cache. An analysis of the data for each trace (Appendix B.2) shows that a large fraction of this difference is due to the performance of this policy on the Perl program. For this program, binary buddy had a miss rate of 1.114% for an 8-way set-associative 16K cache, and a miss rate of only 0.658% for a 16K fully-associative cache, a difference of 69%. The data for the Perl program shows that most allocators produced a high number of conflict misses, indicating that the program itself is problematic for small caches.

	8-way set-associative		
Allocator	16K	64K	256K
binary buddy	1.111%	0.0213%	0.01096%
best fit AO	1.015%	0.0212%	0.00938%
best fit AO 8K	1.009%	0.0211%	0.00917%
best fit AO def AO	1.053%	0.0213%	0.01001%
best fit AO def FIFO	1.275%	0.0215%	0.01312%
best fit AO def LIFO	1.064%	0.0214%	0.00963%
best fit AO no footer	0.955%	0.0205%	0.00914%
best fit FIFO	0.998%	0.0212%	0.00926%
best fit FIFO no footer	0.961%	0.0206%	0.00903%
best fit LIFO	0.995%	0.0212%	0.00934%
best fit LIFO def AO	1.062%	0.0213%	0.00982%
best fit LIFO def FIFO	1.255%	0.0214%	0.01098%
best fit LIFO def LIFO	1.068%	0.0214%	0.01009%
best fit LIFO no footer	0.951%	0.0206%	0.00872%
best fit LIFO split-14	0.993%	0.0212%	0.00939%
best fit LIFO split-7	0.979%	0.0212%	0.00949%
double buddy 10K	0.967%	0.0213%	0.00962%
double buddy 5K	0.947%	0.0212%	0.00970%
first fit AO	0.945%	0.0212%	0.00938%
first fit AO 8K	0.956%	0.0212%	0.00917%
first fit AO def AO	1.063%	0.0213%	0.00972%
first fit AO def FIFO	1.244%	0.0215%	0.01076%
first fit AO def LIFO	1.081%	0.0214%	0.00968%
first fit AO no footer	0.899%	0.0205%	0.00890%
first fit FIFO	1.532%	0.0219%	0.01089%
first fit FIFO no footer	1.415%	0.0209%	0.00964%
first fit LIFO	1.034%	0.0217%	0.01027%

Table 3.9: Cache miss rate, averaged across all traces, 8-way set-associative cache (Part 1)

	8-way set-associative		
Allocator	16K	64K	256K
first fit LIFO def LIFO	1.106%	0.0216%	0.01010%
first fit LIFO no footer	0.950%	0.0211%	0.00952%
first fit LIFO split-14	1.032%	0.0216%	0.01025%
first fit LIFO split-7	1.025%	0.0217%	0.01006%
half fit	1.092%	0.0214%	0.00961%
Lea 2.5.1	1.299%	0.0217%	0.01205%
Lea 2.5.1 no footer	1.177%	0.0210%	0.00919%
Lea 2.6.1	0.959%	0.0205%	0.00862%
multi-fit max	1.148%	0.0216%	0.00955%
multi-fit min	1.165%	0.0217%	0.00981%
next fit AO	1.376%	0.0220%	0.01009%
next fit AO 8K	1.483%	0.0226%	0.00983%
next fit AO def AO	1.196%	0.0217%	0.01000%
next fit AO def FIFO	1.431%	0.0219%	0.01131%
next fit AO def LIFO	1.217%	0.0218%	0.01007%
next fit AO no footer	1.337%	0.0215%	0.00901%
next fit FIFO	1.518%	0.0221%	0.00987%
next fit FIFO no footer	1.467%	0.0217%	0.00913%
next fit LIFO	1.014%	0.0218%	0.01015%
next fit LIFO def LIFO	1.139%	0.0216%	0.01058%
next fit LIFO no footer	0.912%	0.0215%	0.00978%
next fit LIFO split-14	1.031%	0.0222%	0.01059%
next fit LIFO split-7	1.008%	0.0218%	0.01012%
next fit LIFO WPH	0.998%	0.0219%	0.01012%
simple segregated 2^N	1.371%	0.0247%	0.01748%
simple segregated $3 * 2^N$	1.168%	0.0224%	0.01508%

Table 3.10: Cache miss rate, averaged across all traces, 8-way set-associative cache (Part 2)

	fully associative		
Allocator	16K	64K	256K
binary buddy	0.841%	0.0210%	0.01043%
best fit AO	0.890%	0.0211%	0.00912%
best fit AO 8K	0.892%	0.0211%	0.00899%
best fit AO def AO	0.927%	0.0212%	0.00974%
best fit AO def FIFO	1.134%	0.0214%	0.00996%
best fit AO def LIFO	0.959%	0.0213%	0.00919%
best fit AO no footer	0.841%	0.0205%	0.00924%
best fit FIFO	0.893%	0.0211%	0.00910%
best fit FIFO no footer	0.841%	0.0205%	0.00905%
best fit LIFO	0.898%	0.0211%	0.00907%
best fit LIFO def AO	0.932%	0.0213%	0.00958%
best fit LIFO def FIFO	1.129%	0.0214%	0.00940%
best fit LIFO def LIFO	0.955%	0.0213%	0.00974%
best fit LIFO no footer	0.842%	0.0205%	0.00870%
best fit LIFO split-14	0.892%	0.0211%	0.00913%
best fit LIFO split-7	0.889%	0.0211%	0.00900%
double buddy 10K	0.846%	0.0210%	0.00928%
double buddy 5K	0.848%	0.0210%	0.00925%
first fit AO	0.867%	0.0211%	0.00898%
first fit AO 8K	0.866%	0.0211%	0.00898%
first fit AO def AO	0.922%	0.0212%	0.00921%
first fit AO def FIFO	1.114%	0.0214%	0.00949%
first fit AO def LIFO	0.957%	0.0213%	0.00923%
first fit AO no footer	0.813%	0.0205%	0.00894%
first fit FIFO	1.339%	0.0217%	0.00954%
first fit FIFO no footer	1.233%	0.0211%	0.00928%
first fit LIFO	0.922%	0.0216%	0.00995%

Table 3.11: Cache miss rate, averaged across all traces, fully associative cache (Part 1)

	fully associative		
Allocator	16K	64K	256K
first fit LIFO def LIFO	0.991%	0.0216%	0.00995%
first fit LIFO no footer	0.863%	0.0211%	0.00960%
first fit LIFO split-14	0.921%	0.0215%	0.00989%
first fit LIFO split-7	0.923%	0.0216%	0.00956%
half fit	0.959%	0.0213%	0.00911%
Lea 2.5.1	1.083%	0.0215%	0.01102%
Lea 2.5.1 no footer	1.057%	0.0209%	0.00908%
Lea 2.6.1	0.828%	0.0205%	0.00861%
multi-fit max	0.984%	0.0215%	0.00922%
multi-fit min	0.971%	0.0215%	0.00927%
next fit AO	1.208%	0.0219%	0.00976%
next fit AO 8K	1.288%	0.0224%	0.00938%
next fit AO def AO	1.053%	0.0216%	0.00946%
next fit AO def FIFO	1.293%	0.0218%	0.01014%
next fit AO def LIFO	1.071%	0.0217%	0.00984%
next fit AO no footer	1.189%	0.0214%	0.00899%
next fit FIFO	1.302%	0.0220%	0.00932%
next fit FIFO no footer	1.276%	0.0216%	0.00907%
next fit LIFO	0.930%	0.0218%	0.01002%
next fit LIFO def LIFO	0.986%	0.0216%	0.01002%
next fit LIFO no footer	0.865%	0.0214%	0.00973%
next fit LIFO split-14	0.928%	0.0221%	0.01011%
next fit LIFO split-7	0.931%	0.0218%	0.01002%
next fit LIFO WPH	0.903%	0.0215%	0.00993%
simple segregated 2^N	1.070%	0.0240%	0.01621%
simple segregated $3 * 2^N$	0.978%	0.0221%	0.01305%

Table 3.12: Cache miss rate, averaged across all traces, fully associative cache (Part 2)

Allocator name	footer	no footer	% difference
best fit LIFO	106.44%	100.00%	6.44%
best fit FIFO	106.20%	101.34%	4.80%
best fit AO	105.82%	99.65%	6.19%
first fit LIFO	112.48%	106.16%	5.95%
first fit FIFO	114.21%	107.11%	6.63%
first fit AO	103.66%	98.91%	4.80%
next fit LIFO	111.48%	102.90%	8.31%
next fit FIFO	119.51%	109.22%	9.42%
next fit AO	122.28%	117.30%	4.25%

Table 3.13: Comparison of normalized locality for 10% CPU utilization, without compulsory misses.

Allocator name	footer	no footer	% difference
best fit LIFO	106.29%	100.00%	6.29%
best fit FIFO	105.76%	100.35%	5.39%
best fit AO	105.50%	99.87%	5.64%
first fit LIFO	135.13%	125.52%	7.66%
first fit FIFO	111.53%	105.20%	6.02%
first fit AO	104.21%	99.18%	5.07%
next fit LIFO	136.32%	130.27%	4.64%
next fit FIFO	121.03%	114.15%	6.03%
nest fit AO	117.72%	110.69%	6.35%

Table 3.14: Comparison of normalized locality for 50% CPU utilization, without compulsory misses.

3.7 Implementation Overheads

The results presented in Section 3.6 were for actual implementations of the policies under study. Unfortunately, for these experiments, we were not able to factor out all of the implementation costs from these results. In particular, some of our allocators placed headers and footers onto the blocks they allocated. These extra headers and footers are an implementation cost, and not a policy cost, and in this section, we account for this cost.

In Tables 3.13 to 3.15, we present the normalized performance of several of our allocator policies, both with and without footers for 10%, 50%, and 90% CPU utilization. From these tables, we can see that removing a 1 word footer on each object increases locality at the virtual memory level of the memory hierarchy by between 3.91% and 9.42% (an average of 5.87%).

3.8 Comparison of Fragmentation to Locality

We began this research with the question: is it ever a good idea to choose a policy that increases fragmentation in an attempt to improve locality? Clearly from our results, the choice of allocation policy affects locality, but the question of whether the best policies in terms of fragmentation also produce the best locality still remains. To answer this question,

Allocator name	footer	no footer	% difference
best fit LIFO	106.09%	100.00%	6.09%
best fit FIFO	105.57%	100.16%	5.40%
best fit AO	106.42%	100.54%	5.85%
first fit LIFO	130.17%	123.34%	5.54%
first fit FIFO	107.55%	101.85%	5.60%
first fit AO	105.95%	101.09%	4.81%
next fit LIFO	132.03%	124.95%	5.76%
next fit FIFO	115.55%	109.31%	5.71%
nest fit AO	110.37%	106.22%	3.91%

Table 3.15: Comparison of normalized locality for 90% CPU utilization, without compulsory misses.

we computed the correlation of the number of 4K heap pages used by each allocator[4] to the number of 4K pages needed to achieve 10%, 50%, and 90% CPU utilization for each program and each of the selected allocators (Section 2.5.4), not counting compulsory misses. We found that the correlations were 0.5403, 0.6033, and 0.5855 respectively. According to the standard interpretation of correlation, this means that 29.19%, 36.40%, and 34.28% of the difference in locality can be accounted for by differences in fragmentation. These results show that there is little correlation between fragmentation and locality for our selected allocators on our traces. The correlations across all allocators[5] are 0.5689, 0.6800, and 0.6663 respectively. This means that just 32.37%, 46.23%, and 44.40% of the difference in locality can be accounted for by differences in fragmentation.

We also computed the correlation between the number of heap pages used by each allocator to the cache miss rate three sizes of 8-way set-associative caches (32 byte line size). The correlations for our selected allocators for 16K, 64K, and 256K caches were -0.1302, 0.6393, and 0.8692 respectively. According to the standard interpretation of correlation, this means that 1.69%, 40.87%, and 74.68% of the difference in locality can be accounted for by differences in fragmentation. These results show that as caches become larger (and can hold more of the active working set of the programs), the correlations increase. The correlations across all allocators are -0.1766, 0.5802, and 0.8700. This means that 3.12%, 33.66%, and 75.69% of the differences in cache-level locality can be accounted for by differences in fragmentation.

In Chapter 2 we extensively studied the fragmentation caused by each of our 53 allocators linked with our eight programs. Fragmentation, however, is not always the most useful measure of allocator performance. In the presence of virtual memory, it is often more important to measure the amount of real memory needed to achieve good CPU utilization than to measure the total amount of memory used. In this sense, the number of pages of real memory needed to achieve good CPU utilization is a better measure of "fragmentation" than

[4]Because we measured the locality of the implementations of our allocators, we computed the correlation to the memory wasted for these implementations (Section 2.8.5) rather than to the actual fragmentation (Section 2.9).

[5]Recall that our complete set of allocators includes many more versions of the sequential-fit allocator policies than the other policies, and hence our results across all allocators are likely to be skewed towards these policies.

Allocator	90% CPU	Heap Size
binary buddy	280.50	304.29
best fit AO	253.17	190.83
best fit AO 8K	254.17	191.33
best fit AO def AO	256.00	189.83
best fit AO def FIFO	256.50	189.83
best fit AO def LIFO	254.67	189.83
best fit AO no footer	247.00	184.33
best fit FIFO	254.67	190.83
best fit FIFO no footer	247.83	184.17
best fit LIFO	254.17	191.33
best fit LIFO def AO	257.33	189.83
best fit LIFO def FIFO	254.33	189.83
best fit LIFO def LIFO	256.33	189.83
best fit LIFO no footer	247.00	184.17
best fit LIFO split-14	254.67	192.00
best fit LIFO split-7	254.33	191.83
double buddy 10K	265.67	243.94
double buddy 5K	265.50	243.98
first fit AO	253.50	191.00
first fit AO 8K	254.17	192.00
first fit AO def AO	255.33	189.83
first fit AO def FIFO	255.83	189.83
first fit AO def LIFO	254.50	189.83
first fit AO no footer	247.67	184.17
first fit FIFO	258.50	192.33
first fit FIFO no footer	250.00	185.83
first fit LIFO	354.17	287.17

Table 3.16: Number of 4K pages necessary to achieve given percentage of CPU time averaged across all traces, counting compulsory misses, compared to number of 4K pages used by the allocator implementation

Allocator	90% CPU	Heap Size
first fit LIFO def LIFO	281.50	215.17
first fit LIFO no footer	341.17	277.33
first fit LIFO split-14	354.00	288.17
first fit LIFO split-7	353.67	288.83
half fit	254.17	189.83
Lea 2.5.1	274.83	209.17
Lea 2.5.1 no footer	263.00	202.17
Lea 2.6.1	246.17	185.17
multi-fit max	255.00	190.67
multi-fit min	255.67	190.67
next fit AO	265.00	201.50
next fit AO 8K	267.67	200.67
next fit AO def AO	259.17	193.00
next fit AO def FIFO	262.00	195.50
next fit AO def LIFO	261.00	193.83
next fit AO no footer	258.00	193.67
next fit FIFO	278.67	215.00
next fit FIFO no footer	271.83	207.00
next fit LIFO	365.83	299.33
next fit LIFO def LIFO	291.17	225.17
next fit LIFO no footer	348.83	285.17
next fit LIFO split-14	366.50	300.17
next fit LIFO split-7	365.50	299.67
next fit LIFO WPH	359.83	295.83
simple segregated 2^N	317.00	249.83
simple segregated $3 * 2^N$	304.33	239.50

Table 3.17: Number of 4K pages necessary to achieve given percentage of CPU time averaged across all traces, counting compulsory misses, compared to number of 4K pages used by the allocator implementation

fragmentation itself. Tables 3.16 and 3.17 compare the number of 4K pages needed to achieve 90% CPU utilization (total text, global, stack, and heap pages) to the number of pages of heap memory used by each allocator. Here, we see that while some allocators with poor fragmentation results (the double-buddy allocators) achieve good CPU utilization, all of the allocators which performed within 5% of the best allocator in terms of CPU utilization also had very low fragmentation. Thus, we can conclude that it is *not* necessary to trade increased fragmentation to improve locality at the virtual memory level of the memory hierarchy.

3.9 A View of the Heap

In an attempt to better understand how allocator placement choices affect locality, we generated graphs, very similar to those presented in Section 2.14, of program memory access patterns over time. In the pictures that follow, the X-axis is time in instructions, and the Y-axis is the heap (going from low to high addresses). For any given pixel on the graph, the darkness represents how actively that portion of the heap at that point in time is being referenced. So, a black pixel represents a very heavily accessed area, and a white pixel represents an unreferenced area. A gray pixel is somewhere in between, depending on its darkness. For brevity, we only present the graphs for the Espresso and Grobner programs here. The entire set of these graphs can be seen in Appendix E.

The first nine pictures (Figures 3.5 to 3.13) are for the Espresso program. The most striking feature of these pictures is the very large grey areas in the simple segregated storage 2^N and simple segregated storage 2^N & $3 * 2^N$ allocators (Figures 3.12 and 3.13). These are areas of very poor locality of reference as compared to the relatively dark bands of the other figures.

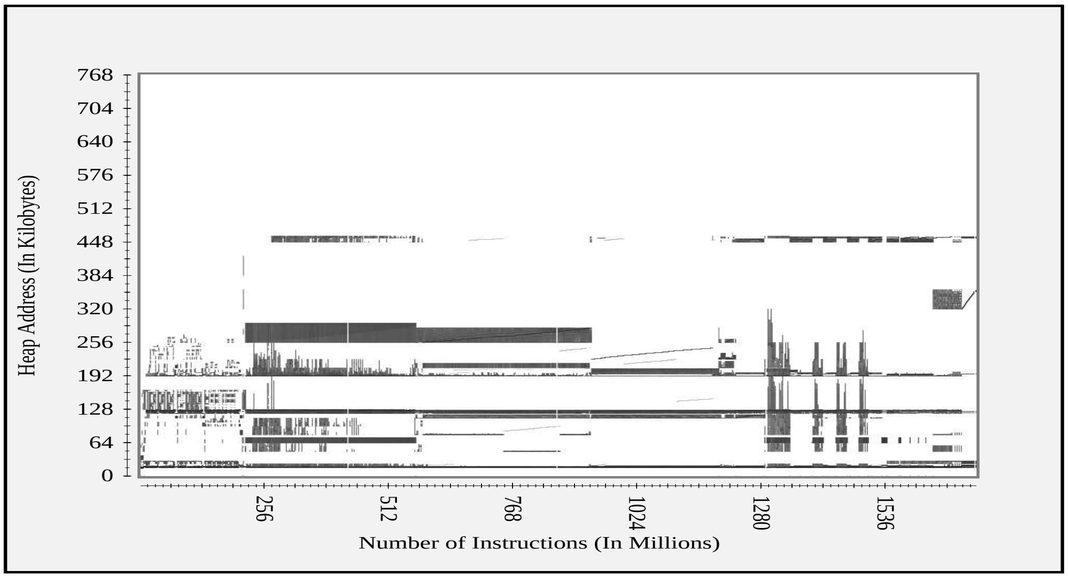

Figure 3.5: Memory access plot for Espresso using the binary-buddy allocator

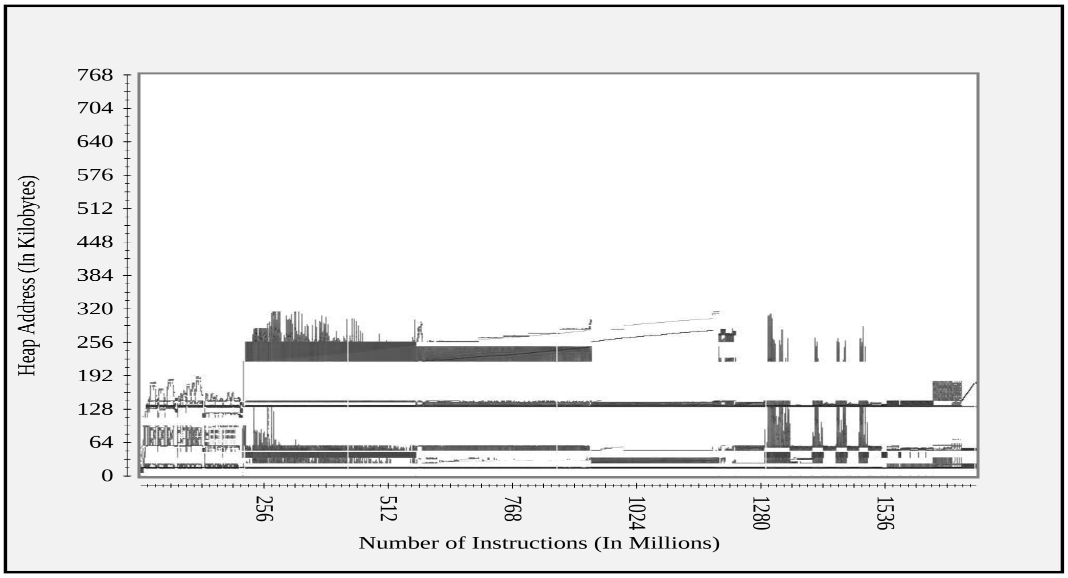

Figure 3.6: Memory access plot for Espresso using the best-fit LIFO no footer allocator

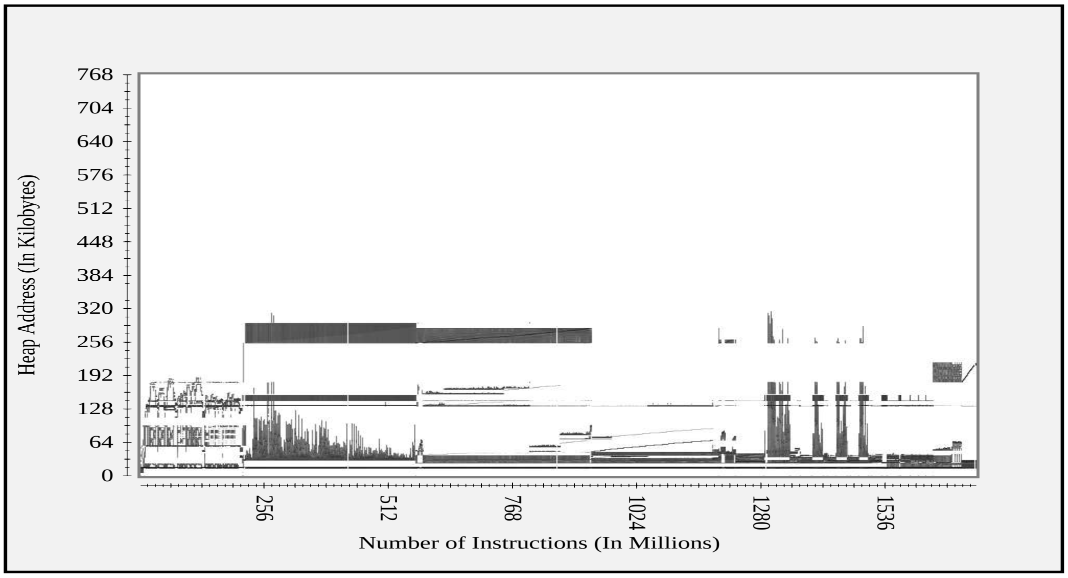

Figure 3.7: Memory access plot for Espresso using the first-fit address-ordered no footer allocator

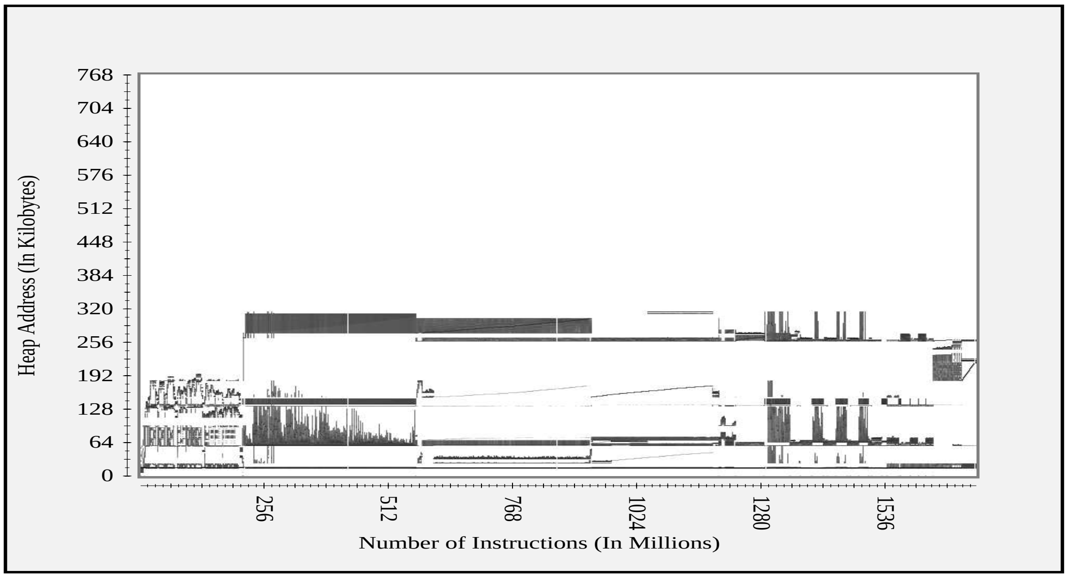

Figure 3.8: Memory access plot for Espresso using the first-fit LIFO no footer allocator

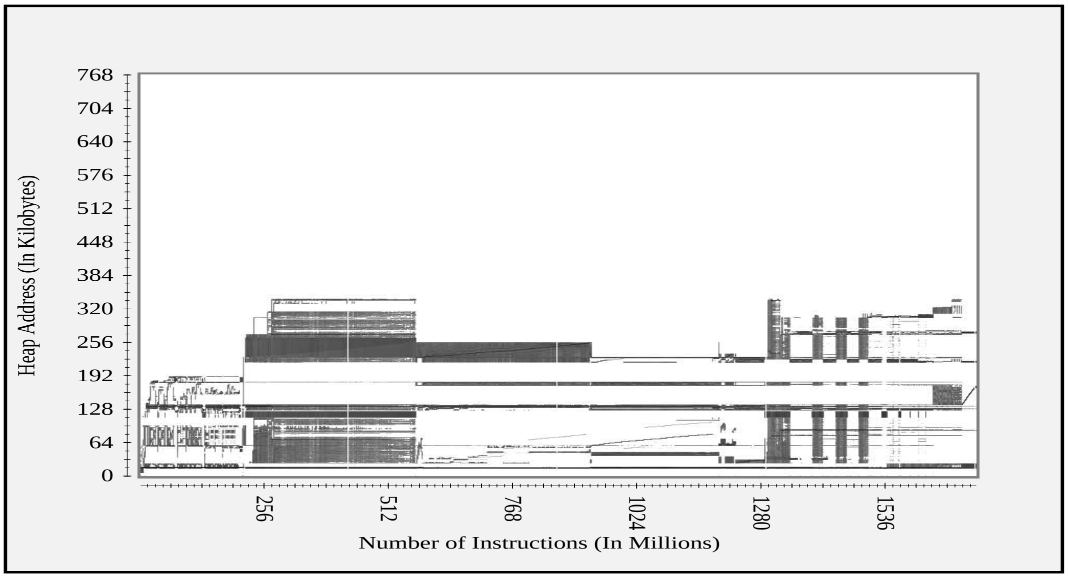

Figure 3.9: Memory access plot for Espresso using the half-fit allocator

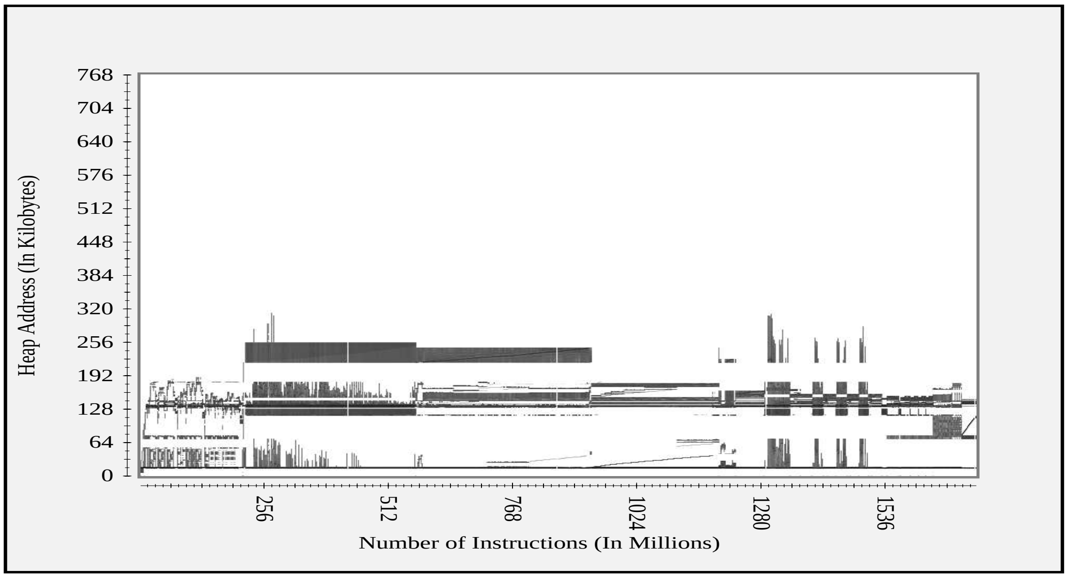

Figure 3.10: Memory access plot for Espresso using the Lea 2.6.1 allocator

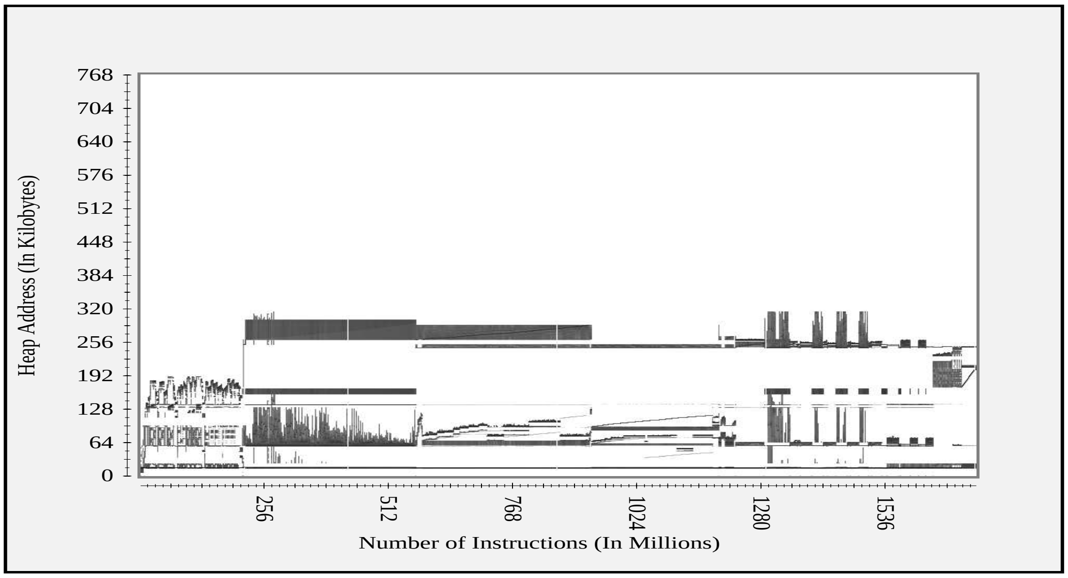

Figure 3.11: Memory access plot for Espresso using the next-fit LIFO no footer allocator

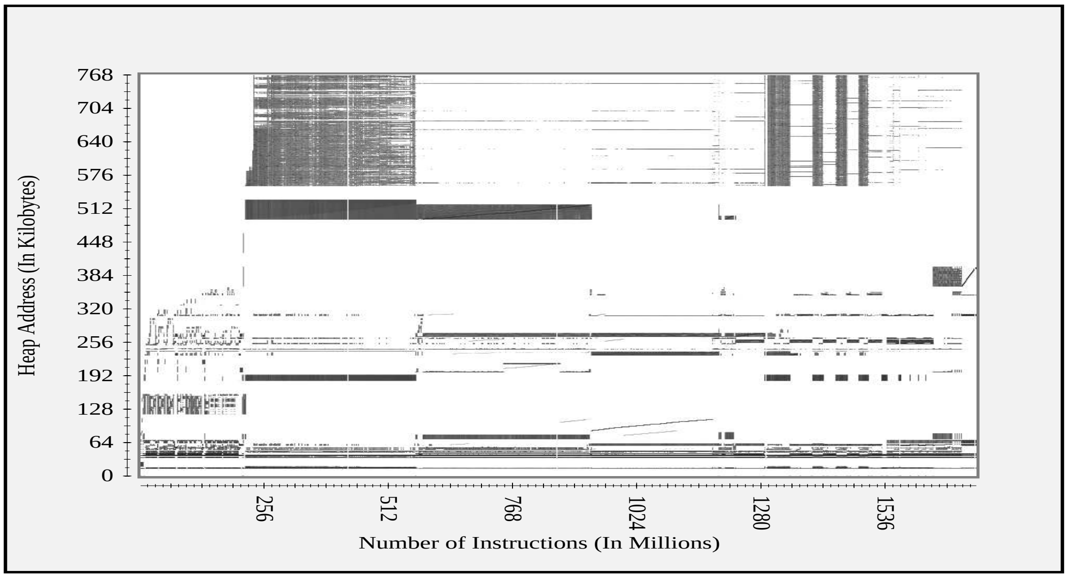

Figure 3.12: Memory access plot for Espresso using the simple segregated storage 2^N allocator

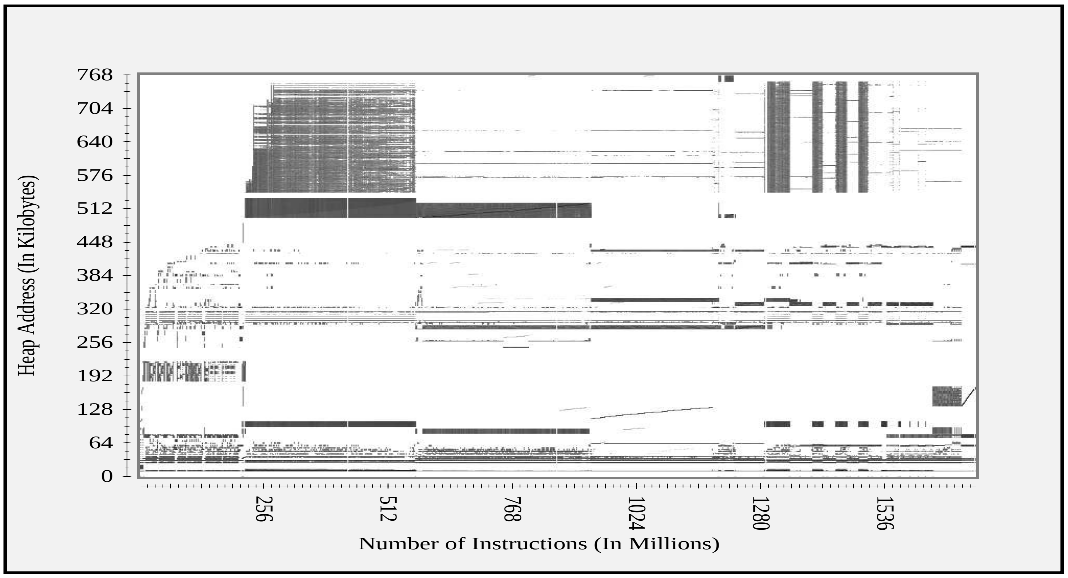

Figure 3.13: Memory access plot for Espresso using the simple segregated storage 2^N & $3*2^N$ allocator

The next nine pictures (Figures 3.14 to 3.22) are for the Grobner program. This program has two strong features. The first is the very large number of accesses to memory newly acquired from the operating system. This can be seen as the dark line bounding the top of the active area of the graphs. The allocators that produce better locality are those that tend to keep this area very finely focused (Figures 3.15, 3.16, and 3.19). The second feature is the repetition of triangular shapes for the entire run of the program. This feature represents a loop in the program's execution. In the Grobner program, one can see that the live memory of the program is cyclically touched. The heavy horizontal banding in the memory access patterns for the simple segregated storage allocators (Figures 3.21 and 3.22) illustrate the unusual allocation patterns of this particular policy.

Figure 3.14: Memory access plot for Grobner using the binary-buddy allocator

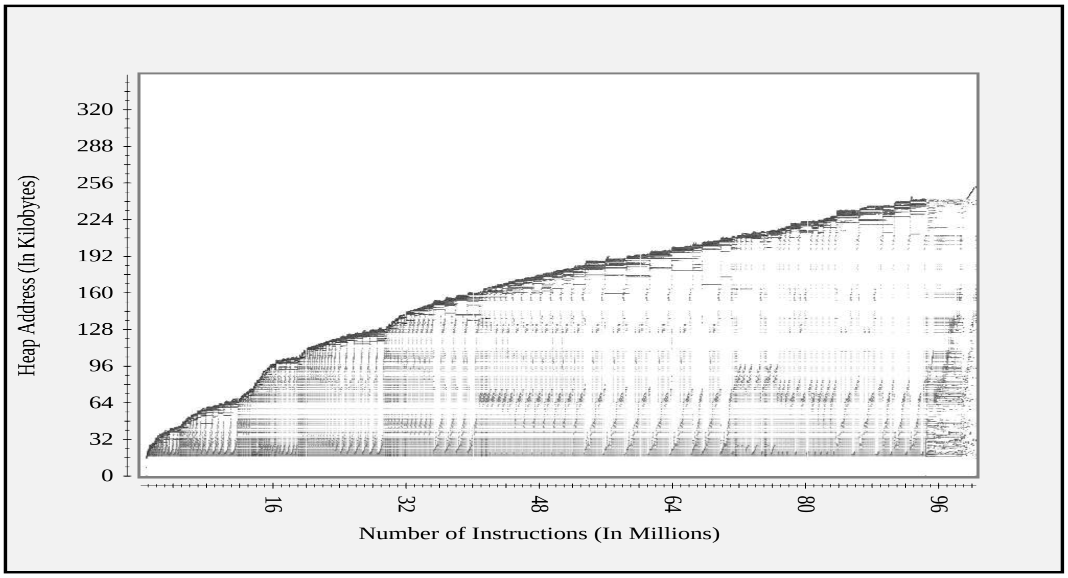

Figure 3.15: Memory access plot for Grobner using the best-fit LIFO no footer allocator

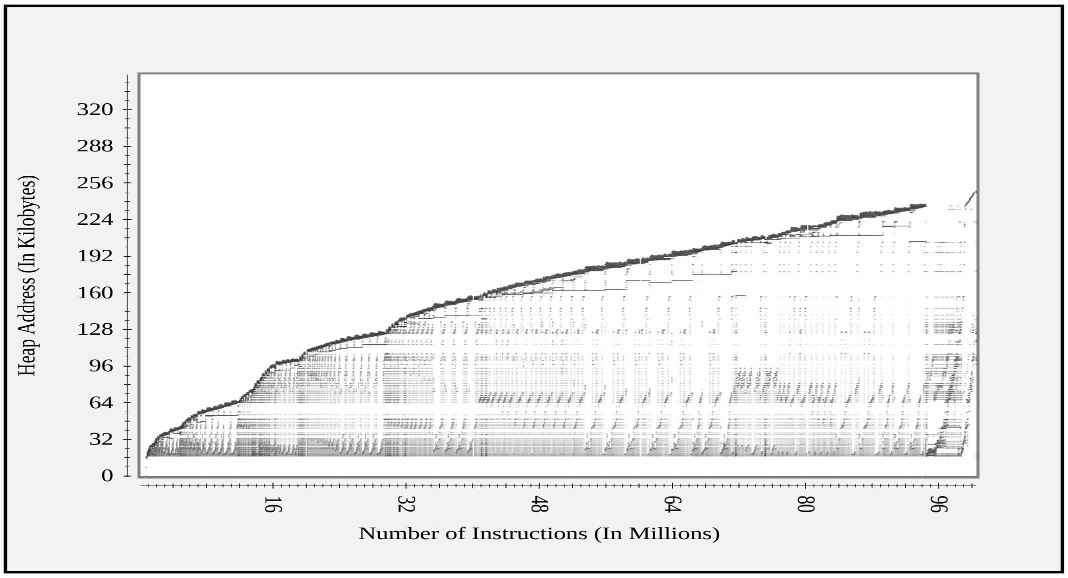

Figure 3.16: Memory access plot for Grobner using the first-fit address-ordered no footer allocator

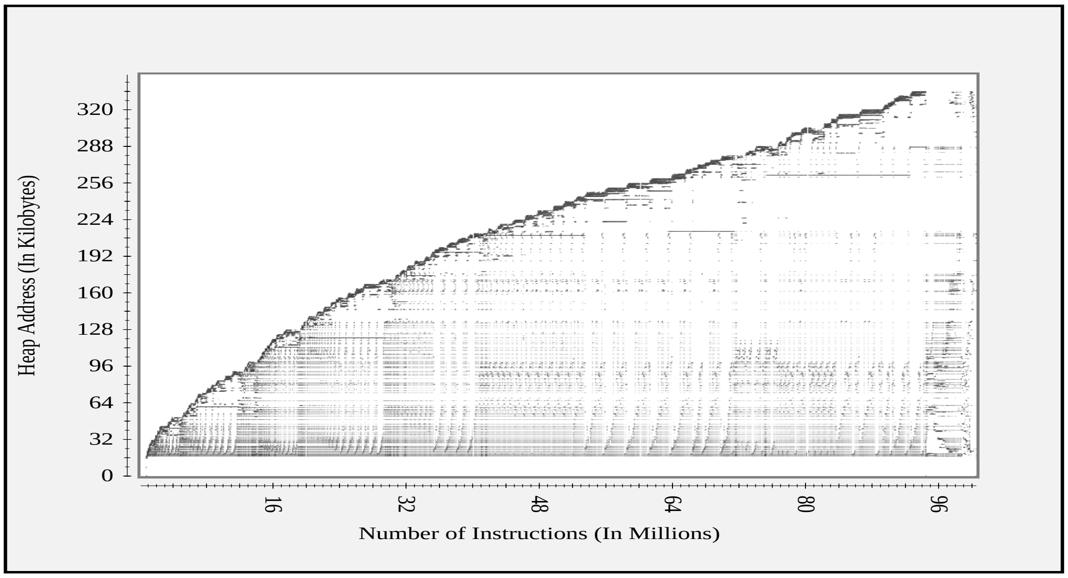

Figure 3.17: Memory access plot for Grobner using the first-fit LIFO no footer allocator

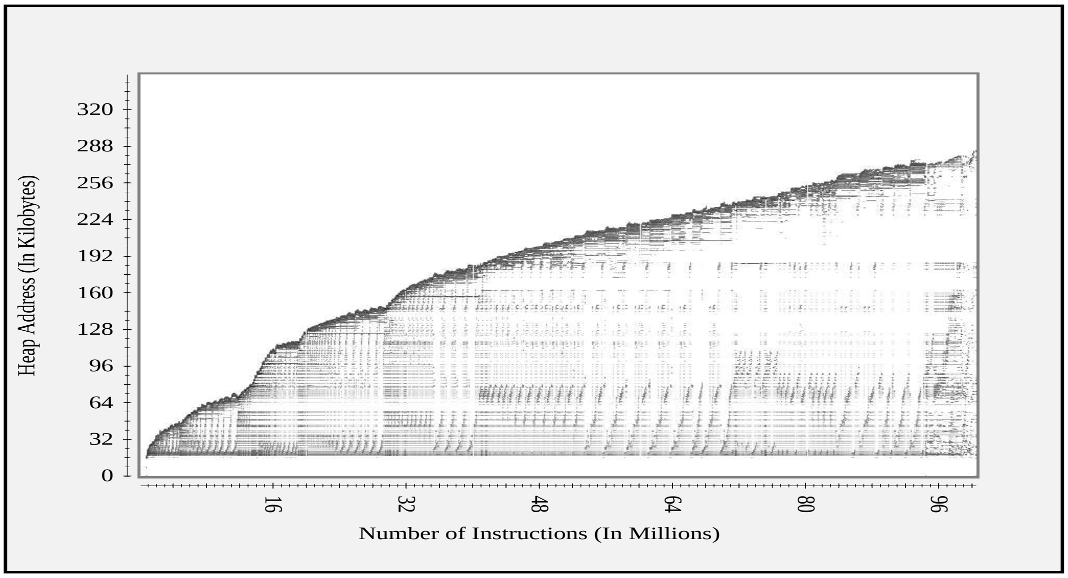

Figure 3.18: Memory access plot for Grobner using the half-fit allocator

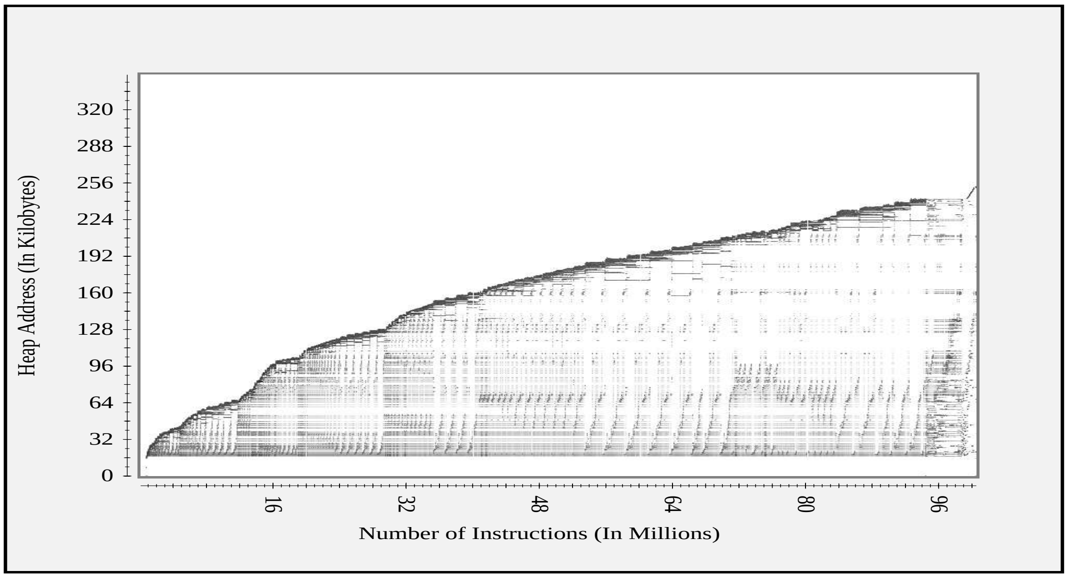

Figure 3.19: Memory access plot for Grobner using the Lea 2.6.1 allocator

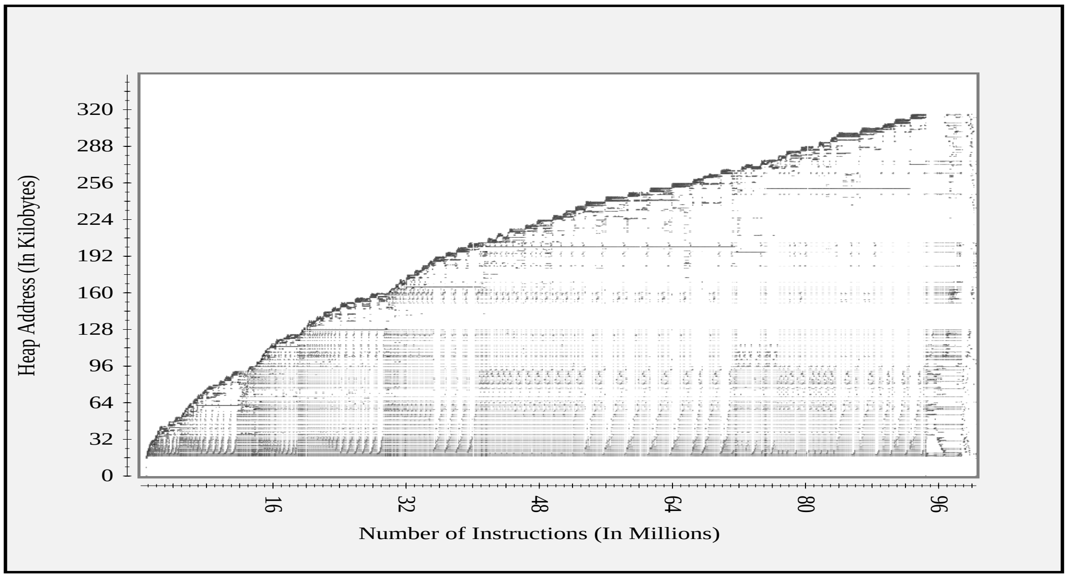

Figure 3.20: Memory access plot for Grobner using the next-fit LIFO no footer allocator

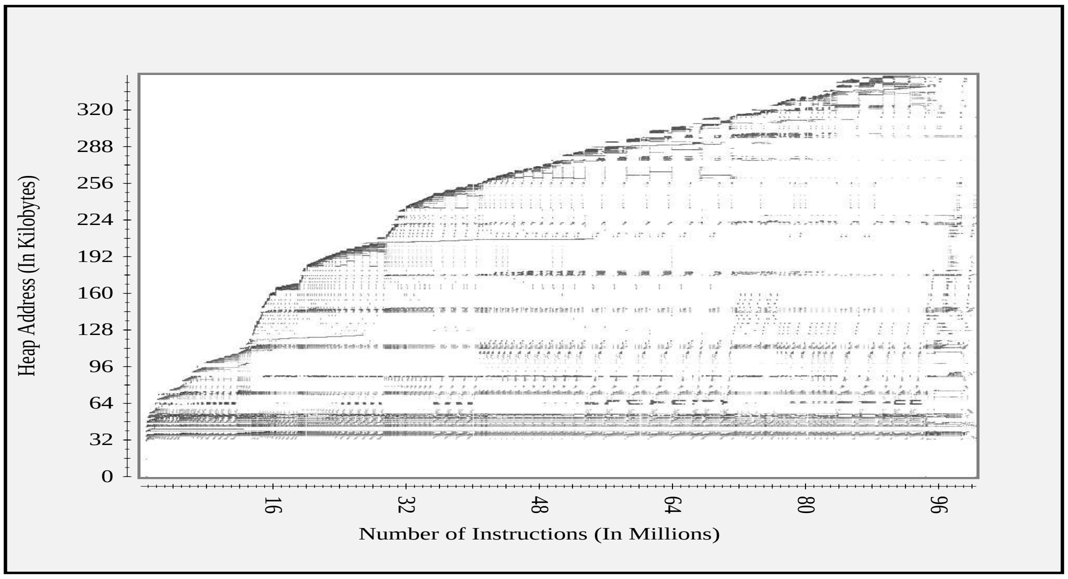

Figure 3.21: Memory access plot for Grobner using the simple segregated storage 2^N allocator

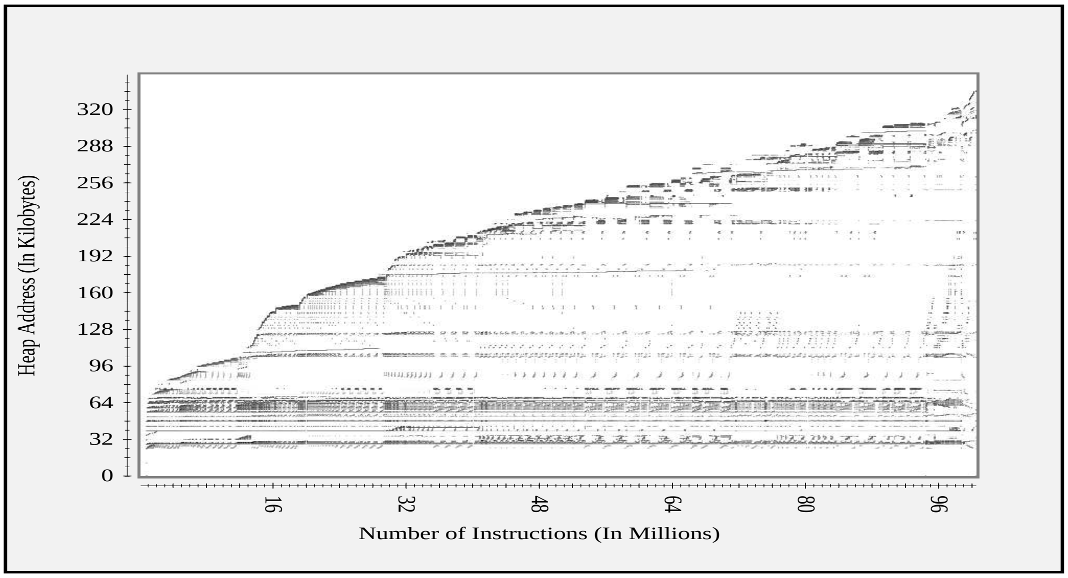

Figure 3.22: Memory access plot for Grobner using the simple segregated storage 2^N & $3*2^N$ allocator

3.10 Summary

In Section 3.6, we showed that allocator placement choice can have a large effect on locality of reference at both the cache and virtual memory level. The best policies in terms of locality performed between 65% and 102% better than the worst at the virtual memory level, and between 20% and 103% better than the worst at the cache level. We also showed that there is some correlation, although not strong, between fragmentation and locality for larger caches and memories, and very little correlation between fragmentation and locality for small caches and memories.

Our most significant result is that the best policies in terms of fragmentation (best fit and first fit address ordered) also were within 10% of the best policies in terms of locality, at both the virtual memory and cache levels of the memory hierarchy, for all but the smallest memory or cache sizes, and within 15% of the best policy for the smallest cache size.

Chapter 4

Real-Time Garbage Collection

There are many similarities between traditional memory allocation algorithms, such as those used to implement `malloc()` and `free()` in the C programming language, and the algorithms used for non-copying garbage collection. In particular, both can use exactly the same policies in deciding which block of memory should be used to satisfy a request for more storage. In addition, these algorithms can be used for real-time garbage collection if a particular policy can be implemented to guarantee that memory can be allocated within a reasonable time-bound.

We have implemented a hard real-time garbage collector using a technique called *non-copying implicit reclamation*, or "fake copying." This technique gives our collector many of the advantages of a copying collector without some of the associated costs. This collector currently works with the locally developed "RScheme" Scheme compiler,[1] the Tower Eiffel compiler developed by Tower Technology Corporation, and the gnu C++ compiler.[2] We have found, through our survey of the real-time garbage collection literature, that there is a pervasive misconception as to what is required for a garbage collector to be called real-time. We will therefore begin with a discussion of the requirements for real-time garbage collection. We will then develop a theoretical framework for the discussion and comparison of incremental garbage collection techniques. After laying this groundwork, we will describe our garbage collector implementation and show how it meets these criteria. Next, we will present preliminary measurements for the performance of our collector. Finally, we will discuss incremental generational garbage collection and its applicability to real-time programming.

4.1 Real-Time Collection: What It Is and When It Is Not

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 and performing large amounts of work without interruption. Strict bounds on individual garbage collection pauses are often used as the only criterion for real-time garbage

[1]Information on RScheme is available from the RScheme web site `http://www.rscheme.org`.

[2]We use the smart pointer idiom [Ss92] to provide the necessary support for garbage collection.

collection, but for practical applications, the requirements are often even stricter.

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.

Finally, because of the critical nature of most real-time applications, a third requirement for real-time garbage collection is to *guarantee space bounds*. This issue is much more complicated for garbage collected systems than for traditional systems, because the application programmer no longer has direct control over when a block of memory becomes available for reuse.

4.2 Incremental Copying Garbage Collectors

In this section, we present related work on incremental copying real-time garbage collection. For a more extensive survey of this and other work on garbage collection, see [Wil].

4.2.1 Baker's Incremental Copying Technique

Baker's incremental copying technique [Bak78] is the best-known "real-time" collection strategy, but it is actually poorly suited to real-time garbage collection on stock hardware: its close coupling between application program actions and collector actions makes it intrinsically more expensive and difficult to use for real-time applications. Even though any given pause caused by the collector is short and strictly bounded, these pauses may be clustered closely together, causing the application to miss its larger granularity deadlines.

Baker uses a *read-barrier* (special code potentially executed at every pointer reference) to maintain consistency between the running program and the garbage collector. Thus, every reference to a pointer potentially causes an increment of garbage collection to be performed. The unfortunate act of traversing a list that has not yet been reached by the collector can cause all of the objects in that list to be copied. While copying each object takes a strictly bounded amount of time, copying the entire list can keep the application from getting a reasonable fraction of the CPU time, making the application miss its real-time deadlines. This read-barrier cost is potentially high, and very unpredictable, because the cost of traversing an ordinary list is strongly dependent on whether or not the list has already been reached and copied by the collector [Nil88, EV91, Wit91]. In addition, Baker's algorithm will systematically have unpredictable performance at the beginning of a garbage collection cycle, because referencing *any* object will trigger an increment of copying; the traversal of a large data structure immediately after the beginning of a garbage collection cycle will cause *all* of the referenced objects to be copied. It is, in general, very difficult to predict when a garbage collection cycle will begin. Even extensive testing of the system is not guaranteed to reveal all interactions between the garbage collector and the running application. In short, Baker's scheme will unpredictably suffer from unacceptably large amounts of garbage collection work, possibly during critical application operations.

This problem is even worse in recent collectors which use page-wise virtual memory protection to trigger larger increments of collector work [AEL88, Det90a, Joh92, BDS91], and is also significant on Lisp-machine style hardware. Even if the necessary checks are performed by dedicated parallel hardware, most of the available CPU time may be used up (in the worst case) by trapping to copying routines and performing the copying operations.

4.2.2 Nilsen's Hardware Assisted Technique

Nilsen and Schmidt [NS90] argue that even if increments of garbage collection work are small, a real-time program may miss its deadlines if too many small increments add up to too much total overhead over some period of time relevant to a deadline. For example, a collector might impose an overhead of 30 instructions per pointer dereference in the worst case. If the program attempts to execute a very large number of pointer dereferences over a period of time relative to a deadline, it may spend the majority of its time doing garbage collection work, and run so slowly that it misses the deadline.

Nilsen's proposed solution to this problem is to build special hardware that guarantees that the worst-case delay for any individual program operation is small relative to that operation's normal execution time [NS90]. Unfortunately, he gives no indication of what that worst-case delay is for his hardware, except to admit that the worst-case for a pointer dereference is 2 microseconds.[3] On a 100 MIPS machine, this would be a slowdown of 200 times. In addition, his scheme requires that the cache be flushed at the end of every garbage collection cycle, causing a further unpredictable loss of performance. The basic problem with his approach is that he is trying to use special hardware to speed up Baker's incremental copying algorithm, and that algorithm has fundamental difficulties guaranteeing that the application will receive enough CPU cycles to meet all of its real-time deadlines.

Even if we could dedicate enough hardware to keep the worst-case bounds to a reasonable level, we believe that this is unnecessarily restrictive. To meet a real-time deadline, it is only necessary that the garbage collector not use up too large a fraction of the available CPU cycles *at a time-scale relevant to the program's deadlines.* Our strategy is therefore to allow the collector to incur dozens of instructions of overhead on some pointer operations, so long as we can guarantee that this will not happen too often. (Naturally, "too often" must be quantified relative to the application's responsiveness requirements.) Such a guarantee requires a weaker coupling between program and collector operations than is generally provided by copying collectors.

The above algorithms all fall into the class of incremental copying *read-barrier* techniques. Others have proposed copying-based algorithms that rely on a combination of a read-barrier and a *write-barrier* (extra instructions executed at every pointer store) [Bro84], or on a write-barrier only [NOPH92] to coordinate the collector's view of the graph with that of the application.

[3]Nilsen gives no indication of the parameters used in timing this worst-case, so it is impossible to evaluate this pause.

4.2.3 Brooks' Technique

Brooks' algorithm [Bro84] deserves special mention as the only copying hard real-time garbage collection algorithm that we know of. This algorithm combines a read-barrier and a write-barrier to coordinate the work of the mutator and the garbage collector in a way that is easier to make real-time than Baker's algorithm. Like Baker's algorithm, this collector incrementally copies objects from one area of memory to another. Unlike Baker's algorithm, however, referencing an uncopied object does not force an increment of collection work to be performed. The application always sees the correct version of all objects on the heap by using an unconditional indirection for all heap references. If an object has been copied, then this indirection points to the new version of the object. If the object has not yet been copied, then the indirection points to the old version of the object. Finally, a write-barrier is used to copy any non-forwarded objects to the new space before any changes to that object are made. Thus, Brooks' algorithm potentially performs garbage collection work on every *write* rather than every *read*, as is the case with Baker's algorithm, making the collection work somewhat more predictable. Like Baker's algorithm, the copying nature of this algorithm nicely controls fragmentation problems that can occur with non-copying algorithms.

Brooks' algorithm is often described as an improvement on Baker's copying algorithm. However, the read/write-barrier strategy makes it more similar to Baker's non-copying algorithm than Baker's copying algorithm.

4.2.4 A Novel Extension of Brooks' Technique

In Brooks' technique, a write-barrier is used to copy objects before any modifications to these objects are made; this maintains the strong tri-color invariant, which we will discuss in Section 4.4.1. This technique can easily be generalized by recognizing that the purpose of the write-barrier is to maintain the tri-color invariants. Thus, Brooks' write-barrier can be substituted with any write-barrier that maintains either the strong or weak tri-color invariants, such as those we use in our collector. Implementation and study of this modification to Brooks' algorithm is future work.

4.2.5 Magnusson and Henriksson's Scheduling Techniques

Magnusson and Henriksson address scheduling considerations for hard real-time garbage collectors [MH95]. While their approach is described as an extension of a Brooks-style copying garbage collector, we believe that it would work equally well with a non-copying garbage collector such as ours. They suggest that the scheduler be modified to accommodate three priority levels:

1. High-priority processes
2. Garbage collection
3. Low-priority processes

The scheduler primarily assigns processor time to the high-priority processes. The remaining time is divided between the garbage collector and the low-priority processes. In order for the low-priority processes not to suffer from starvation, the garbage collector will suspend its work as soon as it can guarantee that the high-priority processes will not run out of memory.

In order to handle the copying requirements of Brooks' algorithm, Magnusson and Henriksson postpone the actual copying of objects while high-priority processes are running. For these processes, they modify the write-barrier to only reserve space in to-space. Objects which need to be forwarded are copied after the high-priority processes complete. Finally, the write-barrier sets the forwarding pointer in the reserved area to point back to the actual object in from-space, so that modifications to objects which have yet to be copied are made to the actual object, and not to the reserved space.

4.2.6 Copying vs. Non-Copying Techniques

There are a number of important considerations when choosing between copying and non-copying real-time garbage collection strategies. Among these are space costs, barrier costs, and available compiler and language support.

Space costs are often the most important consideration when choosing a strategy. Copying garbage collectors require that enough extra space always be available to copy all live objects during garbage collection. Non-copying garbage collectors do not require this extra space. However, non-copying garbage collectors are subject to fragmentation. Based on our results in Chapter 2,[4] we believe that in the usual case this fragmentation is very low, and will generally be much lower than the extra space required by a copying collector. On the other hand, for hard real-time garbage collectors it is the worst-case fragmentation that matters, not the usual-case fragmentation; a copying algorithm will generally require less space than a non-copying algorithm with worst-case fragmentation.

Barrier costs are another important consideration when choosing a strategy. Brooks' write-barrier is essentially the same write-barrier we use in our non-copying collector, except that our collector does not incur the cost of actually copying objects. Furthermore, unlike our algorithm, Brooks' algorithm requires an additional read-barrier, thus making our algorithm strictly faster than his.

Available compiler and language support is the third important consideration when choosing a garbage collection strategy. Copying algorithms require more support than non-copying algorithms. Because a copying collector moves objects during collection, *all* pointers to objects must be identified and updated when an object is copied. A non-copying collector, on the other hand, need only identify *one* pointer to an object to keep that object live.

Finally, a generational collector can be a hybrid of these two methods, with younger generations using a fast non-copying algorithm, and older generations using a space-efficient copying algorithm.

[4]Anecdotal evidence from users of both Boehm's free collector and Geodesic System's commercial collector suggest that usual case fragmentation is low for non-copying garbage collectors.

4.3 Coherence and Conservatism

Incremental garbage collectors must take into account changes to the reachability graph made by the mutator during the collector's traversal. Incremental *copying* collectors pose more severe coordination problems—the mutator must also be protected from changes made by the garbage collector.

It may be enlightening to view these issues as a variety of *coherence* problems: having multiple processes attempt to share changing data, while maintaining some kind of consistent view [NOPH92]. (Readers unfamiliar with coherence problems in parallel systems should not worry too much about this terminology; the issues should become apparent as we go along.)

An incremental mark-sweep traversal poses a *multiple readers, single writer* coherence problem—the collector's traversal must respond to changes, but only the mutator can change the graph of objects. (Similarly, only the traversal can change the mark bits; each process can update values, but any field is writable by only one process. Only the mutator writes to pointer fields, and only the collector writes to mark fields.)

Copying collectors pose a more difficult problem—a *multiple readers, multiple writers* problem. Both the mutator and the collector may modify pointer fields, and each must be protected from inconsistencies introduced by the other.

Garbage collectors can efficiently solve these problems by taking advantage of the semantics of garbage collection, and using forms of *relaxed consistency*—that is, the processes need not always have a consistent view of the data structures, as long as the differences between their views do not matter to the correctness of the algorithm.

In particular, the garbage collector's view of the reachability graph is typically *not* identical to the actual reachability graph visible to the mutator. It is only a safe, *conservative* approximation of the true reachability graph—the garbage collector may view some unreachable objects as reachable, as long as it does not view reachable objects as unreachable, and erroneously reclaim their space. Typically, some garbage objects go unreclaimed for a while; usually, these are objects that become garbage after being reached by the collector's traversal. This so called *floating garbage* is reclaimed at the end of the next garbage collection cycle, since it will be garbage at the *beginning* of that collection, and the tracing process will not conservatively view it as live. The inability to reclaim floating garbage immediately is unfortunate, but may be essential to avoid very expensive coordination between the mutator and collector.

The kind of relaxed consistency used, and the corresponding coherence features of the collection scheme, are closely intertwined with the notion of conservatism. In general, the more we relax the consistency between the mutator's and the collector's views of the reachability graph, the more conservative our collection becomes, and the more floating garbage we must accept. On the positive side, the more relaxed our notion of consistency, the more flexibility we have in the details of the traversal algorithm.[5]

[5]In parallel and distributed garbage collection, a relaxed consistency model also allows more parallelism and/or less synchronization, but that is beyond the scope of this dissertation.

4.4 Tri-Color Marking

The abstraction of *tri-color marking* is helpful in understanding coherence and conservatism in incremental garbage collection [DLM+78]. Garbage collection algorithms can be conceptually described as a process of traversing the graph of reachable objects and coloring them. The objects are originally colored white, and as the graph is traversed, they are colored black[6]. When there are no reachable objects left to blacken, the traversal of live data structures is finished. Those objects that will be retained are colored black, and any remaining white objects are known to be garbage and can be reclaimed. Once the garbage objects are reclaimed, the live objects are reverted from black to white and the process repeats.

In a simple mark-sweep collector, this coloring is directly implemented by setting mark bits: objects whose bit is set are black. In a copy collector, this coloring is the process of copying objects from one area of memory called *from-space*, to another area of memory called *to-space*—unreached objects in from-space are considered white, and objects copied to to-space are considered black. The abstraction of coloring is orthogonal to the distinction between marking and copying collectors, but is important for understanding the basic differences between incremental collectors.

In incremental collectors, the intermediate states of the coloring traversal are also important, because of ongoing mutator activity: the mutator cannot be allowed to change things "behind the collector's back" in such a way that the collector will fail to find all reachable objects.

To understand and prevent such interactions between the mutator and the collector, it is useful to introduce a third color, gray, to signify that an object has been reached by the traversal, but that *its descendants may not have been.* That is, as the traversal proceeds outward from the roots, objects are initially colored gray. When they are scanned and pointers to their offspring are traversed, they are blackened and the offspring are colored gray.

In summary, the significance of these three colors is:

- White objects are those that have not yet been reached by the collector's tracing traversal. If at the end of collection an object is still marked white, then it is known to be garbage.

- Gray objects are those currently under consideration by the collector, because they are known to be reachable, but it is not yet known what other objects are reachable from them. In implementation terms, this just means that the objects are in the stack (or queue) that controls the collector's traversal of reachable data.

- Black objects are those that the collector has finished considering—they have already been examined and their role in the reachability graph is known. In terms of implementation, this means that they have been removed from the traversal stack (or queue).

As shown in Figure 4.1, the traversal proceeds in a wavefront of gray objects, which separates the white (unreached) objects from the black objects that have been passed by the

[6] The colors reflect the garbage collector's state of knowledge about the objects.

wave—that is, there are no pointers directly from black objects to white ones. This abstracts away from the particulars of the traversal algorithm—it may be depth-first, breadth-first, or just about any kind of exhaustive traversal. It is only important that a well-defined gray fringe be identifiable, and that the mutator preserve the invariant that no black object hold a pointer directly to a white object.

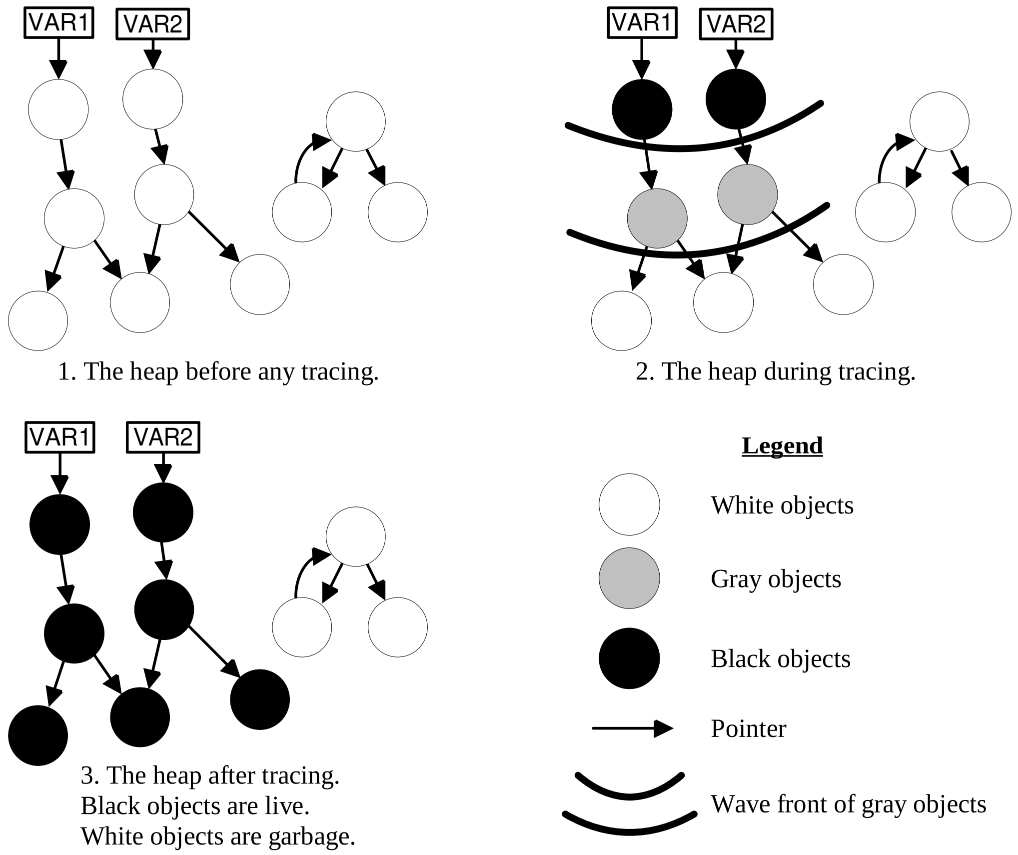

Figure 4.1: Example of tri-color marking

The importance of this invariant[7] is that the collector must be able to assume that it is "finished with" black objects, and can continue to traverse gray objects and move the wavefront forward. If the mutator creates a pointer from a black object to a white one, it must somehow notify the collector that this assumption has been violated. This ensures that the collector's bookkeeping is brought up to date.

4.4.1 The Tri-Color Invariants

The main problem of incremental collection is to ensure that the collector's notion of the reachability graph is always synchronized with the actual reachability graph, regardless of changes made by the running program (see Figure 4.2). If the program creates a pointer from a black object to a white object, and nothing special is done, the pointer will not be found

[7]We call this the *strong* tri-color invariant (see Section 4.4.1). Other invariants are also possible.

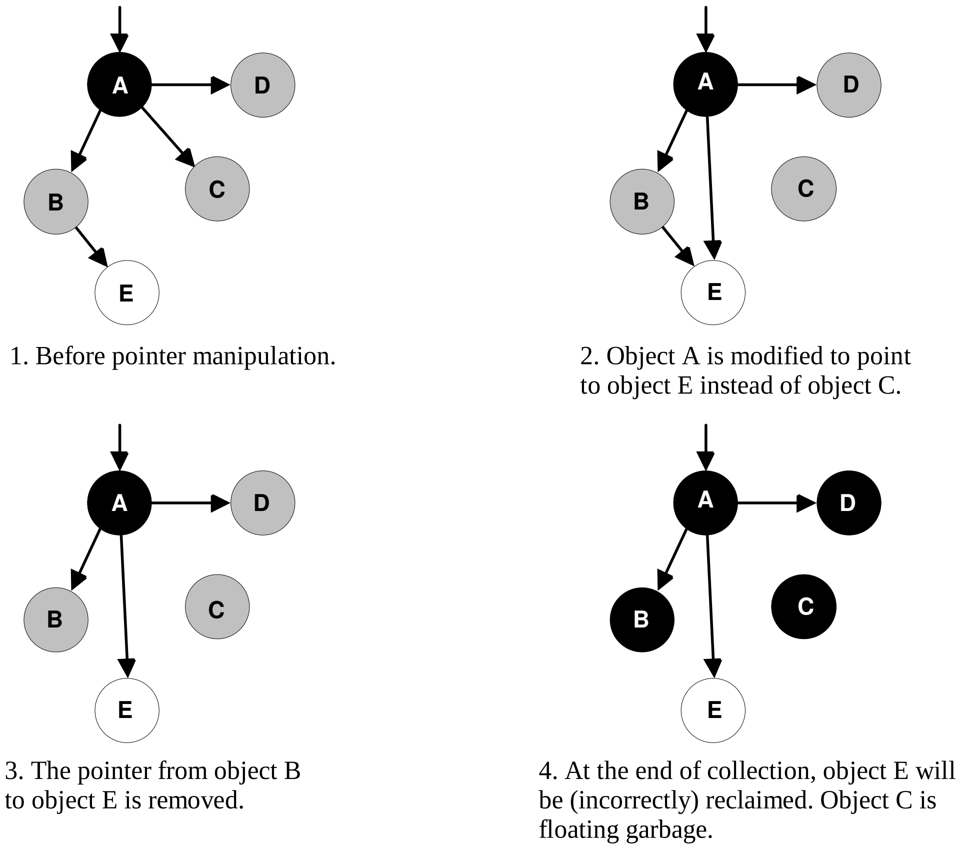

Figure 4.2: Example of violating the tri-color invariant

by the collector. (Recall that the garbage collector has already examined black objects, and will not look at them again.) If all other paths to the white object are broken before being reached by the collector, the white object will be reclaimed. Since the application can still reach the white object through the black object's pointer, this creates a dangling pointer.

To prevent this from ever happening, incremental collection algorithms must preserve the *tri-color invariant*. The tri-color invariant takes on two forms which we now define:

1. *The* **strong** *tri-color invariant*: For all black objects in the graph which have a path to a white object, *all* paths from that black object to that white object must contain at least one gray object [DLM+78].

2. *The* **weak** *tri-color invariant*: For all black objects in the graph which have a path to a white object, *at least one* path from that black object to that white object must contain at least one gray object [Yua90].

In other words, the strong tri-color invariant states that no black object may point directly to a white object, while the weak tri-color invariant states that there may be many pointers from black objects to white objects as long as there is at least one path from a gray object to that white object that does not contain a black object. Note that if the strong tri-color invariant holds, the weak tri-color invariant must also hold. Preserving the weak tri-color invariant is sufficient to ensure that all live objects will eventually be reached by the collector.

Proof Sketch for the Weak Tri-Color Invariant

To see that preserving the weak tri-color invariant will ensure that all live objects are eventually marked, consider that the weak tri-color invariant states that all live white objects have at least one path to them from a gray object and that path does not contain a black object. Garbage collection work will only shorten such paths. The only way these paths can get longer is if the application changes the graph by adding a new white live object. In this case, the ratio of the number of live white objects discovered by the collector to the number of new white objects added to the graph by the application must be greater than one,[8] so that the length of these paths can only decrease, and eventually the garbage collector will find all live white objects and terminate.

Comparison of the String and Weak Tri-Color Invariants

The strong tri-color invariant is much less conservative than the weak tri-color invariant in terms of what is considered reachable—it thus allows an important optimization: if pointers between white objects are broken, the garbage collector need not take them into account. If the white objects become unreachable from any gray objects, so much the better—they are garbage anyway and should not be traversed. It is therefore possible to reclaim some objects that become garbage *during* collection.

[8]The user is required to select this ratio based on the requirements of the application. See Section 4.10.

Still another optimization allowed by the strong tri-color invariant is that the collector can use any available oracle which can tell it if a white object is already garbage. For example, in combining garbage collection of some objects with explicit (programmer invoked) deallocation of others, the programmer may notify the collector that an object will never be used again beyond a particular point in the program's execution.[9] If the object is white or gray, the collector can use that information to short-circuit its traversal at that object. Assuming the oracle is correct, this will not erroneously reclaim any reachable objects—if there are other paths to other white objects, there will be gray objects on those other paths, and the collector will find them. This kind of optimization is also the essence of hierarchical garbage collection, which we will discuss in Section 4.14. Note that this optimization is not possible with the weak tri-color invariant. If an oracle tells you that an object is garbage you can only reclaim its space—you cannot short-circuit the traversal at that object because it might hold the only path, *known to the garbage collector*, to a set of live objects that will otherwise be erroneously reclaimed.

In the worst case, the collector will traverse all objects just before they become garbage, and the above two optimizations will have no effect on the amount of garbage reclaimed by the collector. Even though they provide no benefit for a hard real-time collector, we will see in Section 4.11 that these optimizations can be very useful for a soft real-time collector.

An important optimization for hard real-time collection *is* possible with the weak tri-color invariant: a pointer that has already been traced can be overwritten without violating the invariant. This is because once a pointer is traced, it is a pointer from a black object to a black or gray object, and hence cannot be a pointer on a path from a gray object to a white object with no intervening black objects. Also, any value written over this pointer must either be a copy of a pointer that exists elsewhere in the graph, or a pointer value that points to a new object. As long as all new objects are allocated black (see Section 4.4.2), and we optimistically trace all root objects at the beginning of collection, we can use this observation to avoid having a write-barrier to protect against the mutation of root objects.

In summary, an error in collection like that shown in Figure 4.2 requires that two things happen:

1. The mutator creates a pointer from a black object to a white object.

2. The mutator breaks the old path to the white object.

Maintaining the strong tri-color invariant requires that special action be taken to ensure the first case never happens. Preserving the weak tri-color invariant requires that special action be taken to ensure the second case never happens, because this might be the only path *known to the garbage collector* to that white object.

4.4.2 Allocation Color

When using a tri-color marking technique such as we do, newly allocated objects must be colored one of three colors: white, gray, or black. If newly allocated objects are colored white,

[9] Another example is combining tracing collection with reference counting, as is common in distributed garbage collection.

then no write-barrier is needed on initializing writes to these objects to maintain either the strong or the weak tri-color invariants. In addition, if the strong tri-color invariant is being preserved, and an object becomes garbage before being traced, then it can be reclaimed at the end of this garbage collection cycle. However, since the set of live white objects is no longer monotonically decreasing, the garbage collector must trace at a rate *greater than the rate of allocation* in order to ensure that the collection will eventually terminate.

If newly allocated objects are colored gray, then again neither the strong nor weak tri-color invariants can be violated by initializing writes to these objects. In addition, since the number of live white objects does not increase, the collector can trace at a rate much slower than when allocating white. However, since these objects are already shaded gray, if a newly allocated object becomes garbage before the end of this collection cycle, it cannot be reclaimed before the end of the *next* garbage collection cycle (for example, object C in Figure 4.2). In addition, the newly allocated objects must still be traced by the collector to look for pointers to untraced objects.

Finally, if newly allocated objects are colored black, then, again, the number of live white objects does not increase and the collector can trace at a rate much slower than when allocating white. In addition, since these objects are already black, they need not be considered by the collector at all during this collection cycle. However, initializing writes to these new objects can violate the strong tri-color invariant, so a write-barrier is needed if this invariant is used, although no write-barrier is needed for the weak tri-color invariant. In addition, if a newly allocated object becomes garbage before the end of this collection cycle, it cannot be reclaimed before the end of the next garbage collection cycle.

4.5 Incremental Tracing Algorithms

Two basic incremental tracing strategies are possible [Wil92]. One strategy is to ensure that objects can never get lost, by preventing any pointers from being destroyed [AP87, Yua90]. Before overwriting a pointer, the old pointer value is immediately traversed, or saved away so that the collector can still find it and trace it later. We call this a *snapshot at beginning* algorithm because the collector's view of reachable data structures is fixed when collection begins.

Snapshot at beginning algorithms rely on the weak tri-color invariant to ensure that all live objects are eventually marked. The weak tri-color invariant is preserved by using a write-barrier to detect any attempts to overwrite any pointers in the graph. The overwritten pointers are saved until the collector can process them. For example, in Figure 4.2 step 3, before the pointer from object B to object E is broken, it would be recorded, and object E would be traversed when this pointer is re-examined. This has the effect of generating a *snapshot* of the graph at the beginning of the garbage collection cycle.

An important optimization is possible with a snapshot algorithm: the initialization of new objects (by storing pointers in these new objects) does not need a write-barrier because there are no existing pointers to overwrite.

The other strategy for collection, *incremental update*, focuses on the writing of new pointers in objects that the collector has already reached and examined. When such a pointer is created, the collector is notified so that it can either trace the pointed-to object immediately, or re-examine the location in which the pointer was stored again later to find any "hidden" objects [Ste75, DLM+78, BDS91]. For example, in Figure 4.2 step 2, when the pointer from object A to object E is created, a pointer *to this pointer* is recorded, and object E would be traversed when this pointer is re-examined. That is, the collector's view of reachable data structures is incrementally updated in the face of changes to those data structures by the running program.

Incremental update algorithms are quite different from snapshot algorithms, because they rely on the strong tri-color invariant—*no black object is allowed to hold a pointer directly to a white object.* If a pointer to a white object is stored in a black object, the white object must be immediately grayed (added to the collector's traversal queue), or the black object must be reverted to gray (i.e., put back in the queue so that it will be re-examined later). This ensures that no untraced pointer will be hidden in an object that has been reached. It should be noted that the test for an incremental update write-barrier can be sped up considerably, at the cost of increased conservatism, by always assuming that any pointer store will violate the write-barrier and optimistically shading the R-value without checking the color of the L-value.

4.6 Non-Copying Incremental Read-Barrier Techniques

Wang [Wan89] and Baker [Bak91] independently presented a critical insight that can be used to make a mark-sweep collector have many of the advantages of a copying collector. Their insight was that in a copying collector, the "spaces" of the collector are really just a particular implementation of sets. The tracing process removes objects from the set subject to garbage collection, and when tracing is complete, anything remaining in that set is known to be garbage, and the set can be reclaimed in its entirety. Any implementation of sets will do, provided that the implementation has similar performance characteristics to a copying collector. In particular, given a pointer to an object, it must be easy to determine to which set it belongs. In addition, it must be relatively easy to move an object from one set to another. Finally, it must be easy to switch the roles of the sets at the end of collection.

Baker's incremental non-copying garbage collection algorithm [Bak91][10] uses doubly-linked lists (and per-object color fields) to implement the garbage collection sets, rather than separate memory areas. These lists are linked into a cyclic structure, as shown in Figure 4.3. This cyclic structure is divided into four sections: the *new-set,* the *free-set,* the *from-set* and the *to-set.*

The *new-set* is where allocation of new objects occurs during garbage collection; it is contiguous with the *free-set*, and allocation occurs by advancing the pointer that separates the two sets. In this way, an object is implicitly moved from the free-set to the new-set. New objects are allocated black, and at the beginning of garbage collection, the new-set is empty.

The *from-set* holds objects that were allocated before garbage collection began, and which are currently subject to garbage collection. In terms of tri-color marking, these objects are white. As the collector and mutator traverse data structures, objects are moved from the

[10] This algorithm is called the *treadmill* algorithm.

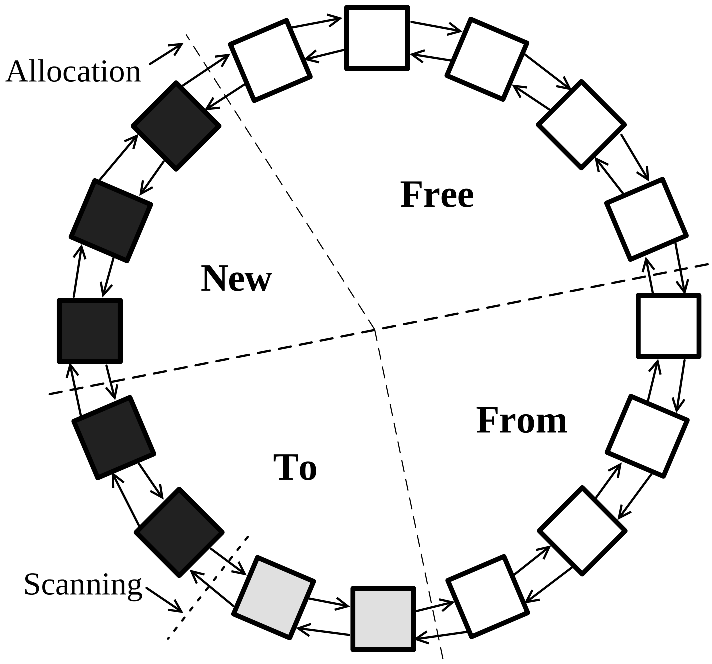

Figure 4.3: Treadmill collector during collection.

from-set to the *to-set* and colored gray by setting a bit in the object's header. The to-set is initially empty, but grows as objects are removed (unlinked) from the from-set and moved (linked) into the to-set during collection.

Eventually, all of the reachable objects in the from-set have been moved to the to-set, and scanned for offspring (converted from gray to black). When no more objects are reachable, all of the objects remaining in the from-set are known to be garbage. The from-set is now available, and can simply be merged with the free-set. The to-set and new-set both hold objects that were preserved, and they can be merged to form the new from-set for the next collection. Since at this point, the free-set is empty, rather than changing the color bits in all of the object headers, the meaning of the bit is changed. This implicitly colors all old objects white so that collection can begin again.

The new state of the collector is very similar to the state at the beginning of the previous garbage collection cycle, except that the segments have "moved" part of the way around the circle—hence the name "treadmill."

In order to keep the mutator from confusing the collector, Baker uses a read-barrier to synchronize the mutator's view of the data with the collector's view. If the mutator is about to access an object in the from-set, the read-barrier first moves the object to the to-set, and then returns a pointer to the object. This approach has a similar disadvantage to Baker's incremental copying collector in that it will systematically perform poorly at the beginning of every garbage collection cycle, as all object references are to objects in the from-set. However, the cost of moving an object in this approach is now constant, rather than proportional to the size of the object as it was in the copying version.

4.7 Non-Copying Incremental Write-Barrier Techniques

We have revived incremental techniques which were designed in the mid-1970's for (non-real-time) concurrent garbage collection, and applied them to a real-time collector for stock uniprocessors. Coordination between the running application and the collector's tracing traversal is via a write-barrier, i.e., the collector's view of data structures is updated only when the application *modifies* the graph of pointer relationships. Compared to Baker's *read-barrier* technique, which coordinates the collector with the application whenever the application reads or compares pointers, a write-barrier reduces coordination costs and makes them much more predictable.

Write-barrier algorithms coordinate the collector's view of data structures with the running application's view. Whenever the application modifies a data structure by changing a pointer, the collector must be protected from being confused and "losing" objects. This can happen if an object is hidden from the collector by storing a pointer to the object in another object that the collector has already examined, and then breaking all other paths to the first object, as in Figure 4.2.

4.8 Our Testbed Implementation

To test the relative effectiveness of the various garbage collection strategies presented above, we have implemented a fully configurable, non-copying, implicit reclamation garbage collector. In what follows, we will describe one possible configuration of this collector, and then present some performance figures for that implementation. It should be noted that the results gathered from these experiments will be valid for any write-barrier collection strategy, not just for non-copying implementations.

When in hard real-time mode, our collector is configured to use a *snapshot at beginning* write-barrier with black allocation. Our collector is currently configured like this for 4 major reasons:

- *No write-barrier on initializing writes:* Initializing writes in newly-allocated objects need incur no write-barrier overhead because such stores overwrite no existing pointers, and therefore cannot violate the weak tri-color invariant.

- *No write-barrier on pointer stores in root variables:* Because we trace all root objects at the beginning of collection, pointer stores in local or global variables need incur no write-barrier overhead; such writes cannot violate the weak tri-color invariant.[11]

- *Slower rate of tracing is required:* The garbage collector can trace at a rate that is much slower than if we allocate white, and therefore incur much less run-time overhead. In fact, with black allocation the tracing rate can be made arbitrarily slow as the amount

[11]See Section 4.4.1 for a more detailed discussion as to why this is true.

of memory is increased. With white allocation, the tracing rate can never be slower than the rate of allocation.

- *New objects need not be considered by the collector:* Newly allocated objects potentially have as much as one full garbage collection cycle to die before they are traced by the collector. If they die before the end of the cycle during which they are allocated, then they are never considered by the collector. Since a majority of objects die very quickly after being allocated, this is likely to significantly reduce the amount of garbage collection work that needs to be performed. We expand on this idea considerably in Section 4.14.

4.8.1 Non-Copying Implicit Reclamation

The testbed garbage collector we have implemented combines an *incremental update* write-barrier with a generalization of Baker's non-copying *implicit-reclamation* strategy [Bak91], so that objects not yet reached need not be traversed to be reclaimed, as is necessary in the sweep phase of a mark-sweep collector.

A non-copying implicit-reclamation collector achieves the same effect as a copying collector by maintaining data structures that record which *set* objects are in, and "moving" an object from one set to another rather than literally copying it from one area of memory to another. We implement these sets as doubly-linked lists, plus a header field in each object denoting the set in which it resides.

Figure 4.4 shows an example of one of these doubly-linked lists. The white objects are all of the objects between the free pointer (inclusive) and black pointer (exclusive); the black objects are those objects between the black pointer (inclusive) and the scan pointer (exclusive); and the gray objects are those from the scan pointer on to the right.

Figure 4.5 shows how an object is grayed. It is unlinked from the white set, linked into the gray set, and a bit in its header is set to indicate that it is now gray. If it is put on the right end of this list (as shown in the example) then the garbage collector's traversal is breadth-first. If it is put on the left end, then the traversal is depth-first.

Figure 4.6 shows how an object is blackened. First the object under the scan pointer is scanned for pointers to white objects. If any white objects are found, they are put into the gray set. Finally, the scan pointer is moved one object to the right. After all reachable data have been moved from the white set to the black set, the remaining objects in the white set are known to be garbage and that list can simply be appended to the free list (the objects to the left of the free pointer, not shown in figures 4.4 to 4.6) in small constant time.

This pseudo-copying is cheaper than real copying for most objects, and also avoids the need to keep the relocation of objects from confusing the running application.[12]

Our collector generalizes this scheme by combining it with a simple *segregated storage* scheme for the management of different-sized chunks of memory. In such a scheme, separate

[12]The main motivation for Baker's read-barrier is really to keep the running application from seeing temporary inconsistencies in data structures while they are being copied by the collector—that is, the read-barrier protects the application program from changes made by the collector, as well as keeping the collector from being confused by the application's changes.

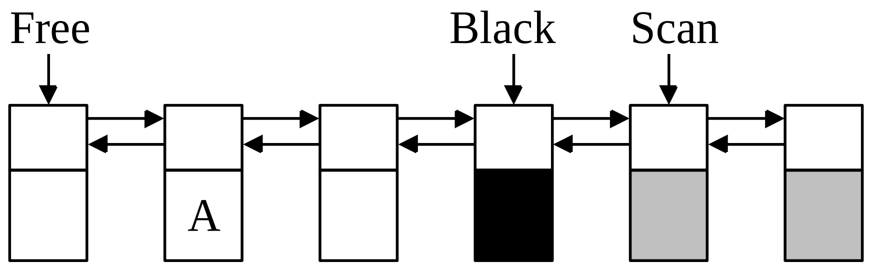

Figure 4.4: The initial state of the heap

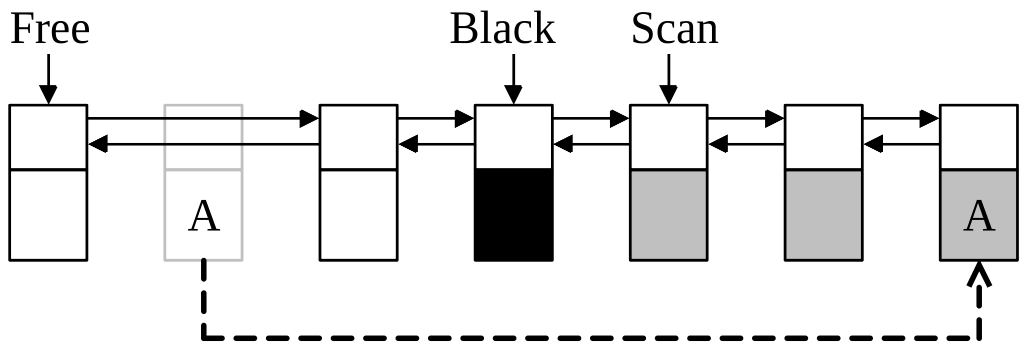

Figure 4.5: Graying a white object

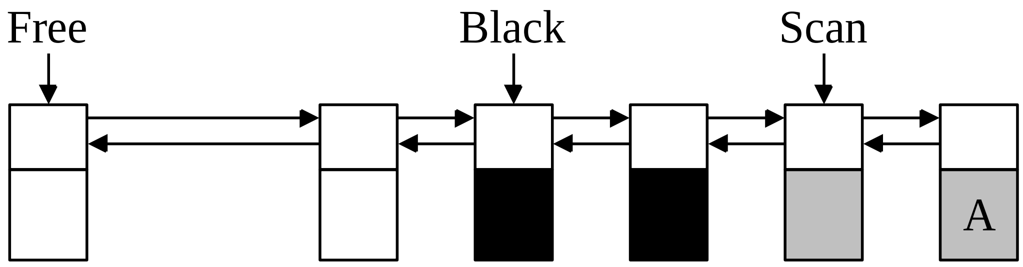

Figure 4.6: Blackening a gray object

sets of lists are used to manage different-sized objects, with objects of similar sizes grouped into "size classes."

Currently, our size classes are powers of two. When allocating an object, we round its size up to the next power of two and allocate it in that size class. Like any non-compacting storage scheme, our segregated storage allocator is vulnerable to fragmentation. As we showed in Chapter 2, a simple segregated storage policy is one of the worst policies in terms of average expected fragmentation. We chose this scheme to make garbage collecting C++ easier (see Section 4.13); *any* non-moving policy could be used as long as memory can be allocated in bounded time. The current scheme does no coalescing of adjacent free blocks into larger free blocks.

As we said in the discussion of Figures 4.4 to 4.6, a bit is set in the header of an object when it becomes gray. When a garbage collection cycle is finished, the entire black set of objects is considered to be white so that the next cycle can begin. Baker can do this in his treadmill without changing the color bit in the header of every black object, by changing the meaning of the bit itself.

One complication of using segregated storage is that we cannot count on ever exhausting a given free list, as is required by Baker's scheme; that is, free chunks may remain on a free list for an arbitrary number of collection cycles. This means that after a set of objects is reclaimed, we cannot simply change the meaning of the color bit patterns in the object's header as Baker describes; we must actually change the color field of each object that we reclaim. While this is not as elegant as Baker's scheme, because it introduces a cost proportional to the number of objects reclaimed, in practice it does not really matter much because this cost is very small and can be done in real-time.

We do this recoloring in real-time by deferring it until the garbage blocks are reallocated. I.e., we append the garbage objects to the list of free objects, so that the free-set contains objects of multiple colors, and lazily reset the headers at allocation time.[13] Since neither the application nor the collector examines the objects in the free-set until they are reallocated, this works just as well, and spreads the cost over the allocations, rather than incurring it all at once.

4.9 Real-Time Timing Requirements

We describe our garbage collector as a truly real-time collector. Before we can make that claim we must demonstrate that two requirements are met. First, we must show that every garbage collector pause is strictly bounded, and second, we must show that these pauses do not occur too frequently *at a time-scale relevant to the running application.* To show that every garbage collector pause is bounded, we will describe the algorithm in some detail, focusing on the computational costs of each operation. To show that these pauses do not occur too frequently, we will show how our collector's work is tied to the rate of allocation, and to a lesser extent, to the rate of mutation of pointers.

In the following subsections, we will only be describing the hard real-time mode of our

[13] Conceptually, we re-color all of the freed chunks of memory at the instant a collection is complete, but the chunks' color fields are only reset when they are reused to allocate new objects.

collector. In Section 4.11 we will look at these same details with respect to the soft real-time mode of our collector.

4.9.1 Allocating Memory

As we said earlier, the current implementation of our garbage collector uses a segregated storage system for its lists of free blocks.[14] When a request for a free block is processed, the smallest power of two size which is larger than the requested size is computed. This computation is done in very small constant time by a simple table lookup.[15]

If bounds on the memory usage of the running application can be computed in advance (see Section 4.10 for a detailed discussion on this) then memory can be pre-allocated for each size class. In this case, the remainder of the allocation work reduces to a check to see if a garbage collection increment must be performed, and if not, the size of this request is recorded, the free pointer is incremented in the doubly linked list that represents the appropriate size class, and a pointer to the newly allocated block is returned.

If bounds on the memory usage of the running application cannot be computed in advance, then the additional case of needing to request more memory from the operating system is added to the work needed to be performed to allocate a block. If, at any time, there are no free blocks in the appropriate size class, then another free block is created out of a raw page of memory which is allocated by the operating system.[16] For size classes of less than 4K bytes, a single 4K page of memory is used for each size class to satisfy the need for additional objects. Objects are carved out of this page one at a time and returned to the application. This operation involves a conditional and the adjustment of a few pointers. If this page was exhausted by the previous new object request, then another page is requested from the operating system. For size classes greater than 4K bytes, the memory for each new object is directly requested from the operating system and these pages are recorded as being large object pages.

4.9.2 The Write-Barrier

When our garbage collector is configured for snapshot at beginning, our write-barrier is very simple. It only needs to check whether the object on the right hand side of the assignment (R-value) is already shaded (colored black or gray). If not, then the R-value object is immediately grayed. On most modern architectures, this can be performed in a small number of instructions.[17] On a Pentium using the GNU g++ compiler version 2.7.0, our write-barrier's

[14] The choice of a simple segregated storage scheme was made for simplicity of implementation, and for adaptability to C++. Nevertheless, we feel it necessary to describe the timing requirements of the segregated storage algorithm here, so that the reader can feel comfortable that at least one algorithm exists which can meet our hard real-time goals as set forth in Section 4.1.

[15] For object sizes less than 255 words, we use a simple table lookup. For larger object sizes, we use a conditional, a bit shift, and a table lookup. For the largest object sizes, we have four conditionals, one bit shift, and a table lookup. If the size is known at compilation time, the conditionals and bit shift can be optimized away. With very aggressive optimization, the table lookup can also be eliminated.

[16] Here, by page, we mean a logical page of memory and not a physical page as defined by the virtual memory system of the host machine. In fact, for a hard real-time application, virtual memory (or at least paging) will not be used at all.

conditional is just 11 instructions. If this conditional determines that the R-value needs to be shaded, then an additional 48 instructions are required to gray the object. Note that graying the old R-value object is an operation that *must* eventually be done; if these objects were not found by the write-barrier, then they would eventually be found by the collector's marking traversal, so over the course of the computation, no additional work is performed by shading these objects eagerly.

4.9.3 Performing an Increment of Garbage Collection Work

During the first increment of a garbage collection cycle, the root set is traced *atomically* and all objects pointed to by root variables are grayed. This gives us a cost directly proportional to the number of root variables. As we described in Section 4.8 this allows us to forgo the need for a write-barrier on assignments into root variables.

Subsequent increments of garbage collection work attempt to blacken gray objects that were created by either the atomic root scan, or the blackening of previous gray objects. These increments are limited to blackening a user-defined maximum number of bytes. So, the cost of an increment is proportional to the number of bytes the user specifies to blacken per increment, and the worst-case cost is incurred when every one of these bytes makes up a pointer.

Once there are no more grays to blacken, collection completes. At this point, the garbage collector has a choice of whether to reclaim garbage objects immediately, or to continue allocating without collecting. If the collector is coloring newly allocated objects white, then the garbage objects are immediately reclaimed, and collection starts again. If the collector is allocating black (recommended for hard real-time applications) and there still exist available free objects, then garbage collection stops until some object size is unavailable. At that time the garbage objects are reclaimed and garbage collection begins again.

The rate and duration of pauses in the running application due to garbage collection work is directly related to the number of bytes allocated by the application. The user chooses how many bytes can be allocated between interruptions by the collector, and how many bytes should be reclaimed during each interruption. These two parameters determine the length and frequency of garbage collection pauses. Figure 4.7 is a histogram of garbage collection pauses for the Hyper program being traced at a rate of blackening 4K bytes for every 8K bytes allocated.[18] The X-axis shows the length of garbage collection pauses (in milliseconds on a Pentium running Linux).[19] The minimum, average, and maximum pauses for this

[17]We first load and mask to get the color of the R-value object from its header. We then load the current shade value (recall that the bit pattern used to indicate that an object is shaded (gray or black) is not changed in each *object* from collection cycle to collection cycle, but rather the *meaning* of the bit pattern changes) and compare that to the color of the R-value object. Finally, we conditionally gray the R-value object if it is not already the current shade value. Graying an object involves setting a bit in the header of the object, and six pointer modifications.

[18]We refer to these values as "throttle settings". A throttle setting of 2 means that 2 bytes are traced for every byte allocated. The term "throttle setting" is meant to suggest the throttle on an engine. The higher the throttle setting, the faster the garbage collector runs.

[19]Timings are in milliseconds on a 90 MHz Pentium running Linux 1.2.13, compiled with g++ 2.7.0 and full optimization. The cycle counts were recorded using the **rdtsc** opcode. This opcode returns the value of a 64-bit register that is incremented with every cycle.

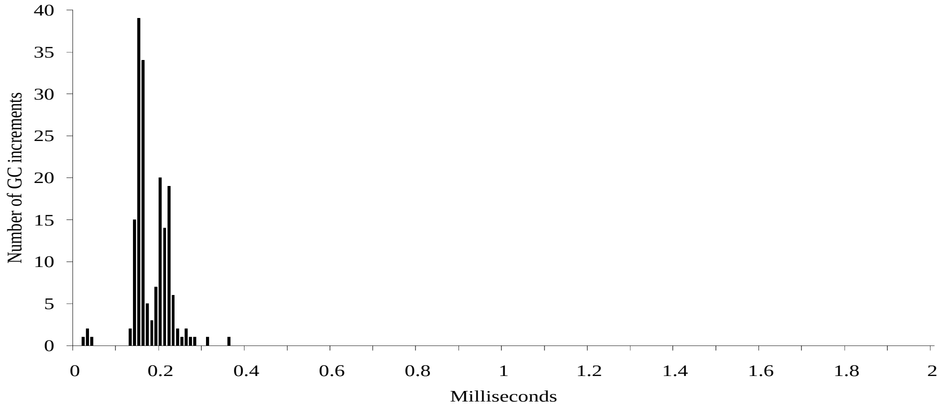

Figure 4.7: Histogram of garbage collection increment costs for the Hyper program (Throttle 0.5)

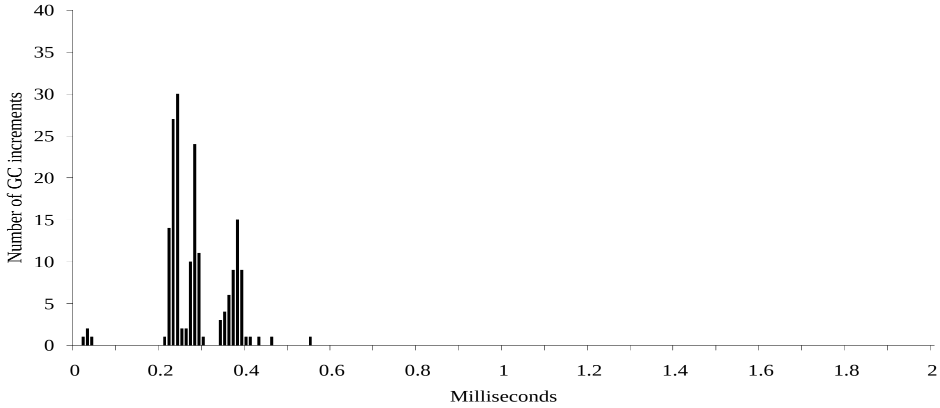

Figure 4.8: Histogram of garbage collection increment costs for the Hyper program (Throttle 1.0)

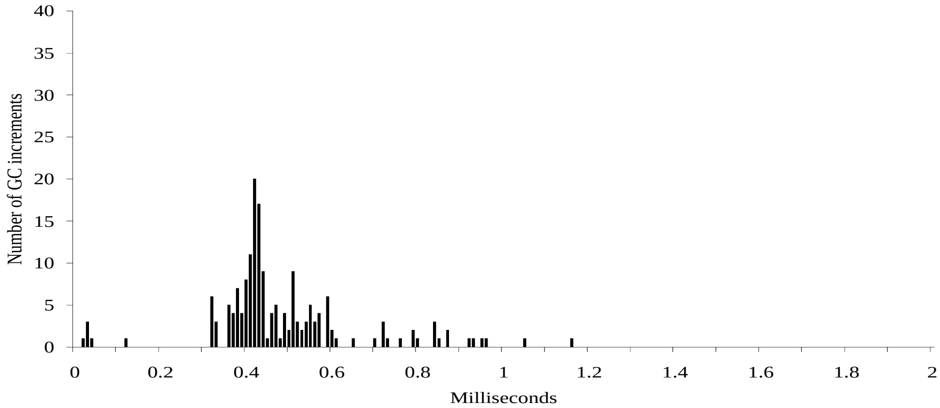

Figure 4.9: Histogram of garbage collection increment costs for the Hyper program (Throttle 2.0)

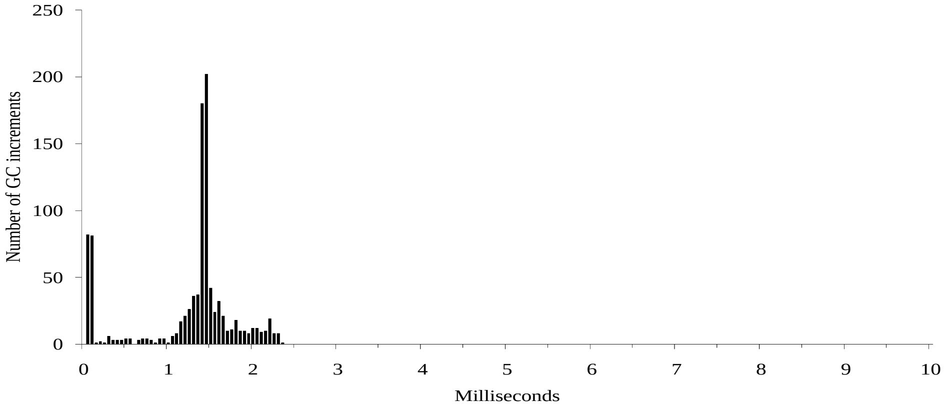

Figure 4.10: Histogram of garbage collection increment costs for the Grobnerprogram (Throttle 0.5)

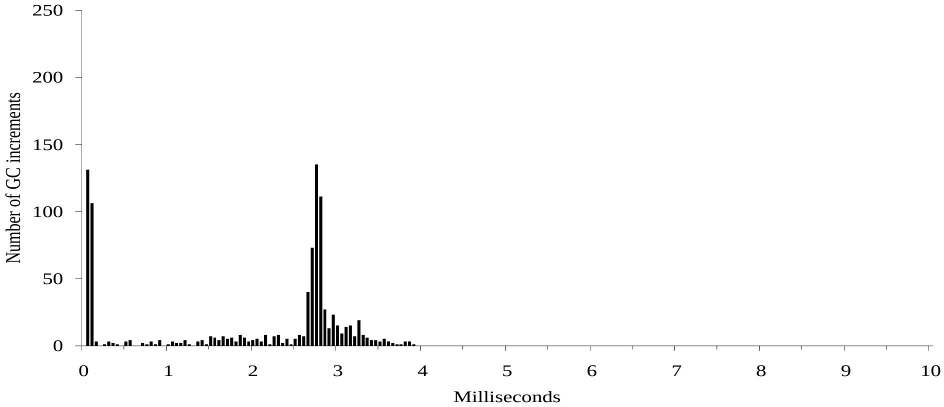

Figure 4.11: Histogram of garbage collection increment costs for the Grobnerprogram (Throttle 1.0)

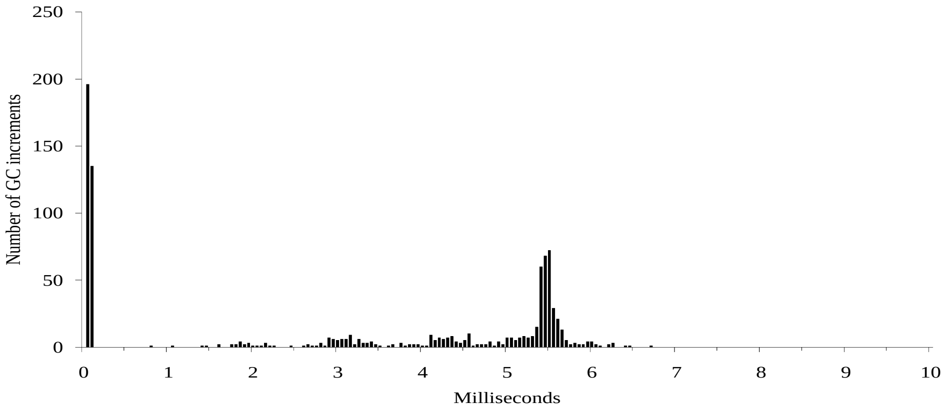

Figure 4.12: Histogram of garbage collection increment costs for the Grobnerprogram (Throttle 2.0)

program were 0.01, 0.17, and 0.35 milliseconds, respectively. Figure 4.8 is a histogram of the garbage collection pauses for the same program tracing at a rate twice as fast (8K bytes for every 8K bytes allocated). The minimum, average, and maximum pauses for this program were 0.01, 0.28, and 0.55 milliseconds, respectively. Finally, Figure 4.9 is a histogram of the garbage collection pauses for the same program tracing 16K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were 0.01, 0.48, and 1.18 milliseconds, respectively.

Figures 4.10 is a histogram of garbage collection pauses for the Grobner program tracing 4K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were 0.03, 1.21, and 2.30 milliseconds, respectively. Figure 4.11 is a histogram of garbage collection pauses for the Grobner program tracing 8K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were 0.03, 1.94, and 3.89 milliseconds, respectively. Finally, Figure 4.12 is a histogram of garbage collection pauses for the Grobner program tracing 16K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were 0.03, 3.01, and 6.69 milliseconds, respectively.

The above timing results were for C++ programs, and include many costs not directly related to the garbage collector. The most significant of these costs could be removed if the compiler supplied better support for garbage collection. In particular, the garbage collection increment costs include the time required to derive the start of objects. Languages like Scheme, Eiffel, and Java provide this support.

4.10 Memory Bounds

For hard real-time applications, it is not good enough to ensure that all program pauses can be predicted. In addition, the program must not exceed its resource bounds. In particular, the user of our garbage collector must know what the worst-case memory usage is for a particular application, in order to guarantee that it will have enough memory to run within the required real-time parameters.

For a copying garbage collector, the worst-case memory bound is fairly straightforward to compute. When allocating new objects black, the bound is computed by adding the maximum number of live bytes to the number of bytes allocated per full garbage collection, and then multiplying this value by two (to account for the two spaces a copying collector uses).

There are many different issues involved in calculating the space bounds for a non-copying algorithm, and all of the answers are not immediately clear. The current configuration of our testbed garbage collector uses the simple segregated storage policy. To calculate the worst-case memory usage for a garbage collector using this policy, the user must first know the maximum amount of live data for each size class. Given this information, the worst-case memory usage for the program can be computed by determining the worst-case memory usage for each size class and then summing these values.

The worst case memory usage for a single size class is simply the maximum amount of memory that can be live in that size class at any given time, plus twice the amount of memory that is allocated per full garbage collection cycle.[20] To see this, consider the case where the maximum number of objects for that size class are live, and we have just completed a garbage collection cycle. In the worst case, one new object will be allocated and immediately shaded, and then immediately freed, making it garbage. This process repeats for the entire garbage collection cycle, and no other objects of any other size are allocated. In this case, none of the newly allocated objects will be reclaimable until the end of the next full garbage collection. We will need enough memory for the maximum live objects, and for every object that was allocated during *this and the next* full garbage collection. At the end of the next full garbage collection, the objects that become free during this garbage collection will be reclaimed and can be reused if this process continues.

If even a single object is allocated from some other size class, then that is a few less bytes that need to be allocated from this size class before two complete collections finish, and hence is not as bad as the worst-case discussed above. Since the application can do this for one size class, and then repeat it for another size class two full collection cycles later, and since we do not move memory from one size class to another, the worst case is the summation of the worst cases for each size class.

Note that the above analysis assumes that nothing is known about the relative phase behavior of the use of the different sizes of objects. In particular, it assumes the worst case, which is that there is *no overlap* in the use of different sizes of objects. If, for example, it is known that during one phase of a program, a minimum of 100K of a particular size of object will always be in use, then that is 100K that cannot be allocated in another size during that phase.

Another important optimization possibility comes from recognizing that, for every distinct size class of objects, there is an overhead which is some factor of the maximum number of live bytes that is added to the overall memory requirement. So, reducing the number of different size classes can have a profound influence on the total memory requirement. One very simple way to reduce the number of different size classes in use is to simply add padding to objects of one size so that they fall into a larger size class.[21] Another slightly more difficult way to reduce the number of different size classes in use is to split some objects into two or more smaller objects. While this adds slightly to the overall complication of the program, it can have profound results in lowering the overall memory requirements.[22]

4.10.1 Naive Memory Computations for Eight Real C and C++ Programs

A disadvantage of simple segregated storage is that it can possibly suffer from severe memory fragmentation, as we showed in Chapter 2. We believe that in practice, for real programs, this

[20] For example, if the maximum number of 32 byte objects that can be live at any given time is 10,000 and a full garbage collection is performed for every megabyte of memory allocated, then the maximum memory bound for this size class is 2.3 megabytes. That is, 2 megabytes + 10,000 * 32 bytes.

[21] While at first glance, this would seem to only make things worse, what we are actually doing is trading internal fragmentation for external fragmentation—something that many common allocator algorithms do all of the time. As we showed in Section 2.9.2, on average this is not a good idea. However, in the *worst case*, this can result in much smaller memory requirements.

[22] The gains in simplicity due to adding garbage collection to real-time programs are likely to far outweigh any increase in complication due to splitting the occasional class into two or more pieces.

will not be much of a problem. To get an idea of the amount of memory that would be needed to use simple segregated storage with real programs, we performed the analysis described in Section 4.10 on eight memory-intensive C and C++ programs. These programs are the same programs that we used for our allocator studies, and are described in detail in Section 2.6.2. We stress that these are naive computations with no knowledge of the application being studied.

The first part of our analysis involves discovering the maximum number of objects of each size class that could be alive at the same time. Since we do not have a deep understanding of the algorithms used by all eight of these programs, we substitute *measured* values. Although these values are by no means the maximum possible values, they are at least representative for a real workload, and it is reasonable to assume that these numbers will provide a flavor for what the actual worst case memory requirements will be. However, this method is not suggested as a substitute for real worst case analysis. Tables 4.1 and 4.2 show the maximum memory usage of our eight programs for each size class in the program *for the particular input set described in Section 2.6.*

Object Size	16	32	64	128	256	512	1K	2K	4K
LRUsim	41	30	39010	0	9	0	1	0	1
gcc	14665	57204	11846	3420	172	18	73	18	18
Espresso	10	98	4325	18	11	13	5	8	6
Ghostscript	0	552	10472	2644	1317	318	36	4	33
Grobner	3664	7185	483	36	80	19	4	1	0
Hyper	0	1	292	0	0	0	0	2	0
P2C	1383	5595	5336	1517	1	1	10	1	0
Perl	377	248	1284	28	4	5	3	2	1

Table 4.1: Maximum number of live objects per size class (part 1)

Object Size	8K	16K	32K	64K	128K	256K	512K	1M	2M
LRUsim	1	1	0	0	0	0	0	0	0
gcc	18	18	18	4	1	4	4	0	0
Espresso	8	7	2	4	0	0	0	0	0
Ghostscript	2	2	4	0	0	0	0	0	0
Grobner	0	1	0	0	0	0	0	0	0
Hyper	0	0	0	0	0	0	1	0	1
P2C	0	2	0	0	0	0	0	0	0
Perl	1	1	0	0	0	0	0	0	0

Table 4.2: Maximum number of live objects per size class (part 2)

By looking at Tables 4.1 and 4.2 and rounding up some objects to a larger size class to reduce the number of different size classes, we can come up with an upper bound on the amount of memory that would be needed to run these eight programs with a hard real-time collector, *for this input set.*

Table 4.3 shows the amount of memory that would be required to run each of our programs in the cases of tracing 0.5, 1, and 2 bytes for every byte allocated. "Max Live Bytes" is the total maximum number of live bytes for any point in the program run; "Bytes Allocated" is the total number of bytes allocated in the run of the program; and "Memory Required 0.5", "Memory Required 1", and "Memory Required 2" are upper bounds on the amount of memory required to run the programs for throttle settings of 0.5, 1, and 2 respectively.

	Max Live Bytes	Bytes Allocated	Memory Required 0.5	Memory Required 1	Memory Required 2
LRUsim	1,413,565	1,430,400	14,006,240	8,352,628	5,524,858
GCC	2,376,053	18,403,819	60,022,800	32,611,098	22,622,153
Espresso	269,647	106,892,887	5,067,060	3,449,050	2,309,771
Ghostscript	1,136,887	50,169,140	21,736,560	12,641,464	8,349,660
Grobner	148,537	4,081,352	2,527,404	1,636,182	1,190,603
Hyper	2,097,980	7,555,396	11,031,984	6,836,088	4,738,044
P2C	402,432	4,752,046	5,219,072	3,609,472	2,418,944
Perl	71,327	33,844,674	1,083,508	655,546	441,565

Table 4.3: Memory needed to run real C and C++ programs

Note that in some cases, the memory usage is startlingly poor. However, for the given input sets, these are strictly upper bounds computed without any consideration towards optimizing the memory usage, and actual memory use would most likely be much lower. If more effort is put into understanding the characteristics of the programs, particularly paying attention to objects which can be split into two or more smaller objects to eliminate a size class, and to phase behaviors which can cause use of one size class of objects to overlap the use of another, the worst-case amount of memory required can be made considerably smaller.

Again, we stress that these are naive computations with no understanding of the application. We believe that in practice, with some understanding of the application's memory requirements, it will not be difficult to achieve good memory bounds for most hard real-time programs. Showing this more conclusively is future work. We also believe that for soft real-time programs, adequate testing will be sufficient to show that fragmentation is not much of a problem.

4.11 Soft Real-Time Programs

Our garbage collector can be configured to run for soft real-time programs with a slight degradation in the worst-case performance, but with considerable improvements in the average-case running times.

There are two different aspects of soft real-time:

1. *Soft time.* A soft-time collector should be configured with an emphasis on improving run time overhead, at the expense of very occasionally missing hard real-time bounds.

2. *Soft space.* A soft space collector should be configured with an emphasis on improving memory usage, at the expense of very occasionally exceeding resource bounds.[23]

A soft-time or soft-space collector can use probabilistic reasoning for computing its worst case performance, and can be tested with representative workloads. This is a very different approach than with a hard-time or hard-space collector where absolute guarantees are required.

4.12 Adjusting the Rate of Garbage Collection

There are three related parameters to the garbage collector that need to be set by the user: the amount of memory to allocate per *full* garbage collection, the amount to allocate per garbage collection *increment*, and the amount that should be *traced* per increment. We call this last parameter the garbage collector's "throttle setting" and define it as the ratio of bytes traced to bytes allocated. For hard real-time applications, the user is required to determine the maximum number of live bytes that can occur at any point in the program. If the user chooses a throttle setting of 0.5, then two times max live bytes will be allocated per full garbage collection cycle. The user is free to set the number of bytes allocated per increment of collection as needed in order to meet the real-time bounds.[24]

4.13 Interface to C++

Currently, our garbage collector uses a *smart pointer* interface to collect C++ programs [Ss92]. In this interface, garbage-collected objects have an associated pointer type defined in a library as a parameterized class, and client code must use these pointers rather than raw C++ pointers. (Parameterization and operator overloading make this relatively easy, although smart pointers cannot be used quite as flexibly as raw pointers [Ede92].) The main difference between our parameterized pointers and normal pointers is that pointer assignments execute an additional few lines of code, which constitute the write-barrier.

In our system, each object has a hidden header field, created by our overloaded version of the C++ `new` operator. This header is used by the garbage collector to look up a *type descriptor* which describes the layout of pointers within the object. The actual type descriptor information is constructed by compiling the program with the debugging option turned on, and by using a special program which extracts structure layouts from the debugging information [Kak97]. (We use code from the GNU `gdb` debugger for this, so our collector should be easily portable to any system that uses a debugging output format that `gdb` understands.)

[23] Clearly, a program that exceeds its hard resource bounds is simply incorrect. However, a program that exceeds its soft resource bounds may be able to resort to paging to continue execution. It is this second sense that we are interested in.

[24] Actually, the computation is slightly more complicated if the user is using a *snapshot at beginning* write-barrier. In this case, since the root set is atomically scanned at the beginning of collection, and because this is the only work done during this increment, one extra increment of garbage collection work is needed to complete the collection cycle. So, if the user chooses a throttle setting of 0.5, then two times maximum number of live bytes *plus* the number of bytes to allocate per increment will be allocated per full garbage collection cycle.

Reading this header information from an object requires that the garbage collector always have access to the start of the object. In C++, however, there are many common cases where a pointer will point to the middle of an object.[25] We therefore chose to use a segregated storage scheme, with all objects aligned on known word boundaries, to make it relatively easy to derive the start of objects from these interior pointers.

Unfortunately, recovering object headers from derived pointers is the major source of overhead slowing down the write-barrier. If the compiler or programmer can declare that a pointer will always point to the beginning of an object (or some fixed offset into it), much of this cost can be optimized away. In many languages this is trivial, and it appears that this optimization is easy for a C++ compiler in many common cases.

With compiler cooperation our collector would be trivial to use, and more efficient than the current smart-pointer version. While we currently use our collector for C++, it could easily be adapted for use with any garbage-collected programming language. We have ported the collector to the "RScheme" Scheme system, and the implementors of the Tower Eiffel compiler have also ported it for use with their system.

4.14 Generational Collection

Generational techniques can greatly improve the efficiency of garbage collection for most programs by focusing garbage collection on young objects, which are likely to be short-lived. The minority of objects that survive for a longer period are made exempt from most garbage collection cycles so that they may have more time to die before again being considered for garbage collection [LH83, Moo84, Ung84, Wil92].

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. Therefore, for purely hard real-time systems, they may not be attractive. However, for general-purpose systems with mixed hard and soft deadlines, or for hard real-time systems with very regular periodic tasks,[26] the normal-case efficiency gain is likely to be highly worthwhile.

The choice of an *incremental update* write-barrier strategy works well for generational collection. A generational collector must use a write-barrier so that it can find pointers from old (infrequently collected) objects to young (frequently collected) ones. The generational write-barrier essentially records very similar information to that of an *incremental update* write-barrier, so most of the overhead should be able to serve both purposes.

4.14.1 Discussion

Generational collection can be combined with real-time techniques, but in the general case, the marriage is not a particularly happy one [WJ93]. Typically, generational techniques improve

[25] Pointers may point to an element of an array or a substructure of a record. In addition, because of the usual C++ implementation strategy for multiple inheritance, a pointer may also point to a subcomponent of an object whose layout is a concatenation of the layouts of classes from which it is derived.

[26] In a hard real-time system, periodic tasks may determine lifetimes in a predictable way, making the generational "heuristic" reliable.

expected performance at the expense of worst-case performance, while real-time garbage collection is oriented toward providing absolute worst-case guarantees. If the generational heuristic fails and most data are long-lived, garbage collecting the young generation(s) will be a waste of effort because no space will be reclaimed. In that case, the full-scale garbage collection must proceed just as fast as if the collector were a simple, non-generational, incremental scheme.

Real-time generational collection may be desirable for many applications, however, provided that the programmer can supply guarantees about object lifetimes to ensure that this scheme will be effective. This may be relatively easy to do for a class of real-time programs that are made up of a set of periodic tasks. Alternatively, the programmer may supply weaker "assurances," at the risk of a failure to meet a real-time deadline if an assurance is wrong. The former reasoning is necessary for mission-critical hard real-time systems, and is necessarily application-specific. The latter "near-real-time" approach is suitable for many other applications such as typical interactive audio and video control programs, where the possibility of a reduction in responsiveness is not catastrophic.

When it is desirable to combine generational and incremental techniques, the details of the generational scheme may be important to enabling proper incremental performance. For example, the Symbolics, LMI, and TI Lisp machines' collectors are the best-known "real-time" generational systems, but the interactions between their generational and incremental features have resulted in a major effect on their worst-case performance. Rather than garbage collecting older generations slowly over the course of several collections of younger generations, only one garbage collection is ongoing at any time, and that collection processes only the youngest generation, or the youngest two, or the youngest three, etc. That is, when an older generation is collected, it and *all younger generations* are effectively regarded as a single generation, and collected together. This makes it impossible to benefit from the generational effect of younger generations *while* garbage collecting older generations; in the case of a full garbage collection, it effectively degenerates into a simple non-generational incremental copying scheme. This will cause systematic performance losses during large-scale garbage collections.

During such large-scale collections, the collector must operate fast enough to finish tracing before the available free space is exhausted, since there are no younger generations that can reclaim space and reduce the safe tracing rate. Alternatively, the collection speed can be kept the same, but space requirements will be much greater during large-scale collections. Therefore, for programs with a significant amount of long-lived data, this scheme can be expected to have systematic and periodic performance losses. This will be the case even if the program has an object lifetime distribution favorable to generational collection and the programmer can provide the appropriate guarantees or assurances to the collector. Either the collector must operate at a much higher speed during full collections, or memory usage will go up dramatically. The former typically causes major performance degradation because the collector uses most of the CPU cycles; the latter either requires very large amounts of memory, negating the advantage of generational collection, or incurs performance degradation due to virtual memory paging.

4.14.2 How to Make a Generational Collector Real-Time

To avoid the problem of having to periodically collect all of memory, as discussed above, we largely decouple the collection of one generation from that of the other generations. The idea is that older generations (generations that contain long-lived objects) should be collected slowly and steadily, while younger generations (generations that contain relatively short-lived objects) should be collected quickly and steadily. Recall that in the non-generational version of our collector, one increment of garbage collection work is done for each increment of memory allocation. Similarly, in the generational version of our collector, one increment of garbage collection is done in the older generation for each increment of collection in the younger generation, where the increment performed in the older generation is some percentage of the duration of the increment in the younger generation.

We divide the collection of each generation into two phases to take advantage of the observation that many short-lived objects which hold pointers to older objects will be created, and then later die before the older generation is finished with its collection. In this case, it is advantageous to postpone tracing these pointers for as long as possible, giving these objects time to die and thus eliminating the need to trace the pointers at all.

During the first phase of collection, the generation is collected as if we were using a non-generational collector, with two exceptions. First, while traversing these objects, if a pointer to an object in an older generation is found, then the older generation is informed of the existence of this pointer, and traversal within the current generation proceeds with other pointers. Second, if a pointer into a younger generation is detected, this pointer is recorded in a special list called the Inter-Generational Pointer (IGP) list. This list is later used as part of the root set for the younger generation.[27] If, during this first phase, information is received from younger generations that a pointer points to an object in the current generation, this information is ignored.

During the second phase of collection, marks from younger generations are passed to the current generation and recorded as pointers that need to be traversed before collection can complete.

4.14.3 Object Advancement

During traversal of a generation, as each object is blackened, it can be promoted to the next older generation. In general, however, objects should not be promoted too quickly or else the older generations will fill with objects that die shortly after they are advanced. One solution to this problem is to associate a counter with each object, and only advance an object when it is blackened *and* its counter reaches some threshold value. In the current configuration of our collector, we allocate new objects black, so they spend at least one entire collection cycle in the youngest generation before being advanced.[28] This gives us some of the benefit of the

[27]Typically, the roots for a garbage collector are just the global- and stack-allocated variables. However, for a generational collector, it is important to conceptually consider all pointers from older generations into younger ones also to be roots.

[28]While it is not clear how many generations a collector should have, two generations appear to give good results because it gives many of the advantages of generational collection without too many repeated traversals of the objects [Wil88].

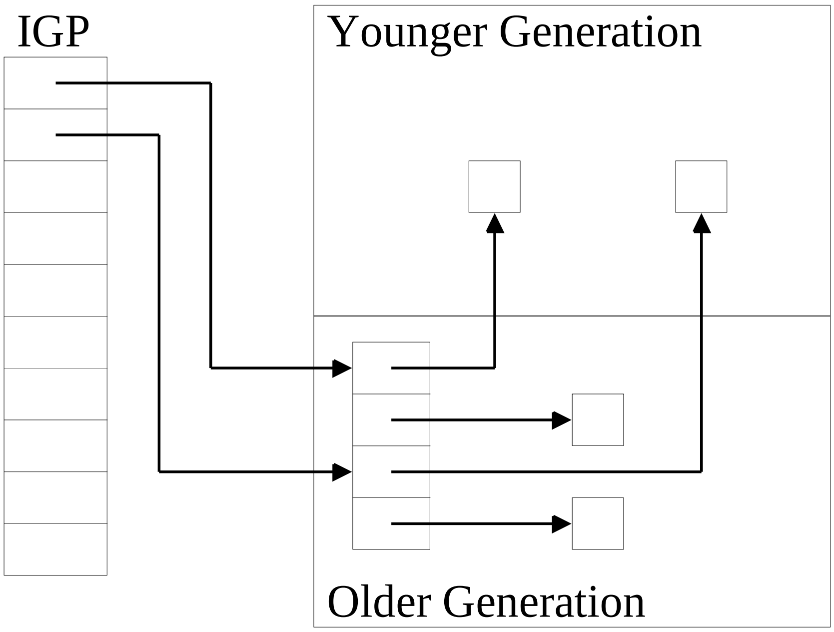

Figure 4.13: Example of the inter-generational pointer list

counter without the associated cost. Once an object reaches the oldest generation, it simply remains there until it dies or the program terminates.

The second consideration in object advancement is what color to advance the object. If we advance the object and color it white, then we run the risk of again advancing the object too soon. We also have the problem of increasing the number of objects in the older generation as we are trying to trace them, requiring the garbage collector to trace much faster to ensure that collection terminates.

We therefore choose to color objects black when we advance them. However, this too has its cost. These objects must be scanned for any pointers that point into younger generations so that these pointers can be recorded in the IGP list. We believe that this cost can be reduced enough to make it more advantageous to advance objects black than white.

4.14.4 Managing Inter-Generational Pointers

The Inter-Generational Pointer (IGP) list is implemented as a list of *pointers to* the pointers in old objects that point to young objects (see Figure 4.13). This is an important optimization. If a memory location in an object in the older generation is assigned many pointers to different younger objects, this extra level of indirection allows us to only trace the last object that is referenced by this memory location, and not all the earlier objects.

If we were to naively record each such pointer in its corresponding IGP list, it would monotonically increase, making it virtually impossible to follow all these pointers in any reasonable amount of time. We alleviate this problem by relying on the observation that objects can only spend a limited number of garbage collection cycles in the younger generation before they either die or are promoted. So, we actually manage a series of IGP lists between each older and younger generation. Each of these lists is associated with the garbage collection cycle of the younger generation in which the IGP was created. So, if an object can remain in the younger generation for, say, at most three complete garbage collection cycles, then there will be three IGP lists. At the end of each cycle, the oldest IGP list is thrown away, the second oldest IGP list becomes the oldest IGP list, and so on. The final complication is that the IGP list grows with the number of pointer assignments. This list can be kept to a bounded size by keying the rate of garbage collection to the rate of growth of this list, much like we do for the *incremental update* version of our write-barrier.

4.15 Generational Real-Time GC Status

Our real-time garbage collector is fully implemented and has been tested using our C++ smart-pointer interface. In addition, it has been integrated and tested with the "RScheme" Scheme system, and the Tower Eiffel compiler. Preliminary results are promising but still need considerably more work. In particular, we have not yet fully optimized the garbage collector's code for maximum performance. The results of running programs with our C++ smart pointer interface (where direct comparison with standard `new/delete` C++ code is possible) show a slowdown from 10% to 90%. We believe that most of this overhead is due to the lack of compiler cooperation for garbage collection in C++, and that when measurements are made for RScheme and Eiffel, these overheads will be considerably lower. However, it is clear that further tuning and measurements are needed.

Our real-time garbage collector has also been extended with the generational techniques we discussed in this chapter, and tested with the "RScheme" Scheme system, and the "Tower Eiffel" compiler.

4.16 Summary

In this chapter, we clarified the issues for real-time garbage collection. In addition, we developed 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 can be used to reason about the space and time tradeoffs between different incremental and real-time garbage collectors.

We also explored some novel generational garbage collection algorithms in an attempt to provide the benefit of generational techniques for many *soft* real-time applications. We proposed and implemented a design for a generational garbage collector that is more amenable to real-time applications than any other design that we know of. The key point of our design was 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.

Finally, we explored many different real-time garbage collection designs and considered the performance tradeoffs of each part of these designs. In particular, we implemented a non-copying implicit-reclamation collector which is fully configurable to a number of different write-barrier approaches.

Chapter 5

Conclusions and Future Work

In this dissertation, we studied many of the issues pertaining to memory allocation. We showed that for most programs, fragmentation is not a problem, provided that the memory allocator use reasonable implementations of well-known policies. In addition, we showed that these results have gone unknown largely because the predominating experimental methodology is fundamentally flawed. We developed a sound methodology and showed the importance of separating strategy, policy, and mechanism in allocator design.

Next, we studied the effects of memory allocator placement policy on the locality of programs, an area that has gone almost completely unstudied. We showed that even though there is little correlation between fragmentation and locality, the best placement policies in terms of fragmentation are also the best placement policies in terms of locality.

Finally, we explored and clarified the issues involved with garbage collection in general, and real-time garbage collection in particular. We developed a model for garbage collection that can be used to reason about the space and time tradeoffs for a number of different copying and non-copying garbage collector designs. We implemented a testbed garbage collector that can be configured for many different design possibilities and can then be used to measure these tradeoffs. Finally, we proposed and implemented a novel new generational garbage collection algorithm that maintains many of the advantages of traditional generational collection, but is much more suitable to real-time applications.

This research is not complete. We have, perhaps, introduced far more new questions than we have answered. In the remaining part of this chapter, we will discuss additional work that naturally follows from the work that we completed for this dissertation. We hope to address some of these issues ourselves, and encourage others to follow these lines of research.

Our fragmentation results are the most complete results in this dissertation. Although we have enough test programs to achieve statistical significance for our major conclusions, it would be far better to study additional programs from other application areas. In particular, we have recently received a trace of the X-windows system from Wolfram Gloger. Preliminary results from this trace show that for some allocation policies, using `mmap` to allocate very large objects *might* be necessary in order to achieve near-zero fragmentation.

Douglas Lea has implemented a memory allocator, which we studied in this dissertation, that is freely available and works quite well. We believe that this allocator can be improved in some small ways. We would like to attempt to implement a simpler mechanism of the same policy to produce an even faster implementation than he has provided.

Our locality results are less complete than our fragmentation results. We have shown that a memory allocator's placement choices can have a large effect on locality of reference. However, our work only addresses the most basic questions in this area. As was the case with our fragmentation experiments, the weakest part of this research is the small number of programs studied. We were only able to study six programs, and clearly, more programs are needed to completely validate these results.

One of the two measures of locality that we used in this research was cache miss rate. As we discussed in Section 3.1.2, there are many different cache designs in use. Each of these designs will interact with memory allocation policies in different ways, yielding different measures of locality. In particular, the average of the average object size for the programs studied in this research was 48.7 bytes, which for our experiments was the size of one and one-half cache lines. Thus, on average, only every other line had more than one object in it. As cache lines get longer,[1] false sharing of cache lines between multiple objects will become more of a problem, and we expect to see allocator placement policy variations become more important. Future work is needed to clarify how these designs interact with memory allocators by studying a larger set of cache designs.

Modern computers are quickly heading towards Symmetric Multi-Processing (SMP) designs. For SMP computers, cache miss rate is not a very good measure of locality. Simple cache miss rate measurements fail to capture two important locality characteristics: burstiness of misses, and false sharing. A simple miss rate measurement fails to measure contention for the memory bus. In other words, if two processes each have a miss rate of 1%, the system will have very different performance if both processes are missing the cache at the same time than if they are missing the cache at different times.

The second factor not measured by simple miss rate measurements is false sharing. This problem occurs when two objects are in the same cache line, but are being write-accessed by two different processors. In this case, system performance will suffer considerably as the lines are swapped between processor caches. Future work is to examine memory locality characteristics due to variations in memory allocation policy on SMP machines.

Finally, it is a fallacy to believe that improving cache miss rate by 50% will improve the total system performance by the same 50%. Cache miss rate is only a small component of total system performance. As system designs become more complicated (by using SMP and interleaved memories, for example), it becomes more and more important to measure the effect of memory allocation placement choices on the computer's Cycles Per Instruction (CPI). An interesting research project would be to use a microprocessor simulator to determine the effect of allocation policy on CPI for real computer systems.

Our garbage collection results are the most incomplete. We have implemented and tested our collector, but it has not yet been optimized. We intend to optimize our collector, and use it to measure the relative costs of the different configurations described by our model. These measurements should then make it possible to use this kind of collector in a wide variety of industrial strength applications.

[1]The MIPS R10000 has a 64 byte line size for its data cache.

Appendix A

Fragmentation Results for All Allocators

In this appendix, we present the raw data obtained from each of our fragmentation experiments. We present the data in four sections: we first present the raw data measured using the allocator implementations. We then present the percent fragmentation for each of these allocators for each trace. Then, we present the raw data for each of our allocators, *accounting for all overheads*. Finally, we present the percent fragmentation for each of these allocators for each trace.

A.1 Memory Used by Each Allocator for Each Trace

In this section, we present the amount of memory, in bytes, used by each of the allocators for each of the programs. No attempt has been made to remove any of the overheads of our implementations.

Allocator Name	Waste	Allocator Name	Waste
binary buddy	3998204	first fit LIFO deferred LIFO	3653632
best fit AO	3108864	first fit LIFO no footer	3358720
best fit AO 8K	3112960	first fit LIFO split-14	3530752
best fit AO deferred AO	3108864	first fit LIFO split-7	3526656
best fit AO deferred FIFO	3518464	half fit	3379200
best fit AO deferred LIFO	3383296	Lea 2.5.1	3420160
best fit AO no footer	2953216	Lea 2.5.1 no footer	3268608
best fit FIFO	3108864	Lea 2.6.1	2953216
best fit FIFO no footer	2953216	multi-fit max	3383296
best fit LIFO	3108864	multi-fit min	3244032
best fit LIFO deferred AO	3108864	next fit AO	3117056
best fit LIFO deferred FIFO	3518464	next fit AO 8K	3112960
best fit LIFO deferred LIFO	3383296	next fit AO deferred AO	3383296
best fit LIFO no footer	2953216	next fit AO deferred FIFO	3657728
best fit LIFO split-14	3121152	next fit AO deferred LIFO	3387392
best fit LIFO split-7	3117056	next fit AO no footer	2957312
double buddy 10K	3154344	next fit FIFO	3117056
double buddy 5K	3205544	next fit FIFO no footer	2957312
first fit AO	3112960	next fit LIFO	3657728
first fit AO 8K	3112960	next fit LIFO deferred LIFO	3522560
first fit AO deferred AO	3108864	next fit LIFO no footer	3366912
first fit AO deferred FIFO	3518464	next fit LIFO split-14	3534848
first fit AO deferred LIFO	3321856	next fit LIFO split-7	3530752
first fit AO no footer	2953216	next fit LIFO WPH	3518464
first fit FIFO	3117056	simple segregated 2^N	4755456
first fit FIFO no footer	2957312	simple segregated $3 * 2^N$	4349952
first fit LIFO	3518464		

Table A.1: Memory used by each allocator for GCC

Allocator Name	Waste	Allocator Name	Waste
binary buddy	410108	first fit LIFO def LIFO	315392
best fit AO	315392	first fit LIFO no footer	315392
best fit AO 8K	319488	first fit LIFO split-14	319488
best fit AO def AO	315392	first fit LIFO split-7	315392
best fit AO def FIFO	315392	half fit	315392
best fit AO def LIFO	315392	Lea 2.5.1	327680
best fit AO no footer	290816	Lea 2.5.1 no footer	294912
best fit FIFO	315392	Lea 2.6.1	315392
best fit FIFO no footer	290816	multi-fit max	315392
best fit LIFO	315392	multi-fit min	315392
best fit LIFO def AO	315392	next fit AO	315392
best fit LIFO def FIFO	315392	next fit AO 8K	319488
best fit LIFO def LIFO	315392	next fit AO def AO	315392
best fit LIFO no footer	290816	next fit AO def FIFO	315392
best fit LIFO split-14	323584	next fit AO def LIFO	315392
best fit LIFO split-7	315392	next fit AO no footer	290816
double buddy 10K	422312	next fit FIFO	315392
double buddy 5K	424360	next fit FIFO no footer	290816
first fit AO	315392	next fit LIFO	315392
first fit AO 8K	319488	next fit LIFO def LIFO	315392
first fit AO def AO	315392	next fit LIFO no footer	315392
first fit AO def FIFO	315392	next fit LIFO split-14	315392
first fit AO def LIFO	315392	next fit LIFO split-7	315392
first fit AO no footer	290816	next fit LIFO WPH	319488
first fit FIFO	315392	simple segregated 2^N	724992
first fit FIFO no footer	290816	simple segregated $3 * 2^N$	757760
first fit LIFO	315392		

Table A.2: Memory used by each allocator for Espresso

Allocator Name	Waste	Allocator Name	Waste
binary buddy	1802748	first fit LIFO def LIFO	1720320
best fit AO	1327104	first fit LIFO no footer	3473408
best fit AO 8K	1327104	first fit LIFO split-14	3620864
best fit AO def AO	1314816	first fit LIFO split-7	3645440
best fit AO def FIFO	1314816	half fit	1314816
best fit AO def LIFO	1314816	Lea 2.5.1	1728512
best fit AO no footer	1294336	Lea 2.5.1 no footer	1691648
best fit FIFO	1327104	Lea 2.6.1	1290240
best fit FIFO no footer	1290240	multi-fit max	1318912
best fit LIFO	1339392	multi-fit min	1318912
best fit LIFO def AO	1314816	next fit AO	1544192
best fit LIFO def FIFO	1314816	next fit AO 8K	1507328
best fit LIFO def LIFO	1314816	next fit AO def AO	1331200
best fit LIFO no footer	1290240	next fit AO def FIFO	1355776
best fit LIFO split-14	1335296	next fit AO def LIFO	1331200
best fit LIFO split-7	1339392	next fit AO no footer	1486848
double buddy 10K	1495464	next fit FIFO	1855488
double buddy 5K	1505704	next fit FIFO no footer	1794048
first fit AO	1335296	next fit LIFO	3870720
first fit AO 8K	1343488	next fit LIFO def LIFO	2002944
first fit AO def AO	1314816	next fit LIFO no footer	3637248
first fit AO def FIFO	1314816	next fit LIFO split-14	3895296
first fit AO def LIFO	1314816	next fit LIFO split-7	3887104
first fit AO no footer	1294336	next fit LIFO WPH	3854336
first fit FIFO	1363968	simple segregated 2^N	1597440
first fit FIFO no footer	1335296	simple segregated $3 * 2^N$	1458176
first fit LIFO	3624960		

Table A.3: Memory used by each allocator for Ghostscript

Allocator Name	Waste	Allocator Name	Waste
binary buddy	274940	first fit LIFO def LIFO	491520
best fit AO	294912	first fit LIFO no footer	331776
best fit AO 8K	294912	first fit LIFO split-14	364544
best fit AO def AO	286720	first fit LIFO split-7	360448
best fit AO def FIFO	286720	half fit	286720
best fit AO def LIFO	286720	Lea 2.5.1	307200
best fit AO no footer	253952	Lea 2.5.1 no footer	274432
best fit FIFO	294912	Lea 2.6.1	253952
best fit FIFO no footer	253952	multi-fit max	303104
best fit LIFO	294912	multi-fit min	299008
best fit LIFO def AO	286720	next fit AO	315392
best fit LIFO def FIFO	286720	next fit AO 8K	319488
best fit LIFO def LIFO	286720	next fit AO def AO	339968
best fit LIFO no footer	253952	next fit AO def FIFO	372736
best fit LIFO split-14	294912	next fit AO def LIFO	356352
best fit LIFO split-7	294912	next fit AO no footer	274432
double buddy 10K	260520	next fit FIFO	319488
double buddy 5K	257448	next fit FIFO no footer	286720
first fit AO	290816	next fit LIFO	364544
first fit AO 8K	294912	next fit LIFO def LIFO	446464
first fit AO def AO	286720	next fit LIFO no footer	315392
first fit AO def FIFO	286720	next fit LIFO split-14	356352
first fit AO def LIFO	286720	next fit LIFO split-7	352256
first fit AO no footer	249856	next fit LIFO WPH	319488
first fit FIFO	294912	simple segregated 2^N	294912
first fit FIFO no footer	253952	simple segregated $3 * 2^N$	270336
first fit LIFO	356352		

Table A.4: Memory used by each allocator for Grobner

Allocator Name	Waste	Allocator Name	Waste
binary buddy	4194812	first fit LIFO deferred LIFO	2105344
best fit AO	2105344	first fit LIFO no footer	2101248
best fit AO 8K	2105344	first fit LIFO split-14	2117632
best fit AO deferred AO	2105344	first fit LIFO split-7	2117632
best fit AO deferred FIFO	2105344	half fit	2105344
best fit AO deferred LIFO	2105344	Lea 2.5.1	2105344
best fit AO no footer	2101248	Lea 2.5.1 no footer	2101248
best fit FIFO	2105344	Lea 2.6.1	2101248
best fit FIFO no footer	2101248	multi-fit max	2105344
best fit LIFO	2105344	multi-fit min	2105344
best fit LIFO deferred AO	2105344	next fit AO	2105344
best fit LIFO deferred FIFO	2105344	next fit AO 8K	2105344
best fit LIFO deferred LIFO	2105344	next fit AO deferred AO	2105344
best fit LIFO no footer	2101248	next fit AO deferred FIFO	2105344
best fit LIFO split-14	2117632	next fit AO deferred LIFO	2105344
best fit LIFO split-7	2117632	next fit AO no footer	2101248
double buddy 10K	3156392	next fit FIFO	2105344
double buddy 5K	3151272	next fit FIFO no footer	2101248
first fit AO	2105344	next fit LIFO	2105344
first fit AO 8K	2105344	next fit LIFO deferred LIFO	2105344
first fit AO deferred AO	2105344	next fit LIFO no footer	2101248
first fit AO deferred FIFO	2105344	next fit LIFO split-14	2117632
first fit AO deferred LIFO	2105344	next fit LIFO split-7	2117632
first fit AO no footer	2101248	next fit LIFO WPH	2105344
first fit FIFO	2105344	simple segregated 2^N	2654208
first fit FIFO no footer	2101248	simple segregated $3 * 2^N$	2654208
first fit LIFO	2105344		

Table A.5: Memory used by each allocator for Hyper

Allocator Name	Waste	Allocator Name	Waste
binary buddy	664060	first fit LIFO deferred LIFO	561152
best fit AO	552960	first fit LIFO no footer	503808
best fit AO 8K	557056	first fit LIFO split-14	565248
best fit AO deferred AO	548864	first fit LIFO split-7	565248
best fit AO deferred FIFO	548864	half fit	548864
best fit AO deferred LIFO	548864	Lea 2.5.1	577536
best fit AO no footer	499712	Lea 2.5.1 no footer	516096
best fit FIFO	552960	Lea 2.6.1	499712
best fit FIFO no footer	499712	multi-fit max	548864
best fit LIFO	552960	multi-fit min	552960
best fit LIFO deferred AO	548864	next fit AO	577536
best fit LIFO deferred FIFO	548864	next fit AO 8K	581632
best fit LIFO deferred LIFO	548864	next fit AO deferred AO	557056
best fit LIFO no footer	499712	next fit AO deferred FIFO	561152
best fit LIFO split-14	552960	next fit AO deferred LIFO	561152
best fit LIFO split-7	552960	next fit AO no footer	516096
double buddy 10K	553384	next fit FIFO	593920
double buddy 5K	550312	next fit FIFO no footer	524288
first fit AO	552960	next fit LIFO	606208
first fit AO 8K	557056	next fit LIFO deferred LIFO	569344
first fit AO deferred AO	548864	next fit LIFO no footer	544768
first fit AO deferred FIFO	548864	next fit LIFO split-14	598016
first fit AO deferred LIFO	548864	next fit LIFO split-7	598016
first fit AO no footer	499712	next fit LIFO WPH	577536
first fit FIFO	552960	simple segregated 2^N	741376
first fit FIFO no footer	495616	simple segregated $3 * 2^N$	610304
first fit LIFO	561152		

Table A.6: Memory used by each allocator for P2C

Allocator Name	Waste	Allocator Name	Waste
binary buddy	131580	first fit LIFO deferred LIFO	94208
best fit AO	94208	first fit LIFO no footer	90112
best fit AO 8K	98304	first fit LIFO split-14	94208
best fit AO deferred AO	94208	first fit LIFO split-7	94208
best fit AO deferred FIFO	94208	half fit	94208
best fit AO deferred LIFO	94208	Lea 2.5.1	94208
best fit AO no footer	90112	Lea 2.5.1 no footer	90112
best fit FIFO	94208	Lea 2.6.1	90112
best fit FIFO no footer	90112	multi-fit max	94208
best fit LIFO	94208	multi-fit min	94208
best fit LIFO deferred AO	94208	next fit AO	94208
best fit LIFO deferred FIFO	94208	next fit AO 8K	98304
best fit LIFO deferred LIFO	94208	next fit AO deferred AO	94208
best fit LIFO no footer	90112	next fit AO deferred FIFO	94208
best fit LIFO split-14	94208	next fit AO deferred LIFO	94208
best fit LIFO split-7	94208	next fit AO no footer	90112
double buddy 10K	106920	next fit FIFO	94208
double buddy 5K	106920	next fit FIFO no footer	90112
first fit AO	94208	next fit LIFO	94208
first fit AO 8K	98304	next fit LIFO deferred LIFO	94208
first fit AO deferred AO	94208	next fit LIFO no footer	94208
first fit AO deferred FIFO	94208	next fit LIFO split-14	94208
first fit AO deferred LIFO	94208	next fit LIFO split-7	94208
first fit AO no footer	90112	next fit LIFO WPH	94208
first fit FIFO	94208	simple segregated 2^N	126976
first fit FIFO no footer	90112	simple segregated $3 * 2^N$	135168
first fit LIFO	94208		

Table A.7: Memory used by each allocator for Perl

Allocator Name	Waste	Allocator Name	Waste
binary buddy	2515452	first fit LIFO deferred LIFO	1884160
best fit AO	1884160	first fit LIFO no footer	1572864
best fit AO 8K	1884160	first fit LIFO split-14	1884160
best fit AO deferred AO	1884160	first fit LIFO split-7	1884160
best fit AO deferred FIFO	1884160	half fit	1884160
best fit AO deferred LIFO	1884160	Lea 2.5.1	1884160
best fit AO no footer	1572864	Lea 2.5.1 no footer	1572864
best fit FIFO	1884160	Lea 2.6.1	1572864
best fit FIFO no footer	1572864	multi-fit max	1884160
best fit LIFO	1884160	multi-fit min	1884160
best fit LIFO deferred AO	1884160	next fit AO	1884160
best fit LIFO deferred FIFO	1884160	next fit AO 8K	1884160
best fit LIFO deferred LIFO	1884160	next fit AO deferred AO	1884160
best fit LIFO no footer	1572864	next fit AO deferred FIFO	1884160
best fit LIFO split-14	1884160	next fit AO deferred LIFO	1884160
best fit LIFO split-7	1884160	next fit AO no footer	1572864
double buddy 10K	1894824	next fit FIFO	1884160
double buddy 5K	1889704	next fit FIFO no footer	1572864
first fit AO	1884160	next fit LIFO	1884160
first fit AO 8K	1884160	next fit LIFO deferred LIFO	1884160
first fit AO deferred AO	1884160	next fit LIFO no footer	1572864
first fit AO deferred FIFO	1884160	next fit LIFO split-14	1884160
first fit AO deferred LIFO	1884160	next fit LIFO split-7	1884160
first fit AO no footer	1572864	next fit LIFO WPH	1884160
first fit FIFO	1884160	simple segregated 2^N	2547712
first fit FIFO no footer	1572864	simple segregated $3 * 2^N$	1933312
first fit LIFO	1884160		

Table A.8: Memory used by each allocator for LRUsim

Allocator Name	Waste	Allocator Name	Waste
binary buddy	68.27%	first fit LIFO deferred LIFO	53.77%
best fit AO	30.84%	first fit LIFO no footer	41.36%
best fit AO 8K	31.01%	first fit LIFO split-14	48.60%
best fit AO deferred AO	30.84%	first fit LIFO split-7	48.42%
best fit AO deferred FIFO	48.08%	half fit	42.22%
best fit AO deferred LIFO	42.39%	Lea 2.5.1	43.94%
best fit AO no footer	24.29%	Lea 2.5.1 no footer	37.56%
best fit FIFO	30.84%	Lea 2.6.1	24.29%
best fit FIFO no footer	24.29%	multi-fit max	42.39%
best fit LIFO	30.84%	multi-fit min	36.53%
best fit LIFO deferred AO	30.84%	next fit AO	31.19%
best fit LIFO deferred FIFO	48.08%	next fit AO 8K	31.01%
best fit LIFO deferred LIFO	42.39%	next fit AO deferred AO	42.39%
best fit LIFO no footer	24.29%	next fit AO deferred FIFO	53.94%
best fit LIFO split-14	31.36%	next fit AO deferred LIFO	42.56%
best fit LIFO split-7	31.19%	next fit AO no footer	24.46%
double buddy 10K	32.76%	next fit FIFO	31.19%
double buddy 5K	34.91%	next fit FIFO no footer	24.46%
first fit AO	31.01%	next fit LIFO	53.94%
first fit AO 8K	31.01%	next fit LIFO deferred LIFO	48.25%
first fit AO deferred AO	30.84%	next fit LIFO no footer	41.70%
first fit AO deferred FIFO	48.08%	next fit LIFO split-14	48.77%
first fit AO deferred LIFO	39.81%	next fit LIFO split-7	48.60%
first fit AO no footer	24.29%	next fit LIFO WPH	48.08%
first fit FIFO	31.19%	simple segregated 2^N	100.14%
first fit FIFO no footer	24.46%	simple segregated $3 * 2^N$	83.07%
first fit LIFO	48.08%		

Table A.9: Percent fragmentation for GCC

A.2 Percent Fragmentation for Each Allocator for Each Trace

In this section, we present the percentage of fragmentation for each of the allocators for each of the programs. No attempt has been made to remove any of the overheads of our implementations. Percentage fragmentation is measured as the percent above the maximum amount of memory requested by each trace, as described in Sectionmalloc:fragmentation-definition.

Allocator Name	Waste	Allocator Name	Waste
binary buddy	52.09%	first fit LIFO def LIFO	16.96%
best fit AO	16.96%	first fit LIFO no footer	16.96%
best fit AO 8K	18.48%	first fit LIFO split-14	18.48%
best fit AO def AO	16.96%	first fit LIFO split-7	16.96%
best fit AO def FIFO	16.96%	half fit	16.96%
best fit AO def LIFO	16.96%	Lea 2.5.1	21.52%
best fit AO no footer	7.85%	Lea 2.5.1 no footer	9.37%
best fit FIFO	16.96%	Lea 2.6.1	16.96%
best fit FIFO no footer	7.85%	multi-fit max	16.96%
best fit LIFO	16.96%	multi-fit min	16.96%
best fit LIFO def AO	16.96%	next fit AO	16.96%
best fit LIFO def FIFO	16.96%	next fit AO 8K	18.48%
best fit LIFO def LIFO	16.96%	next fit AO def AO	16.96%
best fit LIFO no footer	7.85%	next fit AO def FIFO	16.96%
best fit LIFO split-14	20.00%	next fit AO def LIFO	16.96%
best fit LIFO split-7	16.96%	next fit AO no footer	7.85%
double buddy 10K	56.62%	next fit FIFO	16.96%
double buddy 5K	57.38%	next fit FIFO no footer	7.85%
first fit AO	16.96%	next fit LIFO	16.96%
first fit AO 8K	18.48%	next fit LIFO def LIFO	16.96%
first fit AO def AO	16.96%	next fit LIFO no footer	16.96%
first fit AO def FIFO	16.96%	next fit LIFO split-14	16.96%
first fit AO def LIFO	16.96%	next fit LIFO split-7	16.96%
first fit AO no footer	7.85%	next fit LIFO WPH	18.48%
first fit FIFO	16.96%	simple segregated 2^N	168.87%
first fit FIFO no footer	7.85%	simple segregated $3 * 2^N$	181.02%
first fit LIFO	16.96%		

Table A.10: Percent fragmentation for Espresso

Allocator Name	Waste	Allocator Name	Waste
binary buddy	58.57%	first fit LIFO def LIFO	51.32%
best fit AO	16.73%	first fit LIFO no footer	205.52%
best fit AO 8K	16.73%	first fit LIFO split-14	218.49%
best fit AO def AO	15.65%	first fit LIFO split-7	220.65%
best fit AO def FIFO	15.65%	half fit	15.65%
best fit AO def LIFO	15.65%	Lea 2.5.1	52.04%
best fit AO no footer	13.85%	Lea 2.5.1 no footer	48.80%
best fit FIFO	16.73%	Lea 2.6.1	13.49%
best fit FIFO no footer	13.49%	multi-fit max	16.01%
best fit LIFO	17.81%	multi-fit min	16.01%
best fit LIFO def AO	15.65%	next fit AO	35.83%
best fit LIFO def FIFO	15.65%	next fit AO 8K	32.58%
best fit LIFO def LIFO	15.65%	next fit AO def AO	17.09%
best fit LIFO no footer	13.49%	next fit AO def FIFO	19.25%
best fit LIFO split-14	17.45%	next fit AO def LIFO	17.09%
best fit LIFO split-7	17.81%	next fit AO no footer	30.78%
double buddy 10K	31.54%	next fit FIFO	63.21%
double buddy 5K	32.44%	next fit FIFO no footer	57.80%
first fit AO	17.45%	next fit LIFO	240.47%
first fit AO 8K	18.17%	next fit LIFO def LIFO	76.18%
first fit AO def AO	15.65%	next fit LIFO no footer	219.93%
first fit AO def FIFO	15.65%	next fit LIFO split-14	242.63%
first fit AO def LIFO	15.65%	next fit LIFO split-7	241.91%
first fit AO no footer	13.85%	next fit LIFO WPH	239.03%
first fit FIFO	19.97%	simple segregated 2^N	40.51%
first fit FIFO no footer	17.45%	simple segregated $3 * 2^N$	28.26%
first fit LIFO	218.85%		

Table A.11: Percent fragmentation for Ghostscript

Allocator Name	Waste	Allocator Name	Waste
binary buddy	85.10%	first fit LIFO def LIFO	230.91%
best fit AO	98.54%	first fit LIFO no footer	123.36%
best fit AO 8K	98.54%	first fit LIFO split-14	145.42%
best fit AO def AO	93.03%	first fit LIFO split-7	142.67%
best fit AO def FIFO	93.03%	half fit	93.03%
best fit AO def LIFO	93.03%	Lea 2.5.1	106.82%
best fit AO no footer	70.97%	Lea 2.5.1 no footer	84.76%
best fit FIFO	98.54%	Lea 2.6.1	70.97%
best fit FIFO no footer	70.97%	multi-fit max	104.06%
best fit LIFO	98.54%	multi-fit min	101.30%
best fit LIFO def AO	93.03%	next fit AO	112.33%
best fit LIFO def FIFO	93.03%	next fit AO 8K	115.09%
best fit LIFO def LIFO	93.03%	next fit AO def AO	128.88%
best fit LIFO no footer	70.97%	next fit AO def FIFO	150.94%
best fit LIFO split-14	98.54%	next fit AO def LIFO	139.91%
best fit LIFO split-7	98.54%	next fit AO no footer	84.76%
double buddy 10K	75.39%	next fit FIFO	115.09%
double buddy 5K	73.32%	next fit FIFO no footer	93.03%
first fit AO	95.79%	next fit LIFO	145.42%
first fit AO 8K	98.54%	next fit LIFO def LIFO	200.57%
first fit AO def AO	93.03%	next fit LIFO no footer	112.33%
first fit AO def FIFO	93.03%	next fit LIFO split-14	139.91%
first fit AO def LIFO	93.03%	next fit LIFO split-7	137.15%
first fit AO no footer	68.21%	next fit LIFO WPH	115.09%
first fit FIFO	98.54%	simple segregated 2^N	98.54%
first fit FIFO no footer	70.97%	simple segregated $3 * 2^N$	82.00%
first fit LIFO	139.91%		

Table A.12: Percent fragmentation for Grobner

Allocator Name	Waste	Allocator Name	Waste
binary buddy	99.95%	first fit LIFO deferred LIFO	0.35%
best fit AO	0.35%	first fit LIFO no footer	0.16%
best fit AO 8K	0.35%	first fit LIFO split-14	0.94%
best fit AO deferred AO	0.35%	first fit LIFO split-7	0.94%
best fit AO deferred FIFO	0.35%	half fit	0.35%
best fit AO deferred LIFO	0.35%	Lea 2.5.1	0.35%
best fit AO no footer	0.16%	Lea 2.5.1 no footer	0.16%
best fit FIFO	0.35%	Lea 2.6.1	0.16%
best fit FIFO no footer	0.16%	multi-fit max	0.35%
best fit LIFO	0.35%	multi-fit min	0.35%
best fit LIFO deferred AO	0.35%	next fit AO	0.35%
best fit LIFO deferred FIFO	0.35%	next fit AO 8K	0.35%
best fit LIFO deferred LIFO	0.35%	next fit AO deferred AO	0.35%
best fit LIFO no footer	0.16%	next fit AO deferred FIFO	0.35%
best fit LIFO split-14	0.94%	next fit AO deferred LIFO	0.35%
best fit LIFO split-7	0.94%	next fit AO no footer	0.16%
double buddy 10K	50.45%	next fit FIFO	0.35%
double buddy 5K	50.21%	next fit FIFO no footer	0.16%
first fit AO	0.35%	next fit LIFO	0.35%
first fit AO 8K	0.35%	next fit LIFO deferred LIFO	0.35%
first fit AO deferred AO	0.35%	next fit LIFO no footer	0.16%
first fit AO deferred FIFO	0.35%	next fit LIFO split-14	0.94%
first fit AO deferred LIFO	0.35%	next fit LIFO split-7	0.94%
first fit AO no footer	0.16%	next fit LIFO WPH	0.35%
first fit FIFO	0.35%	simple segregated 2^N	26.51%
first fit FIFO no footer	0.16%	simple segregated $3 * 2^N$	26.51%
first fit LIFO	0.35%		

Table A.13: Percent fragmentation for Hyper

Allocator Name	Waste	Allocator Name	Waste
binary buddy	65.01%	first fit LIFO deferred LIFO	39.44%
best fit AO	37.40%	first fit LIFO no footer	25.19%
best fit AO 8K	38.42%	first fit LIFO split-14	40.46%
best fit AO deferred AO	36.39%	first fit LIFO split-7	40.46%
best fit AO deferred FIFO	36.39%	half fit	36.39%
best fit AO deferred LIFO	36.39%	Lea 2.5.1	43.51%
best fit AO no footer	24.17%	Lea 2.5.1 no footer	28.24%
best fit FIFO	37.40%	Lea 2.6.1	24.17%
best fit FIFO no footer	24.17%	multi-fit max	36.39%
best fit LIFO	37.40%	multi-fit min	37.40%
best fit LIFO deferred AO	36.39%	next fit AO	43.51%
best fit LIFO deferred FIFO	36.39%	next fit AO 8K	44.53%
best fit LIFO deferred LIFO	36.39%	next fit AO deferred AO	38.42%
best fit LIFO no footer	24.17%	next fit AO deferred FIFO	39.44%
best fit LIFO split-14	37.40%	next fit AO deferred LIFO	39.44%
best fit LIFO split-7	37.40%	next fit AO no footer	28.24%
double buddy 10K	37.51%	next fit FIFO	47.58%
double buddy 5K	36.75%	next fit FIFO no footer	30.28%
first fit AO	37.40%	next fit LIFO	50.64%
first fit AO 8K	38.42%	next fit LIFO deferred LIFO	41.48%
first fit AO deferred AO	36.39%	next fit LIFO no footer	35.37%
first fit AO deferred FIFO	36.39%	next fit LIFO split-14	48.60%
first fit AO deferred LIFO	36.39%	next fit LIFO split-7	48.60%
first fit AO no footer	24.17%	next fit LIFO WPH	43.51%
first fit FIFO	37.40%	simple segregated 2^N	84.22%
first fit FIFO no footer	23.16%	simple segregated $3 * 2^N$	51.65%
first fit LIFO	39.44%		

Table A.14: Percent fragmentation for P2C

Allocator Name	Waste	Allocator Name	Waste
binary buddy	85.95%	first fit LIFO deferred LIFO	33.14%
best fit AO	33.14%	first fit LIFO no footer	27.35%
best fit AO 8K	38.92%	first fit LIFO split-14	33.14%
best fit AO deferred AO	33.14%	first fit LIFO split-7	33.14%
best fit AO deferred FIFO	33.14%	half fit	33.14%
best fit AO deferred LIFO	33.14%	Lea 2.5.1	33.14%
best fit AO no footer	27.35%	Lea 2.5.1 no footer	27.35%
best fit FIFO	33.14%	Lea 2.6.1	27.35%
best fit FIFO no footer	27.35%	multi-fit max	33.14%
best fit LIFO	33.14%	multi-fit min	33.14%
best fit LIFO deferred AO	33.14%	next fit AO	33.14%
best fit LIFO deferred FIFO	33.14%	next fit AO 8K	38.92%
best fit LIFO deferred LIFO	33.14%	next fit AO deferred AO	33.14%
best fit LIFO no footer	27.35%	next fit AO deferred FIFO	33.14%
best fit LIFO split-14	33.14%	next fit AO deferred LIFO	33.14%
best fit LIFO split-7	33.14%	next fit AO no footer	27.35%
double buddy 10K	51.10%	next fit FIFO	33.14%
double buddy 5K	51.10%	next fit FIFO no footer	27.35%
first fit AO	33.14%	next fit LIFO	33.14%
first fit AO 8K	38.92%	next fit LIFO deferred LIFO	33.14%
first fit AO deferred AO	33.14%	next fit LIFO no footer	33.14%
first fit AO deferred FIFO	33.14%	next fit LIFO split-14	33.14%
first fit AO deferred LIFO	33.14%	next fit LIFO split-7	33.14%
first fit AO no footer	27.35%	next fit LIFO WPH	33.14%
first fit FIFO	33.14%	simple segregated 2^N	79.44%
first fit FIFO no footer	27.35%	simple segregated $3 * 2^N$	91.02%
first fit LIFO	33.14%		

Table A.15: Percent fragmentation for Perl

Allocator Name	Waste	Allocator Name	Waste
binary buddy	77.95%	first fit LIFO deferred LIFO	33.29%
best fit AO	33.29%	first fit LIFO no footer	11.27%
best fit AO 8K	33.29%	first fit LIFO split-14	33.29%
best fit AO deferred AO	33.29%	first fit LIFO split-7	33.29%
best fit AO deferred FIFO	33.29%	half fit	33.29%
best fit AO deferred LIFO	33.29%	Lea 2.5.1	33.29%
best fit AO no footer	11.27%	Lea 2.5.1 no footer	11.27%
best fit FIFO	33.29%	Lea 2.6.1	11.27%
best fit FIFO no footer	11.27%	multi-fit max	33.29%
best fit LIFO	33.29%	multi-fit min	33.29%
best fit LIFO deferred AO	33.29%	next fit AO	33.29%
best fit LIFO deferred FIFO	33.29%	next fit AO 8K	33.29%
best fit LIFO deferred LIFO	33.29%	next fit AO deferred AO	33.29%
best fit LIFO no footer	11.27%	next fit AO deferred FIFO	33.29%
best fit LIFO split-14	33.29%	next fit AO deferred LIFO	33.29%
best fit LIFO split-7	33.29%	next fit AO no footer	11.27%
double buddy 10K	34.05%	next fit FIFO	33.29%
double buddy 5K	33.68%	next fit FIFO no footer	11.27%
first fit AO	33.29%	next fit LIFO	33.29%
first fit AO 8K	33.29%	next fit LIFO deferred LIFO	33.29%
first fit AO deferred AO	33.29%	next fit LIFO no footer	11.27%
first fit AO deferred FIFO	33.29%	next fit LIFO split-14	33.29%
first fit AO deferred LIFO	33.29%	next fit LIFO split-7	33.29%
first fit AO no footer	11.27%	next fit LIFO WPH	33.29%
first fit FIFO	33.29%	simple segregated 2^N	80.23%
first fit FIFO no footer	11.27%	simple segregated $3 * 2^N$	36.77%
first fit LIFO	33.29%		

Table A.16: Percent fragmentation for LRUsim

Allocator Name	Memory	Allocator Name	Memory
binary buddy	3187999	first fit LIFO deferred LIFO	2654208
best fit AO	2383872	first fit LIFO no footer	2940928
best fit AO 8K	2383872	first fit LIFO split-14	2809856
best fit AO deferred AO	2650112	first fit LIFO split-7	2809856
best fit AO deferred FIFO	2785280	half fit	2789376
best fit AO deferred LIFO	2789376	Lea 2.5.1	2924544
best fit AO no footer	2383872	Lea 2.5.1 no footer	2924544
best fit FIFO	2383872	Lea 2.6.1	2383872
best fit FIFO no footer	2383872	multi-fit max	2789376
best fit LIFO	2387968	multi-fit min	2789376
best fit LIFO deferred AO	2650112	next fit AO	2392064
best fit LIFO deferred FIFO	2785280	next fit AO 8K	2390016
best fit LIFO deferred LIFO	2789376	next fit AO deferred AO	2523136
best fit LIFO no footer	2387968	next fit AO deferred FIFO	2883584
best fit LIFO split-14	2404352	next fit AO deferred LIFO	2674688
best fit LIFO split-7	2404352	next fit AO no footer	2392064
double buddy 10K	3120927	next fit FIFO	2396160
double buddy 5K	3120799	next fit FIFO no footer	2396160
first fit AO	2387968	next fit LIFO	2875392
first fit AO 8K	2384384	next fit LIFO deferred LIFO	2736128
first fit AO deferred AO	2519040	next fit LIFO no footer	2875392
first fit AO deferred FIFO	2654208	next fit LIFO split-14	2818048
first fit AO deferred LIFO	2519040	next fit LIFO split-7	2818048
first fit AO no footer	2387968	next fit LIFO WPH	2793472
first fit FIFO	2387968	simple segregated 2^N	4739584
first fit FIFO no footer	2387968	simple segregated $3 * 2^N$	4749568
first fit LIFO	2940928		

Table A.17: Memory used by each allocator for the GCC program, accounting for all overheads

A.3 Memory Used by Each Allocator for Each Trace, Accounting for All Overheads

In this section, we present the amount of memory, in bytes, used by each of the allocators for each of the programs. All implementation overheads have been removed from these results.

Allocator Name	Memory	Allocator Name	Memory
binary buddy	393247	first fit LIFO def LIFO	294912
best fit AO	270336	first fit LIFO no footer	294912
best fit AO 8K	270336	first fit LIFO split-14	299008
best fit AO def AO	270336	first fit LIFO split-7	299008
best fit AO def FIFO	270336	half fit	294912
best fit AO def LIFO	274432	Lea 2.5.1	299008
best fit AO no footer	270336	Lea 2.5.1 no footer	299008
best fit FIFO	270336	Lea 2.6.1	270336
best fit FIFO no footer	270336	multi-fit max	270336
best fit LIFO	270336	multi-fit min	299008
best fit LIFO def AO	274432	next fit AO	274432
best fit LIFO def FIFO	274432	next fit AO 8K	272384
best fit LIFO def LIFO	270336	next fit AO def AO	278528
best fit LIFO no footer	270336	next fit AO def FIFO	294912
best fit LIFO split-14	274432	next fit AO def LIFO	311296
best fit LIFO split-7	270336	next fit AO no footer	274432
double buddy 10K	454943	next fit FIFO	270336
double buddy 5K	454815	next fit FIFO no footer	270336
first fit AO	270336	next fit LIFO	294912
first fit AO 8K	270336	next fit LIFO def LIFO	270336
first fit AO def AO	270336	next fit LIFO no footer	294912
first fit AO def FIFO	270336	next fit LIFO split-14	294912
first fit AO def LIFO	270336	next fit LIFO split-7	299008
first fit AO no footer	270336	next fit LIFO WPH	303104
first fit FIFO	270336	simple segregated 2^N	707584
first fit FIFO no footer	270336	simple segregated $3 * 2^N$	699648
first fit LIFO	294912		

Table A.18: Memory used by each allocator for the Espresso program, accounting for all overheads

Allocator Name	Memory	Allocator Name	Memory
binary buddy	1572895	first fit LIFO def LIFO	1642496
best fit AO	1175552	first fit LIFO no footer	3170304
best fit AO 8K	1170944	first fit LIFO split-14	3190784
best fit AO def AO	1155072	first fit LIFO split-7	3178496
best fit AO def FIFO	1163264	half fit	1155072
best fit AO def LIFO	1155072	Lea 2.5.1	2101248
best fit AO no footer	1175552	Lea 2.5.1 no footer	2101248
best fit FIFO	1171456	Lea 2.6.1	1175552
best fit FIFO no footer	1171456	multi-fit max	1171456
best fit LIFO	1175552	multi-fit min	1171456
best fit LIFO def AO	1155072	next fit AO	1327104
best fit LIFO def FIFO	1155072	next fit AO 8K	1349120
best fit LIFO def LIFO	1155072	next fit AO def AO	1171456
best fit LIFO no footer	1175552	next fit AO def FIFO	1175552
best fit LIFO split-14	1179648	next fit AO def LIFO	1171456
best fit LIFO split-7	1171456	next fit AO no footer	1327104
double buddy 10K	1372959	next fit FIFO	1757184
double buddy 5K	1373087	next fit FIFO no footer	1757184
first fit AO	1175552	next fit LIFO	3411968
first fit AO 8K	1168384	next fit LIFO def LIFO	1695744
first fit AO def AO	1155072	next fit LIFO no footer	3411968
first fit AO def FIFO	1163264	next fit LIFO split-14	3403776
first fit AO def LIFO	1155072	next fit LIFO split-7	3407872
first fit AO no footer	1175552	next fit LIFO WPH	3371008
first fit FIFO	1220608	simple segregated 2^N	1579776
first fit FIFO no footer	1220608	simple segregated $3 * 2^N$	1505536
first fit LIFO	3170304		

Table A.19: Memory used by each allocator for the Ghostscript program, accounting for all overheads

Allocator Name	Memory	Allocator Name	Memory
binary buddy	203295	first fit LIFO def LIFO	413696
best fit AO	151552	first fit LIFO no footer	241664
best fit AO 8K	150528	first fit LIFO split-14	266240
best fit AO def AO	155648	first fit LIFO split-7	249856
best fit AO def FIFO	159744	half fit	159744
best fit AO def LIFO	155648	Lea 2.5.1	184320
best fit AO no footer	151552	Lea 2.5.1 no footer	184320
best fit FIFO	151552	Lea 2.6.1	151552
best fit FIFO no footer	151552	multi-fit max	163840
best fit LIFO	151552	multi-fit min	159744
best fit LIFO def AO	155648	next fit AO	188416
best fit LIFO def FIFO	155648	next fit AO 8K	168448
best fit LIFO def LIFO	155648	next fit AO def AO	303104
best fit LIFO no footer	151552	next fit AO def FIFO	323584
best fit LIFO split-14	155648	next fit AO def LIFO	307200
best fit LIFO split-7	151552	next fit AO no footer	188416
double buddy 10K	166815	next fit FIFO	253952
double buddy 5K	166815	next fit FIFO no footer	253952
first fit AO	151552	next fit LIFO	225280
first fit AO 8K	150016	next fit LIFO def LIFO	376832
first fit AO def AO	155648	next fit LIFO no footer	225280
first fit AO def FIFO	155648	next fit LIFO split-14	241664
first fit AO def LIFO	155648	next fit LIFO split-7	233472
first fit AO no footer	151552	next fit LIFO WPH	196608
first fit FIFO	155648	simple segregated 2^N	233728
first fit FIFO no footer	155648	simple segregated $3 * 2^N$	209408
first fit LIFO	241664		

Table A.20: Memory used by each allocator for the Grobner program, accounting for all overheads

Allocator Name	Memory	Allocator Name	Memory
binary buddy	4194335	first fit LIFO deferred LIFO	2101248
best fit AO	2101248	first fit LIFO no footer	2101248
best fit AO 8K	2098176	first fit LIFO split-14	2113536
best fit AO deferred AO	2101248	first fit LIFO split-7	2113536
best fit AO deferred FIFO	2101248	half fit	2101248
best fit AO deferred LIFO	2101248	Lea 2.5.1	2101248
best fit AO no footer	2101248	Lea 2.5.1 no footer	2101248
best fit FIFO	2101248	Lea 2.6.1	2101248
best fit FIFO no footer	2101248	multi-fit max	2101248
best fit LIFO	2101248	multi-fit min	2101248
best fit LIFO deferred AO	2101248	next fit AO	2101248
best fit LIFO deferred FIFO	2101248	next fit AO 8K	2099200
best fit LIFO deferred LIFO	2101248	next fit AO deferred AO	2101248
best fit LIFO no footer	2101248	next fit AO deferred FIFO	2101248
best fit LIFO split-14	2113536	next fit AO deferred LIFO	2101248
best fit LIFO split-7	2113536	next fit AO no footer	2101248
double buddy 10K	3146399	next fit FIFO	2101248
double buddy 5K	3146079	next fit FIFO no footer	2101248
first fit AO	2101248	next fit LIFO	2101248
first fit AO 8K	2098176	next fit LIFO deferred LIFO	2101248
first fit AO deferred AO	2101248	next fit LIFO no footer	2101248
first fit AO deferred FIFO	2101248	next fit LIFO split-14	2113536
first fit AO deferred LIFO	2101248	next fit LIFO split-7	2113536
first fit AO no footer	2101248	next fit LIFO WPH	2101248
first fit FIFO	2101248	simple segregated 2^N	2642688
first fit FIFO no footer	2101248	simple segregated $3 * 2^N$	2642688
first fit LIFO	2101248		

Table A.21: Memory used by each allocator for the Hyper program, accounting for all overheads

Allocator Name	Memory	Allocator Name	Memory
binary buddy	623647	first fit LIFO deferred LIFO	425984
best fit AO	409600	first fit LIFO no footer	421888
best fit AO 8K	406528	first fit LIFO split-14	421888
best fit AO deferred AO	409600	first fit LIFO split-7	421888
best fit AO deferred FIFO	409600	half fit	409600
best fit AO deferred LIFO	409600	Lea 2.5.1	409600
best fit AO no footer	409600	Lea 2.5.1 no footer	409600
best fit FIFO	409600	Lea 2.6.1	409600
best fit FIFO no footer	409600	multi-fit max	409600
best fit LIFO	409600	multi-fit min	409600
best fit LIFO deferred AO	409600	next fit AO	434176
best fit LIFO deferred FIFO	409600	next fit AO 8K	431104
best fit LIFO deferred LIFO	409600	next fit AO deferred AO	425984
best fit LIFO no footer	409600	next fit AO deferred FIFO	425984
best fit LIFO split-14	409600	next fit AO deferred LIFO	434176
best fit LIFO split-7	409600	next fit AO no footer	434176
double buddy 10K	533279	next fit FIFO	442368
double buddy 5K	532959	next fit FIFO no footer	442368
first fit AO	409600	next fit LIFO	462848
first fit AO 8K	407040	next fit LIFO deferred LIFO	442368
first fit AO deferred AO	409600	next fit LIFO no footer	462848
first fit AO deferred FIFO	409600	next fit LIFO split-14	458752
first fit AO deferred LIFO	409600	next fit LIFO split-7	462848
first fit AO no footer	409600	next fit LIFO WPH	434176
first fit FIFO	409600	simple segregated 2^N	702208
first fit FIFO no footer	409600	simple segregated $3 * 2^N$	628480
first fit LIFO	421888		

Table A.22: Memory used by each allocator for the P2C program, accounting for all overheads

Allocator Name	Memory	Allocator Name	Memory
binary buddy	98335	first fit LIFO deferred LIFO	77824
best fit AO	77824	first fit LIFO no footer	77824
best fit AO 8K	71168	first fit LIFO split-14	77824
best fit AO deferred AO	77824	first fit LIFO split-7	77824
best fit AO deferred FIFO	77824	half fit	77824
best fit AO deferred LIFO	77824	Lea 2.5.1	77824
best fit AO no footer	77824	Lea 2.5.1 no footer	77824
best fit FIFO	77824	Lea 2.6.1	77824
best fit FIFO no footer	77824	multi-fit max	77824
best fit LIFO	77824	multi-fit min	77824
best fit LIFO deferred AO	77824	next fit AO	77824
best fit LIFO deferred FIFO	77824	next fit AO 8K	73216
best fit LIFO deferred LIFO	77824	next fit AO deferred AO	77824
best fit LIFO no footer	77824	next fit AO deferred FIFO	77824
best fit LIFO split-14	77824	next fit AO deferred LIFO	77824
best fit LIFO split-7	77824	next fit AO no footer	77824
double buddy 10K	88351	next fit FIFO	77824
double buddy 5K	88351	next fit FIFO no footer	77824
first fit AO	77824	next fit LIFO	77824
first fit AO 8K	71168	next fit LIFO deferred LIFO	77824
first fit AO deferred AO	77824	next fit LIFO no footer	77824
first fit AO deferred FIFO	77824	next fit LIFO split-14	77824
first fit AO deferred LIFO	77824	next fit LIFO split-7	77824
first fit AO no footer	77824	next fit LIFO WPH	77824
first fit FIFO	77824	simple segregated 2^N	107008
first fit FIFO no footer	77824	simple segregated $3 * 2^N$	94720
first fit LIFO	77824		

Table A.23: Memory used by each allocator for the Perl program, accounting for all overheads

Allocator Name	Memory	Allocator Name	Memory
binary buddy	2512415	first fit LIFO deferred LIFO	1417216
best fit AO	1417216	first fit LIFO no footer	1417216
best fit AO 8K	1414656	first fit LIFO split-14	1417216
best fit AO deferred AO	1417216	first fit LIFO split-7	1417216
best fit AO deferred FIFO	1417216	half fit	1417216
best fit AO deferred LIFO	1417216	Lea 2.5.1	1417216
best fit AO no footer	1417216	Lea 2.5.1 no footer	1417216
best fit FIFO	1417216	Lea 2.6.1	1417216
best fit FIFO no footer	1417216	multi-fit max	1417216
best fit LIFO	1417216	multi-fit min	1417216
best fit LIFO deferred AO	1417216	next fit AO	1417216
best fit LIFO deferred FIFO	1417216	next fit AO 8K	1414656
best fit LIFO deferred LIFO	1417216	next fit AO deferred AO	1417216
best fit LIFO no footer	1417216	next fit AO deferred FIFO	1417216
best fit LIFO split-14	1417216	next fit AO deferred LIFO	1417216
best fit LIFO split-7	1417216	next fit AO no footer	1417216
double buddy 10K	1889183	next fit FIFO	1417216
double buddy 5K	1889311	next fit FIFO no footer	1417216
first fit AO	1417216	next fit LIFO	1417216
first fit AO 8K	1414656	next fit LIFO deferred LIFO	1417216
first fit AO deferred AO	1417216	next fit LIFO no footer	1417216
first fit AO deferred FIFO	1417216	next fit LIFO split-14	1417216
first fit AO deferred LIFO	1417216	next fit LIFO split-7	1417216
first fit AO no footer	1417216	next fit LIFO WPH	1417216
first fit FIFO	1417216	simple segregated 2^N	2530048
first fit FIFO no footer	1417216	simple segregated $3 * 2^N$	2024704
first fit LIFO	1417216		

Table A.24: Memory used by each allocator for the LRUsim program, accounting for all overheads

Allocator Name	% frag	Allocator Name	% frag
binary buddy	34.17%	first fit LIFO deferred LIFO	11.71%
best fit AO	0.33%	first fit LIFO no footer	23.77%
best fit AO 8K	0.33%	first fit LIFO split-14	18.26%
best fit AO deferred AO	11.53%	first fit LIFO split-7	18.26%
best fit AO deferred FIFO	17.22%	half fit	17.40%
best fit AO deferred LIFO	17.40%	Lea 2.5.1	23.08%
best fit AO no footer	0.33%	Lea 2.5.1 no footer	23.08%
best fit FIFO	0.33%	Lea 2.6.1	0.33%
best fit FIFO no footer	0.33%	multi-fit max	17.40%
best fit LIFO	0.50%	multi-fit min	17.40%
best fit LIFO deferred AO	11.53%	next fit AO	0.67%
best fit LIFO deferred FIFO	17.22%	next fit AO 8K	0.59%
best fit LIFO deferred LIFO	17.40%	next fit AO deferred AO	6.19%
best fit LIFO no footer	0.50%	next fit AO deferred FIFO	21.36%
best fit LIFO split-14	1.19%	next fit AO deferred LIFO	12.57%
best fit LIFO split-7	1.19%	next fit AO no footer	0.67%
double buddy 10K	31.35%	next fit FIFO	0.85%
double buddy 5K	31.34%	next fit FIFO no footer	0.85%
first fit AO	0.50%	next fit LIFO	21.02%
first fit AO 8K	0.35%	next fit LIFO deferred LIFO	15.15%
first fit AO deferred AO	6.02%	next fit LIFO no footer	21.02%
first fit AO deferred FIFO	11.71%	next fit LIFO split-14	18.60%
first fit AO deferred LIFO	6.02%	next fit LIFO split-7	18.60%
first fit AO no footer	0.50%	next fit LIFO WPH	17.57%
first fit FIFO	0.50%	simple segregated 2^N	99.47%
first fit FIFO no footer	0.50%	simple segregated $3 * 2^N$	99.89%
first fit LIFO	23.77%		

Table A.25: Percent fragmentation for each allocator for the GCC program, accounting for all overheads

A.4 Percent Actual Fragmentation for Each Allocator for Each Trace – All Overheads Removed

In this section, we present the percentage of actual fragmentation for each of the allocators for each of the programs. All implementation overheads have been removed from these results. Percentage actual fragmentation is computed as described in Section 2.9.

Allocator Name	% frag	Allocator Name	% frag
binary buddy	45.84%	first fit LIFO def LIFO	9.37%
best fit AO	0.26%	first fit LIFO no footer	9.37%
best fit AO 8K	0.26%	first fit LIFO split-14	10.89%
best fit AO def AO	0.26%	first fit LIFO split-7	10.89%
best fit AO def FIFO	0.26%	half fit	9.37%
best fit AO def LIFO	1.77%	Lea 2.5.1	10.89%
best fit AO no footer	0.26%	Lea 2.5.1 no footer	10.89%
best fit FIFO	0.26%	Lea 2.6.1	0.26%
best fit FIFO no footer	0.26%	multi-fit max	0.26%
best fit LIFO	0.26%	multi-fit min	10.89%
best fit LIFO def AO	1.77%	next fit AO	1.77%
best fit LIFO def FIFO	1.77%	next fit AO 8K	1.02%
best fit LIFO def LIFO	0.26%	next fit AO def AO	3.29%
best fit LIFO no footer	0.26%	next fit AO def FIFO	9.37%
best fit LIFO split-14	1.77%	next fit AO def LIFO	15.45%
best fit LIFO split-7	0.26%	next fit AO no footer	1.77%
double buddy 10K	68.72%	next fit FIFO	0.26%
double buddy 5K	68.67%	next fit FIFO no footer	0.26%
first fit AO	0.26%	next fit LIFO	9.37%
first fit AO 8K	0.26%	next fit LIFO def LIFO	0.26%
first fit AO def AO	0.26%	next fit LIFO no footer	9.37%
first fit AO def FIFO	0.26%	next fit LIFO split-14	9.37%
first fit AO def LIFO	0.26%	next fit LIFO split-7	10.89%
first fit AO no footer	0.26%	next fit LIFO WPH	12.41%
first fit FIFO	0.26%	simple segregated 2^N	162.41%
first fit FIFO no footer	0.26%	simple segregated $3 * 2^N$	159.47%
first fit LIFO	9.37%		

Table A.26: Percent fragmentation for each allocator for the Espresso program, accounting for all overheads

Allocator Name	% frag	Allocator Name	% frag
binary buddy	38.35%	first fit LIFO def LIFO	44.47%
best fit AO	3.40%	first fit LIFO no footer	178.86%
best fit AO 8K	3.00%	first fit LIFO split-14	180.66%
best fit AO def AO	1.60%	first fit LIFO split-7	179.58%
best fit AO def FIFO	2.32%	half fit	1.60%
best fit AO def LIFO	1.60%	Lea 2.5.1	84.82%
best fit AO no footer	3.40%	Lea 2.5.1 no footer	84.82%
best fit FIFO	3.04%	Lea 2.6.1	3.40%
best fit FIFO no footer	3.04%	multi-fit max	3.04%
best fit LIFO	3.40%	multi-fit min	3.04%
best fit LIFO def AO	1.60%	next fit AO	16.73%
best fit LIFO def FIFO	1.60%	next fit AO 8K	18.67%
best fit LIFO def LIFO	1.60%	next fit AO def AO	3.04%
best fit LIFO no footer	3.40%	next fit AO def FIFO	3.40%
best fit LIFO split-14	3.76%	next fit AO def LIFO	3.04%
best fit LIFO split-7	3.04%	next fit AO no footer	16.73%
double buddy 10K	20.76%	next fit FIFO	54.56%
double buddy 5K	20.78%	next fit FIFO no footer	54.56%
first fit AO	3.40%	next fit LIFO	200.11%
first fit AO 8K	2.77%	next fit LIFO def LIFO	49.16%
first fit AO def AO	1.60%	next fit LIFO no footer	200.11%
first fit AO def FIFO	2.32%	next fit LIFO split-14	199.39%
first fit AO def LIFO	1.60%	next fit LIFO split-7	199.75%
first fit AO no footer	3.40%	next fit LIFO WPH	196.51%
first fit FIFO	7.36%	simple segregated 2^N	38.96%
first fit FIFO no footer	7.36%	simple segregated $3 * 2^N$	32.43%
first fit LIFO	178.86%		

Table A.27: Percent fragmentation for each allocator for the Ghostscript program, accounting for all overheads

Allocator Name	% frag	Allocator Name	% frag
binary buddy	36.86%	first fit LIFO def LIFO	178.51%
best fit AO	2.03%	first fit LIFO no footer	62.70%
best fit AO 8K	1.34%	first fit LIFO split-14	79.24%
best fit AO def AO	4.79%	first fit LIFO split-7	68.21%
best fit AO def FIFO	7.54%	half fit	7.54%
best fit AO def LIFO	4.79%	Lea 2.5.1	24.09%
best fit AO no footer	2.03%	Lea 2.5.1 no footer	24.09%
best fit FIFO	2.03%	Lea 2.6.1	2.03%
best fit FIFO no footer	2.03%	multi-fit max	10.30%
best fit LIFO	2.03%	multi-fit min	7.54%
best fit LIFO def AO	4.79%	next fit AO	26.85%
best fit LIFO def FIFO	4.79%	next fit AO 8K	13.40%
best fit LIFO def LIFO	4.79%	next fit AO def AO	104.06%
best fit LIFO no footer	2.03%	next fit AO def FIFO	117.85%
best fit LIFO split-14	4.79%	next fit AO def LIFO	106.82%
best fit LIFO split-7	2.03%	next fit AO no footer	26.85%
double buddy 10K	12.31%	next fit FIFO	70.97%
double buddy 5K	12.31%	next fit FIFO no footer	70.97%
first fit AO	2.03%	next fit LIFO	51.67%
first fit AO 8K	1.00%	next fit LIFO def LIFO	153.70%
first fit AO def AO	4.79%	next fit LIFO no footer	51.67%
first fit AO def FIFO	4.79%	next fit LIFO split-14	62.70%
first fit AO def LIFO	4.79%	next fit LIFO split-7	57.18%
first fit AO no footer	2.03%	next fit LIFO WPH	32.36%
first fit FIFO	4.79%	simple segregated 2^N	57.35%
first fit FIFO no footer	4.79%	simple segregated $3 * 2^N$	40.98%
first fit LIFO	62.70%		

Table A.28: Percent fragmentation for each allocator for the Grobner program, accounting for all overheads

Allocator Name	% frag	Allocator Name	% frag
binary buddy	99.92%	first fit LIFO deferred LIFO	0.16%
best fit AO	0.16%	first fit LIFO no footer	0.16%
best fit AO 8K	0.01%	first fit LIFO split-14	0.74%
best fit AO deferred AO	0.16%	first fit LIFO split-7	0.74%
best fit AO deferred FIFO	0.16%	half fit	0.16%
best fit AO deferred LIFO	0.16%	Lea 2.5.1	0.16%
best fit AO no footer	0.16%	Lea 2.5.1 no footer	0.16%
best fit FIFO	0.16%	Lea 2.6.1	0.16%
best fit FIFO no footer	0.16%	multi-fit max	0.16%
best fit LIFO	0.16%	multi-fit min	0.16%
best fit LIFO deferred AO	0.16%	next fit AO	0.16%
best fit LIFO deferred FIFO	0.16%	next fit AO 8K	0.06%
best fit LIFO deferred LIFO	0.16%	next fit AO deferred AO	0.16%
best fit LIFO no footer	0.16%	next fit AO deferred FIFO	0.16%
best fit LIFO split-14	0.74%	next fit AO deferred LIFO	0.16%
best fit LIFO split-7	0.74%	next fit AO no footer	0.16%
double buddy 10K	49.97%	next fit FIFO	0.16%
double buddy 5K	49.96%	next fit FIFO no footer	0.16%
first fit AO	0.16%	next fit LIFO	0.16%
first fit AO 8K	0.01%	next fit LIFO deferred LIFO	0.16%
first fit AO deferred AO	0.16%	next fit LIFO no footer	0.16%
first fit AO deferred FIFO	0.16%	next fit LIFO split-14	0.74%
first fit AO deferred LIFO	0.16%	next fit LIFO split-7	0.74%
first fit AO no footer	0.16%	next fit LIFO WPH	0.16%
first fit FIFO	0.16%	simple segregated 2^N	25.96%
first fit FIFO no footer	0.16%	simple segregated $3 * 2^N$	25.96%
first fit LIFO	0.16%		

Table A.29: Percent fragmentation for each allocator for the Hyper program, accounting for all overheads

Allocator Name	% frag	Allocator Name	% frag
binary buddy	54.97%	first fit LIFO deferred LIFO	5.85%
best fit AO	1.78%	first fit LIFO no footer	4.83%
best fit AO 8K	1.02%	first fit LIFO split-14	4.83%
best fit AO deferred AO	1.78%	first fit LIFO split-7	4.83%
best fit AO deferred FIFO	1.78%	half fit	1.78%
best fit AO deferred LIFO	1.78%	Lea 2.5.1	1.78%
best fit AO no footer	1.78%	Lea 2.5.1 no footer	1.78%
best fit FIFO	1.78%	Lea 2.6.1	1.78%
best fit FIFO no footer	1.78%	multi-fit max	1.78%
best fit LIFO	1.78%	multi-fit min	1.78%
best fit LIFO deferred AO	1.78%	next fit AO	7.89%
best fit LIFO deferred FIFO	1.78%	next fit AO 8K	7.12%
best fit LIFO deferred LIFO	1.78%	next fit AO deferred AO	5.85%
best fit LIFO no footer	1.78%	next fit AO deferred FIFO	5.85%
best fit LIFO split-14	1.78%	next fit AO deferred LIFO	7.89%
best fit LIFO split-7	1.78%	next fit AO no footer	7.89%
double buddy 10K	32.51%	next fit FIFO	9.92%
double buddy 5K	32.43%	next fit FIFO no footer	9.92%
first fit AO	1.78%	next fit LIFO	15.01%
first fit AO 8K	1.15%	next fit LIFO deferred LIFO	9.92%
first fit AO deferred AO	1.78%	next fit LIFO no footer	15.01%
first fit AO deferred FIFO	1.78%	next fit LIFO split-14	13.99%
first fit AO deferred LIFO	1.78%	next fit LIFO split-7	15.01%
first fit AO no footer	1.78%	next fit LIFO WPH	7.89%
first fit FIFO	1.78%	simple segregated 2^N	74.49%
first fit FIFO no footer	1.78%	simple segregated $3 * 2^N$	56.17%
first fit LIFO	4.83%		

Table A.30: Percent fragmentation for each allocator for the P2C program, accounting for all overheads

Allocator Name	% frag	Allocator Name	% frag
binary buddy	38.97%	first fit LIFO deferred LIFO	9.98%
best fit AO	9.98%	first fit LIFO no footer	9.98%
best fit AO 8K	0.58%	first fit LIFO split-14	9.98%
best fit AO deferred AO	9.98%	first fit LIFO split-7	9.98%
best fit AO deferred FIFO	9.98%	half fit	9.98%
best fit AO deferred LIFO	9.98%	Lea 2.5.1	9.98%
best fit AO no footer	9.98%	Lea 2.5.1 no footer	9.98%
best fit FIFO	9.98%	Lea 2.6.1	9.98%
best fit FIFO no footer	9.98%	multi-fit max	9.98%
best fit LIFO	9.98%	multi-fit min	9.98%
best fit LIFO deferred AO	9.98%	next fit AO	9.98%
best fit LIFO deferred FIFO	9.98%	next fit AO 8K	3.47%
best fit LIFO deferred LIFO	9.98%	next fit AO deferred AO	9.98%
best fit LIFO no footer	9.98%	next fit AO deferred FIFO	9.98%
best fit LIFO split-14	9.98%	next fit AO deferred LIFO	9.98%
best fit LIFO split-7	9.98%	next fit AO no footer	9.98%
double buddy 10K	24.86%	next fit FIFO	9.98%
double buddy 5K	24.86%	next fit FIFO no footer	9.98%
first fit AO	9.98%	next fit LIFO	9.98%
first fit AO 8K	0.58%	next fit LIFO deferred LIFO	9.98%
first fit AO deferred AO	9.98%	next fit LIFO no footer	9.98%
first fit AO deferred FIFO	9.98%	next fit LIFO split-14	9.98%
first fit AO deferred LIFO	9.98%	next fit LIFO split-7	9.98%
first fit AO no footer	9.98%	next fit LIFO WPH	9.98%
first fit FIFO	9.98%	simple segregated 2^N	51.22%
first fit FIFO no footer	9.98%	simple segregated $3 * 2^N$	33.86%
first fit LIFO	9.98%		

Table A.31: Percent fragmentation for each allocator for the Perl program, accounting for all overheads

Allocator Name	% frag	Allocator Name	% frag
binary buddy	77.74%	first fit LIFO deferred LIFO	0.26%
best fit AO	0.26%	first fit LIFO no footer	0.26%
best fit AO 8K	0.08%	first fit LIFO split-14	0.26%
best fit AO deferred AO	0.26%	first fit LIFO split-7	0.26%
best fit AO deferred FIFO	0.26%	half fit	0.26%
best fit AO deferred LIFO	0.26%	Lea 2.5.1	0.26%
best fit AO no footer	0.26%	Lea 2.5.1 no footer	0.26%
best fit FIFO	0.26%	Lea 2.6.1	0.26%
best fit FIFO no footer	0.26%	multi-fit max	0.26%
best fit LIFO	0.26%	multi-fit min	0.26%
best fit LIFO deferred AO	0.26%	next fit AO	0.26%
best fit LIFO deferred FIFO	0.26%	next fit AO 8K	0.08%
best fit LIFO deferred LIFO	0.26%	next fit AO deferred AO	0.26%
best fit LIFO no footer	0.26%	next fit AO deferred FIFO	0.26%
best fit LIFO split-14	0.26%	next fit AO deferred LIFO	0.26%
best fit LIFO split-7	0.26%	next fit AO no footer	0.26%
double buddy 10K	33.65%	next fit FIFO	0.26%
double buddy 5K	33.66%	next fit FIFO no footer	0.26%
first fit AO	0.26%	next fit LIFO	0.26%
first fit AO 8K	0.08%	next fit LIFO deferred LIFO	0.26%
first fit AO deferred AO	0.26%	next fit LIFO no footer	0.26%
first fit AO deferred FIFO	0.26%	next fit LIFO split-14	0.26%
first fit AO deferred LIFO	0.26%	next fit LIFO split-7	0.26%
first fit AO no footer	0.26%	next fit LIFO WPH	0.26%
first fit FIFO	0.26%	simple segregated 2^N	78.98%
first fit FIFO no footer	0.26%	simple segregated $3 * 2^N$	43.23%
first fit LIFO	0.26%		

Table A.32: Percent fragmentation for each allocator for the LRUsim program, accounting for all overheads

Appendix B

Locality Results for All Allocators

In this appendix, we present the locality results for each trace we studied. We present this data in two sections. In the first section, we present the number of 4K pages necessary to achieve 10%, 50%, and 90% CPU utilization, given a 100 MIP machine and a 10 millisecond disk access time. In the second section, we present the cache miss rates for 16K, 64K, and 256K data caches. The two caches studied are 8-way set associative and fully associative.

B.1 Number of 4K Pages Necessary to Achieve Given Percentage of CPU Time

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
binary buddy	44	66	84	44	66	118
best fit AO	38	56	75	38	57	81
best fit AO 8K	38	56	74	38	57	81
best fit AO deferred AO	49	63	75	49	63	100
best fit AO deferred FIFO	72	76	76	72	76	95
best fit AO deferred LIFO	68	72	76	68	72	83
best fit AO no footer	34	49	66	34	50	80
best fit FIFO	39	56	75	39	57	82
best fit FIFO no footer	36	50	66	36	50	78
best fit LIFO	39	57	75	39	57	82
best fit LIFO deferred AO	52	64	76	52	64	99
best fit LIFO deferred FIFO	73	77	77	73	77	86
best fit LIFO deferred LIFO	69	72	77	69	73	94
best fit LIFO no footer	35	50	66	35	51	75
best fit LIFO split-14	38	56	75	38	57	81
best fit LIFO split-7	39	56	75	39	57	81
double buddy 10K	44	66	81	45	66	115
double buddy 5K	45	65	81	45	65	114
first fit AO	38	55	74	38	56	81
first fit AO 8K	38	57	76	38	58	83
first fit AO deferred AO	46	63	75	46	63	92
first fit AO deferred FIFO	67	75	81	67	76	93
first fit AO deferred LIFO	68	73	78	68	74	85
first fit AO no footer	35	50	68	35	51	80
first fit FIFO	44	66	77	44	67	90
first fit FIFO no footer	42	62	69	43	62	86
first fit LIFO	38	58	74	38	59	97

Table B.1: Number of 4K pages necessary to achieve given percentage of CPU time for Espresso (Part 1)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
first fit LIFO deferred LIFO	65	73	77	65	73	101
first fit LIFO no footer	36	52	68	36	53	85
first fit LIFO split-14	38	57	75	38	58	96
first fit LIFO split-7	38	57	74	38	58	90
half fit	67	72	75	67	72	83
Lea 2.5.1	61	72	81	61	72	98
Lea 2.5.1 no footer	54	64	72	54	65	84
Lea 2.6.1	35	51	67	35	51	75
multi-fit max	64	71	75	64	71	84
multi-fit min	68	72	75	68	72	90
next fit AO	41	59	75	41	59	96
next fit AO 8K	40	57	76	40	58	86
next fit AO deferred AO	48	64	76	48	65	98
next fit AO deferred FIFO	65	74	81	65	74	96
next fit AO deferred LIFO	65	73	78	65	73	96
next fit AO no footer	41	55	71	41	56	79
next fit FIFO	41	62	75	41	63	82
next fit FIFO no footer	38	56	69	38	57	78
next fit LIFO	39	58	75	39	59	100
next fit LIFO deferred LIFO	63	71	77	63	71	92
next fit LIFO no footer	36	54	68	36	55	87
next fit LIFO split-14	38	56	74	38	57	96
next fit LIFO split-7	38	57	75	38	58	94
next fit LIFO WPH	38	57	75	38	58	95
simple segregated 2^N	98	108	118	98	108	247
simple segregated $3 * 2^N$	85	93	113	85	94	249

Table B.2: Number of 4K pages necessary to achieve given percentage of CPU time for Espresso (Part 2)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
binary buddy	142	397	438	148	516	516
best fit AO	286	302	339	286	420	420
best fit AO 8K	281	299	338	282	423	423
best fit AO deferred AO	270	289	338	270	418	418
best fit AO deferred FIFO	272	291	338	272	420	420
best fit AO deferred LIFO	270	293	337	271	421	421
best fit AO no footer	279	295	328	280	409	409
best fit FIFO	289	305	338	290	422	422
best fit FIFO no footer	278	293	327	279	412	412
best fit LIFO	291	306	340	292	422	422
best fit LIFO deferred AO	272	290	338	272	421	421
best fit LIFO deferred FIFO	270	293	338	271	417	417
best fit LIFO deferred LIFO	278	296	335	279	421	421
best fit LIFO no footer	281	297	327	282	411	411
best fit LIFO split-14	279	296	340	280	423	423
best fit LIFO split-7	285	298	341	285	421	421
double buddy 10K	132	318	374	137	455	455
double buddy 5K	133	320	376	139	456	456
first fit AO	288	302	339	289	422	422
first fit AO 8K	288	303	340	289	423	423
first fit AO deferred AO	283	300	337	284	419	419
first fit AO deferred FIFO	271	292	337	272	418	418
first fit AO deferred LIFO	275	295	338	276	419	419
first fit AO no footer	282	296	323	283	412	412
first fit FIFO	309	321	347	309	438	438
first fit FIFO no footer	295	309	333	295	421	421
first fit LIFO	241	878	901	253	981	981

Table B.3: Number of 4K pages necessary to achieve given percentage of CPU time for Ghostscript (Part 1)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
first fit LIFO deferred LIFO	262	398	440	265	519	519
first fit LIFO no footer	229	836	863	241	944	944
first fit LIFO split-14	243	874	898	255	978	978
first fit LIFO split-7	239	884	910	250	988	988
half fit	270	290	338	271	418	418
Lea 2.5.1	199	356	438	206	521	521
Lea 2.5.1 no footer	192	329	414	198	484	484
Lea 2.6.1	109	267	330	113	402	402
multi-fit max	283	298	335	283	420	420
multi-fit min	279	294	336	279	419	419
next fit AO	348	362	376	349	466	466
next fit AO 8K	365	378	394	366	487	487
next fit AO deferred AO	289	305	333	290	424	424
next fit AO deferred FIFO	299	315	336	299	430	430
next fit AO deferred LIFO	292	307	335	293	424	424
next fit AO no footer	336	348	371	337	463	463
next fit FIFO	298	414	468	300	553	553
next fit FIFO no footer	248	408	462	250	543	543
next fit LIFO	223	936	962	259	1045	1045
next fit LIFO deferred LIFO	222	474	509	227	589	589
next fit LIFO no footer	190	872	900	214	982	982
next fit LIFO split-14	233	942	966	276	1050	1050
next fit LIFO split-7	228	944	966	269	1050	1050
next fit LIFO WPH	333	933	954	403	1037	1037
simple segregated 2^N	130	418	455	136	527	527
simple segregated $3 * 2^N$	133	346	403	138	485	485

Table B.4: Number of 4K pages necessary to achieve given percentage of CPU time for Ghostscript (Part 2)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
binary buddy	53	70	101	54	115	115
best fit AO	55	71	110	55	123	123
best fit AO 8K	54	69	110	55	124	124
best fit AO deferred AO	53	68	108	53	120	120
best fit AO deferred FIFO	53	66	108	54	123	123
best fit AO deferred LIFO	55	69	108	56	122	122
best fit AO no footer	49	65	102	50	113	113
best fit FIFO	55	70	108	55	124	124
best fit FIFO no footer	51	66	100	52	114	114
best fit LIFO	55	72	110	56	123	123
best fit LIFO deferred AO	52	65	106	52	123	123
best fit LIFO deferred FIFO	55	68	108	56	123	123
best fit LIFO deferred LIFO	56	70	108	57	124	124
best fit LIFO no footer	49	65	99	50	113	113
best fit LIFO split-14	54	69	109	55	123	123
best fit LIFO split-7	57	72	109	58	124	124
double buddy 10K	50	65	101	51	115	115
double buddy 5K	50	65	101	51	114	114
first fit AO	51	67	108	52	124	124
first fit AO 8K	51	67	109	52	124	124
first fit AO deferred AO	52	67	107	52	122	122
first fit AO deferred FIFO	56	71	107	57	123	123
first fit AO deferred LIFO	54	69	110	55	122	122
first fit AO no footer	47	62	100	47	113	113
first fit FIFO	64	77	109	65	124	124
first fit FIFO no footer	54	66	100	54	113	113
first fit LIFO	90	95	132	91	145	145

Table B.5: Number of 4K pages necessary to achieve given percentage of CPU time for Grobner (Part 1)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
first fit LIFO deferred LIFO	98	107	159	98	171	171
first fit LIFO no footer	76	83	123	78	135	135
first fit LIFO split-14	91	95	131	92	144	144
first fit LIFO split-7	79	86	128	80	139	139
half fit	57	68	107	57	123	123
Lea 2.5.1	63	80	109	64	126	126
Lea 2.5.1 no footer	58	74	107	58	117	117
Lea 2.6.1	47	63	101	48	115	115
multi-fit max	84	98	115	85	126	126
multi-fit min	82	95	112	83	125	125
next fit AO	81	99	117	82	129	129
next fit AO 8K	83	99	119	84	130	130
next fit AO deferred AO	81	98	119	81	132	132
next fit AO deferred FIFO	95	111	132	95	143	143
next fit AO deferred LIFO	94	108	129	94	143	143
next fit AO no footer	73	87	106	74	120	120
next fit FIFO	81	94	117	81	130	130
next fit FIFO no footer	72	85	106	72	121	121
next fit LIFO	79	90	125	80	138	138
next fit LIFO deferred LIFO	93	112	150	94	162	162
next fit LIFO no footer	72	88	116	72	130	130
next fit LIFO split-14	70	88	123	71	138	138
next fit LIFO split-7	71	88	122	72	137	137
next fit LIFO WPH	80	90	116	80	127	127
simple segregated 2^N	44	74	128	46	145	145
simple segregated $3 * 2^N$	43	60	117	44	134	134

Table B.6: Number of 4K pages necessary to achieve given percentage of CPU time for Grobner (Part 2)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
binary buddy	123	127	143	123	127	573
best fit AO	123	128	141	123	129	566
best fit AO 8K	123	128	141	123	129	566
best fit AO deferred AO	121	126	138	121	126	567
best fit AO deferred FIFO	122	127	138	122	127	569
best fit AO deferred LIFO	122	127	141	122	127	569
best fit AO no footer	122	127	141	122	127	566
best fit FIFO	122	128	140	122	129	570
best fit FIFO no footer	122	127	141	122	128	568
best fit LIFO	122	128	140	122	129	570
best fit LIFO deferred AO	121	126	137	121	126	569
best fit LIFO deferred FIFO	122	127	137	122	127	567
best fit LIFO deferred LIFO	122	127	142	122	127	567
best fit LIFO no footer	121	127	139	122	127	568
best fit LIFO split-14	123	129	140	123	130	573
best fit LIFO split-7	123	129	140	123	130	571
double buddy 10K	126	129	147	126	130	578
double buddy 5K	125	129	145	125	129	579
first fit AO	122	128	140	122	129	566
first fit AO 8K	122	128	142	122	129	566
first fit AO deferred AO	121	126	142	121	126	567
first fit AO deferred FIFO	122	127	142	122	127	569
first fit AO deferred LIFO	122	127	143	122	127	569
first fit AO no footer	122	127	141	122	127	566
first fit FIFO	123	129	141	123	129	570
first fit FIFO no footer	123	128	141	123	128	568
first fit LIFO	123	129	141	123	129	570

Table B.7: Number of 4K pages necessary to achieve given percentage of CPU time for Hyper (Part 1)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
first fit LIFO deferred LIFO	122	127	143	122	127	567
first fit LIFO no footer	122	128	140	122	128	568
first fit LIFO split-14	124	129	142	124	130	573
first fit LIFO split-7	124	129	142	124	130	571
half fit	122	127	142	122	127	569
Lea 2.5.1	122	126	142	122	126	565
Lea 2.5.1 no footer	122	127	141	122	127	569
Lea 2.6.1	122	126	142	122	126	565
multi-fit max	123	128	142	123	128	569
multi-fit min	123	128	142	123	128	569
next fit AO	123	129	141	123	129	566
next fit AO 8K	124	129	140	124	130	566
next fit AO deferred AO	122	127	139	122	127	567
next fit AO deferred FIFO	122	127	139	122	127	569
next fit AO deferred LIFO	122	127	140	122	127	569
next fit AO no footer	123	128	142	123	128	566
next fit FIFO	123	129	141	123	129	570
next fit FIFO no footer	123	128	141	123	128	568
next fit LIFO	123	129	142	123	129	570
next fit LIFO deferred LIFO	122	126	138	122	126	568
next fit LIFO no footer	123	128	141	123	128	568
next fit LIFO split-14	124	130	143	124	130	573
next fit LIFO split-7	124	130	143	124	130	571
next fit LIFO WPH	123	129	143	123	129	566
simple segregated 2^N	123	128	142	123	128	569
simple segregated $3 * 2^N$	123	129	142	123	129	571

Table B.8: Number of 4K pages necessary to achieve given percentage of CPU time for Hyper (Part 2)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
binary buddy	179	201	215	180	214	254
best fit AO	153	178	193	154	190	228
best fit AO 8K	152	177	193	153	191	229
best fit AO deferred AO	151	176	192	152	188	230
best fit AO deferred FIFO	155	176	192	156	189	230
best fit AO deferred LIFO	155	176	192	155	189	230
best fit AO no footer	142	165	179	143	175	214
best fit FIFO	152	178	192	154	191	228
best fit FIFO no footer	143	165	179	144	175	215
best fit LIFO	153	178	193	154	191	227
best fit LIFO deferred AO	148	175	192	150	188	230
best fit LIFO deferred FIFO	156	176	192	157	189	230
best fit LIFO deferred LIFO	154	176	192	155	188	230
best fit LIFO no footer	141	165	179	142	175	215
best fit LIFO split-14	153	178	192	154	191	228
best fit LIFO split-7	153	178	192	154	191	228
double buddy 10K	145	167	187	146	181	228
double buddy 5K	145	167	186	146	181	228
first fit AO	146	175	190	147	189	228
first fit AO 8K	146	175	190	146	189	229
first fit AO deferred AO	148	174	191	149	187	230
first fit AO deferred FIFO	154	176	191	155	189	231
first fit AO deferred LIFO	152	177	192	153	189	230
first fit AO no footer	136	162	178	137	174	215
first fit FIFO	158	178	192	158	189	227
first fit FIFO no footer	147	163	178	147	174	213
first fit LIFO	160	183	197	161	194	229

Table B.9: Number of 4K pages necessary to achieve given percentage of CPU time for P2C (Part 1)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
first fit LIFO deferred LIFO	157	180	193	157	189	230
first fit LIFO no footer	150	168	180	151	177	216
first fit LIFO split-14	161	184	197	162	194	231
first fit LIFO split-7	161	184	197	162	194	231
half fit	155	176	192	156	189	230
Lea 2.5.1	170	187	198	171	196	237
Lea 2.5.1 no footer	157	173	183	158	180	222
Lea 2.6.1	145	165	178	145	174	221
multi-fit max	165	180	192	166	188	230
multi-fit min	159	178	193	160	189	230
next fit AO	167	183	198	168	195	233
next fit AO 8K	166	184	199	167	196	235
next fit AO deferred AO	154	179	193	155	190	232
next fit AO deferred FIFO	158	181	195	160	193	234
next fit AO deferred LIFO	157	180	194	158	191	233
next fit AO no footer	153	168	182	154	177	221
next fit FIFO	170	186	201	171	198	237
next fit FIFO no footer	155	170	185	156	180	222
next fit LIFO	169	191	205	170	202	241
next fit LIFO deferred LIFO	159	183	196	160	194	235
next fit LIFO no footer	157	176	191	158	186	227
next fit LIFO split-14	170	192	205	171	202	241
next fit LIFO split-7	170	191	205	172	202	241
next fit LIFO WPH	164	184	198	165	194	234
simple segregated 2^N	184	223	246	186	281	294
simple segregated $3 * 2^N$	163	195	217	165	216	270

Table B.10: Number of 4K pages necessary to achieve given percentage of CPU time for P2C (Part 2)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
binary buddy	42	54	75	42	55	107
best fit AO	39	52	70	39	53	101
best fit AO 8K	39	53	69	39	54	102
best fit AO deferred AO	41	54	71	41	55	101
best fit AO deferred FIFO	41	54	72	41	55	102
best fit AO deferred LIFO	40	54	73	40	55	103
best fit AO no footer	38	52	68	38	52	100
best fit FIFO	39	53	69	39	54	102
best fit FIFO no footer	38	52	68	38	53	100
best fit LIFO	39	52	69	39	53	101
best fit LIFO deferred AO	41	53	71	41	54	102
best fit LIFO deferred FIFO	41	54	72	41	55	103
best fit LIFO deferred LIFO	40	54	72	40	55	102
best fit LIFO no footer	38	51	69	38	52	100
best fit LIFO split-14	39	53	70	39	54	100
best fit LIFO split-7	39	53	69	39	54	101
double buddy 10K	44	55	71	44	56	103
double buddy 5K	44	54	71	44	55	102
first fit AO	39	53	72	39	54	100
first fit AO 8K	39	53	73	39	55	100
first fit AO deferred AO	41	55	71	41	56	102
first fit AO deferred FIFO	40	55	71	41	56	101
first fit AO deferred LIFO	41	55	72	41	56	102
first fit AO no footer	38	52	71	38	54	100
first fit FIFO	41	53	72	41	55	102
first fit FIFO no footer	39	53	71	39	54	99
first fit LIFO	39	55	73	39	56	103

Table B.11: Number of 4K pages necessary to achieve given percentage of CPU time for Perl (Part 1)

	w/o compulsory			w/compulsory		
Allocator Name	10%	50%	90%	10%	50%	90%
first fit LIFO deferred LIFO	41	55	72	41	56	101
first fit LIFO no footer	39	52	71	39	53	99
first fit LIFO split-14	40	54	70	40	55	102
first fit LIFO split-7	39	54	72	39	55	103
half fit	41	54	74	42	55	102
Lea 2.5.1	41	57	73	41	58	102
Lea 2.5.1 no footer	40	55	71	40	56	102
Lea 2.6.1	39	54	68	39	55	99
multi-fit max	43	55	75	43	56	101
multi-fit min	44	55	75	44	56	101
next fit AO	44	55	72	44	56	100
next fit AO 8K	44	55	73	44	56	102
next fit AO deferred AO	44	57	72	44	58	102
next fit AO deferred FIFO	44	56	75	44	57	100
next fit AO deferred LIFO	44	56	75	44	57	101
next fit AO no footer	43	53	73	43	54	99
next fit FIFO	44	56	75	44	57	100
next fit FIFO no footer	41	54	71	41	55	99
next fit LIFO	42	55	74	42	56	101
next fit LIFO deferred LIFO	42	56	73	42	57	101
next fit LIFO no footer	39	54	73	39	55	99
next fit LIFO split-14	40	53	74	40	54	101
next fit LIFO split-7	42	55	74	42	56	100
next fit LIFO WPH	39	54	73	39	55	100
simple segregated 2^N	46	56	78	46	57	120
simple segregated $3 * 2^N$	47	65	80	47	66	117

Table B.12: Number of 4K pages necessary to achieve given percentage of CPU time for Perl (Part 2)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
binary buddy	2.85%	0.361%	.00500%	2.83%	0.261%	.0045%
best fit AO	2.99%	0.400%	.00345%	3.03%	0.363%	.0033%
best fit AO 8K	3.01%	0.380%	.00330%	3.03%	0.364%	.0032%
best fit AO def AO	3.02%	0.388%	.00397%	3.06%	0.392%	.0038%
best fit AO def FIFO	3.18%	0.607%	.00906%	3.19%	0.605%	.0040%
best fit AO def LIFO	3.10%	0.463%	.00363%	3.13%	0.434%	.0033%
best fit AO no-foot	2.90%	0.298%	.00339%	2.96%	0.294%	.0035%
best fit FIFO	2.97%	0.382%	.00335%	3.02%	0.369%	.0033%
best fit FIFO no-foot	2.92%	0.303%	.00330%	2.96%	0.295%	.0034%
best fit LIFO	2.99%	0.396%	.00342%	3.04%	0.366%	.0032%
best fit LIFO def AO	3.00%	0.391%	.00379%	3.04%	0.384%	.0036%
best fit LIFO def FIFO	3.20%	0.618%	.00500%	3.21%	0.605%	.0035%
best fit LIFO def LIFO	3.11%	0.461%	.00404%	3.13%	0.437%	.0038%
best fit LIFO no-foot	2.92%	0.337%	.00306%	2.95%	0.294%	.0031%
best fit LIFO split-14	2.95%	0.394%	.00345%	3.01%	0.365%	.0033%
best fit LIFO split-7	2.99%	0.383%	.00353%	3.03%	0.364%	.0032%
double buddy 10K	2.79%	0.318%	.00372%	2.83%	0.247%	.0035%
double buddy 5K	2.78%	0.291%	.00380%	2.82%	0.245%	.0034%
first fit AO	2.99%	0.369%	.00345%	3.03%	0.365%	.0032%
first fit AO 8K	2.98%	0.365%	.00328%	3.03%	0.365%	.0032%
first fit AO def AO	3.03%	0.398%	.00371%	3.06%	0.386%	.0033%
first fit AO def FIFO	3.16%	0.605%	.00475%	3.19%	0.601%	.0036%
first fit AO def LIFO	3.11%	0.470%	.00366%	3.13%	0.433%	.0034%
first fit AO no-foot	2.88%	0.341%	.00320%	2.94%	0.289%	.0033%
first fit FIFO	3.14%	0.511%	.00484%	3.17%	0.521%	.0036%
first fit FIFO no-foot	3.10%	0.466%	.00380%	3.12%	0.456%	.0035%
first fit LIFO	2.98%	0.366%	.00404%	3.03%	0.363%	.0038%

Table B.13: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Espresso (Part 1)

B.2 Cache Miss Rate for Given Set-Associative Cache Size (32 Byte Line Size, 8-Way Set-Associative)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
first fit LIFO def LIFO	3.07%	0.445%	.00397%	3.09%	0.424%	.0039%
first fit LIFO no-foot	2.88%	0.299%	.00353%	2.94%	0.292%	.0036%
first fit LIFO split-14	2.97%	0.351%	.00404%	3.02%	0.361%	.0038%
first fit LIFO split-7	2.93%	0.376%	.00385%	3.00%	0.361%	.0035%
half fit	3.11%	0.450%	.00361%	3.14%	0.435%	.0033%
Lea 2.5.1	3.09%	0.503%	.00649%	3.12%	0.462%	.0051%
Lea 2.5.1 no-foot	3.02%	0.423%	.00336%	3.05%	0.386%	.0033%
Lea 2.6.1	2.90%	0.318%	.00299%	2.95%	0.292%	.0030%
multi-fit max	3.08%	0.445%	.00352%	3.12%	0.437%	.0033%
multi-fit min	3.11%	0.451%	.00375%	3.13%	0.437%	.0033%
next fit AO	3.14%	0.433%	.00391%	3.17%	0.442%	.0037%
next fit AO 8K	3.12%	0.459%	.00357%	3.16%	0.428%	.0033%
next fit AO def AO	3.09%	0.421%	.00392%	3.11%	0.414%	.0035%
next fit AO def FIFO	3.23%	0.646%	.00532%	3.25%	0.617%	.0041%
next fit AO def LIFO	3.12%	0.488%	.00395%	3.15%	0.461%	.0038%
next fit AO no-foot	3.03%	0.388%	.00316%	3.07%	0.361%	.0032%
next fit FIFO	3.12%	0.479%	.00373%	3.15%	0.477%	.0033%
next fit FIFO no-foot	3.02%	0.384%	.00325%	3.05%	0.381%	.0033%
next fit LIFO	2.95%	0.367%	.00391%	3.00%	0.365%	.0038%
next fit LIFO def LIFO	3.11%	0.454%	.00442%	3.12%	0.429%	.0039%
next fit LIFO no-foot	2.89%	0.318%	.00366%	2.94%	0.291%	.0037%
next fit LIFO split-14	2.96%	0.387%	.00425%	3.01%	0.363%	.0039%
next fit LIFO split-7	2.95%	0.357%	.00389%	3.00%	0.362%	.0039%
next fit LIFO WPH	2.99%	0.372%	.00395%	3.02%	0.364%	.0038%
simple seg 2^N	3.20%	0.619%	.03015%	3.08%	0.530%	.0173%
simple seg $3 * 2^N$	3.23%	0.603%	.01675%	3.22%	0.536%	.0086%

Table B.14: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Espresso (Part 2)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
binary buddy	1.51%	0.265%	0.1146%	1.21%	0.236%	0.1087%
best fit AO	1.13%	0.226%	0.1005%	0.97%	0.216%	0.0980%
best fit AO 8K	1.14%	0.229%	0.1003%	0.98%	0.217%	0.0980%
best fit AO def AO	1.15%	0.235%	0.0999%	1.00%	0.225%	0.0972%
best fit AO def FIFO	1.79%	0.248%	0.0999%	1.67%	0.226%	0.0974%
best fit AO def LIFO	1.16%	0.238%	0.0998%	1.00%	0.226%	0.0972%
best fit AO no-foot	1.04%	0.221%	0.0973%	0.91%	0.209%	0.0949%
best fit FIFO	1.12%	0.225%	0.1004%	0.96%	0.216%	0.0980%
best fit FIFO no-foot	1.07%	0.223%	0.0966%	0.91%	0.209%	0.0949%
best fit LIFO	1.11%	0.226%	0.1001%	0.99%	0.218%	0.0976%
best fit LIFO def AO	1.16%	0.239%	0.0996%	1.00%	0.224%	0.0971%
best fit LIFO def FIFO	1.80%	0.248%	0.0996%	1.65%	0.223%	0.0973%
best fit LIFO def LIFO	1.18%	0.235%	0.0993%	0.99%	0.224%	0.0972%
best fit LIFO no-foot	1.05%	0.222%	0.0970%	0.89%	0.211%	0.0949%
best fit LIFO split-14	1.10%	0.226%	0.0999%	0.98%	0.217%	0.0975%
best fit LIFO split-7	1.11%	0.226%	0.0998%	0.98%	0.217%	0.0975%
double buddy 10K	1.17%	0.237%	0.1025%	0.99%	0.224%	0.0998%
double buddy 5K	1.17%	0.235%	0.1022%	1.00%	0.225%	0.0999%
first fit AO	1.10%	0.226%	0.1000%	0.96%	0.215%	0.0980%
first fit AO 8K	1.08%	0.226%	0.0999%	0.96%	0.215%	0.0980%
first fit AO def AO	1.18%	0.238%	0.0998%	1.01%	0.228%	0.0975%
first fit AO def FIFO	1.80%	0.249%	0.0995%	1.68%	0.227%	0.0976%
first fit AO def LIFO	1.17%	0.238%	0.0995%	1.01%	0.226%	0.0973%
first fit AO no-foot	1.01%	0.220%	0.0967%	0.88%	0.209%	0.0948%
first fit FIFO	2.12%	0.407%	0.1075%	1.99%	0.356%	0.1045%
first fit FIFO no-foot	1.98%	0.333%	0.1013%	1.87%	0.298%	0.0993%
first fit LIFO	1.11%	0.335%	0.1970%	1.00%	0.325%	0.1950%

Table B.15: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Ghostscript (Part 1)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
first fit LIFO def LIFO	1.13%	0.253%	0.1180%	1.01%	0.241%	0.1167%
first fit LIFO no-foot	1.01%	0.323%	0.1907%	0.90%	0.313%	0.1886%
first fit LIFO split-14	1.10%	0.335%	0.1978%	1.00%	0.325%	0.1957%
first fit LIFO split-7	1.09%	0.332%	0.1981%	0.98%	0.323%	0.1960%
half fit	1.20%	0.239%	0.0996%	0.98%	0.224%	0.0971%
Lea 2.5.1	1.47%	0.259%	0.1116%	1.19%	0.246%	0.1083%
Lea 2.5.1 no-foot	1.35%	0.255%	0.1079%	1.12%	0.237%	0.1041%
Lea 2.6.1	1.05%	0.219%	0.0964%	0.90%	0.208%	0.0947%
multi-fit max	1.19%	0.248%	0.0998%	1.01%	0.233%	0.0973%
multi-fit min	1.17%	0.245%	0.1000%	0.99%	0.232%	0.0973%
next fit AO	1.97%	0.705%	0.1502%	1.82%	0.670%	0.1380%
next fit AO 8K	1.99%	0.781%	0.1816%	1.87%	0.752%	0.1662%
next fit AO def AO	1.30%	0.256%	0.1025%	1.10%	0.240%	0.0998%
next fit AO def FIFO	1.89%	0.271%	0.1057%	1.75%	0.240%	0.1027%
next fit AO def LIFO	1.30%	0.254%	0.1027%	1.10%	0.239%	0.1000%
next fit AO no-foot	1.79%	0.607%	0.1289%	1.71%	0.636%	0.1308%
next fit FIFO	2.03%	0.461%	0.1362%	1.92%	0.420%	0.1329%
next fit FIFO no-foot	1.81%	0.397%	0.1288%	1.75%	0.376%	0.1275%
next fit LIFO	1.05%	0.351%	0.2207%	0.97%	0.339%	0.2187%
next fit LIFO def LIFO	1.14%	0.268%	0.1315%	0.98%	0.253%	0.1303%
next fit LIFO no-foot	0.97%	0.329%	0.2057%	0.88%	0.323%	0.2043%
next fit LIFO split-14	1.06%	0.351%	0.2197%	0.98%	0.339%	0.2175%
next fit LIFO split-7	1.05%	0.350%	0.2216%	0.97%	0.339%	0.2195%
next fit LIFO WPH	1.08%	0.348%	0.2213%	0.99%	0.339%	0.2185%
simple seg 2^N	1.60%	0.293%	0.1212%	1.31%	0.260%	0.1158%
simple seg $3 * 2^N$	1.32%	0.277%	0.1121%	1.09%	0.245%	0.1065%

Table B.16: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Ghostscript (Part 2)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
binary buddy	0.55%	0.099%	0.0831%	0.38%	0.094%	0.0825%
best fit AO	0.47%	0.111%	0.0904%	0.44%	0.103%	0.0902%
best fit AO 8K	0.45%	0.108%	0.0903%	0.43%	0.103%	0.0901%
best fit AO def AO	0.52%	0.107%	0.0888%	0.47%	0.102%	0.0888%
best fit AO def FIFO	0.56%	0.113%	0.0891%	0.53%	0.105%	0.0889%
best fit AO def LIFO	0.54%	0.110%	0.0891%	0.51%	0.104%	0.0889%
best fit AO no-foot	0.42%	0.097%	0.0804%	0.40%	0.091%	0.0801%
best fit FIFO	0.45%	0.108%	0.0904%	0.44%	0.103%	0.0902%
best fit FIFO no-foot	0.42%	0.096%	0.0804%	0.40%	0.091%	0.0801%
best fit LIFO	0.45%	0.110%	0.0903%	0.43%	0.102%	0.0902%
best fit LIFO def AO	0.50%	0.106%	0.0889%	0.47%	0.102%	0.0889%
best fit LIFO def FIFO	0.55%	0.111%	0.0890%	0.53%	0.104%	0.0889%
best fit LIFO def LIFO	0.53%	0.108%	0.0891%	0.51%	0.104%	0.0889%
best fit LIFO no-foot	0.42%	0.096%	0.0805%	0.40%	0.091%	0.0802%
best fit LIFO split-14	0.45%	0.110%	0.0904%	0.44%	0.103%	0.0902%
best fit LIFO split-7	0.44%	0.108%	0.0904%	0.43%	0.102%	0.0902%
double buddy 10K	0.47%	0.093%	0.0793%	0.39%	0.090%	0.0791%
double buddy 5K	0.46%	0.094%	0.0793%	0.39%	0.090%	0.0791%
first fit AO	0.41%	0.105%	0.0898%	0.40%	0.100%	0.0897%
first fit AO 8K	0.41%	0.105%	0.0898%	0.40%	0.100%	0.0896%
first fit AO def AO	0.50%	0.110%	0.0889%	0.46%	0.101%	0.0889%
first fit AO def FIFO	0.55%	0.114%	0.0890%	0.51%	0.104%	0.0890%
first fit AO def LIFO	0.53%	0.112%	0.0890%	0.51%	0.105%	0.0890%
first fit AO no-foot	0.37%	0.092%	0.0793%	0.37%	0.089%	0.0791%
first fit FIFO	0.96%	0.114%	0.0905%	0.83%	0.105%	0.0903%
first fit FIFO no-foot	0.81%	0.101%	0.0799%	0.68%	0.092%	0.0796%
first fit LIFO	0.49%	0.135%	0.1099%	0.46%	0.127%	0.1099%

Table B.17: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Grobner (Part 1)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
first fit LIFO def LIFO	0.58%	0.174%	0.1343%	0.55%	0.162%	0.1340%
first fit LIFO no-foot	0.44%	0.119%	0.1015%	0.43%	0.115%	0.1015%
first fit LIFO split-14	0.48%	0.130%	0.1089%	0.46%	0.127%	0.1090%
first fit LIFO split-7	0.48%	0.128%	0.1060%	0.47%	0.124%	0.1061%
half fit	0.53%	0.110%	0.0890%	0.51%	0.104%	0.0889%
Lea 2.5.1	0.71%	0.144%	0.0933%	0.68%	0.119%	0.0933%
Lea 2.5.1 no-foot	0.67%	0.131%	0.0854%	0.67%	0.109%	0.0854%
Lea 2.6.1	0.40%	0.094%	0.0804%	0.38%	0.091%	0.0802%
multi-fit max	0.57%	0.128%	0.0933%	0.53%	0.115%	0.0933%
multi-fit min	0.57%	0.115%	0.0918%	0.52%	0.108%	0.0916%
next fit AO	0.76%	0.133%	0.0966%	0.71%	0.120%	0.0965%
next fit AO 8K	0.98%	0.164%	0.0992%	0.88%	0.139%	0.0991%
next fit AO def AO	0.69%	0.169%	0.0980%	0.67%	0.143%	0.0979%
next fit AO def FIFO	0.78%	0.208%	0.1106%	0.75%	0.175%	0.1103%
next fit AO def LIFO	0.71%	0.187%	0.1083%	0.70%	0.160%	0.1080%
next fit AO no-foot	0.80%	0.129%	0.0865%	0.74%	0.111%	0.0865%
next fit FIFO	0.93%	0.140%	0.0972%	0.84%	0.121%	0.0972%
next fit FIFO no-foot	0.96%	0.153%	0.0873%	0.88%	0.125%	0.0874%
next fit LIFO	0.49%	0.133%	0.1063%	0.47%	0.128%	0.1063%
next fit LIFO def LIFO	0.58%	0.166%	0.1241%	0.56%	0.158%	0.1239%
next fit LIFO no-foot	0.44%	0.119%	0.0968%	0.43%	0.114%	0.0968%
next fit LIFO split-14	0.50%	0.134%	0.1067%	0.47%	0.129%	0.1070%
next fit LIFO split-7	0.48%	0.133%	0.1050%	0.47%	0.128%	0.1056%
next fit LIFO WPH	0.45%	0.116%	0.0959%	0.43%	0.111%	0.0959%
simple seg 2^N	0.74%	0.138%	0.1061%	0.54%	0.128%	0.1061%
simple seg $3 * 2^N$	0.54%	0.124%	0.0923%	0.49%	0.110%	0.0922%

Table B.18: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Grobner (Part 2)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
binary buddy	1.12%	0.688%	0.197%	1.04%	0.679%	0.162%
best fit AO	1.14%	0.695%	0.197%	1.06%	0.687%	0.162%
best fit AO 8K	1.14%	0.694%	0.197%	1.06%	0.687%	0.162%
best fit AO def AO	1.14%	0.693%	0.197%	1.05%	0.684%	0.162%
best fit AO def FIFO	1.17%	0.726%	0.204%	1.08%	0.716%	0.166%
best fit AO def LIFO	1.14%	0.692%	0.197%	1.05%	0.683%	0.163%
best fit AO no-foot	1.13%	0.688%	0.193%	1.05%	0.681%	0.160%
best fit FIFO	1.15%	0.694%	0.197%	1.06%	0.687%	0.162%
best fit FIFO no-foot	1.13%	0.689%	0.193%	1.05%	0.681%	0.160%
best fit LIFO	1.14%	0.694%	0.197%	1.05%	0.686%	0.162%
best fit LIFO def AO	1.14%	0.693%	0.197%	1.05%	0.684%	0.162%
best fit LIFO def FIFO	1.17%	0.725%	0.204%	1.08%	0.716%	0.166%
best fit LIFO def LIFO	1.13%	0.692%	0.197%	1.05%	0.683%	0.163%
best fit LIFO no-foot	1.13%	0.689%	0.193%	1.05%	0.681%	0.160%
best fit LIFO split-14	1.14%	0.694%	0.193%	1.05%	0.686%	0.163%
best fit LIFO split-7	1.14%	0.694%	0.193%	1.05%	0.686%	0.163%
double buddy 10K	1.12%	0.691%	0.199%	1.04%	0.679%	0.162%
double buddy 5K	1.12%	0.688%	0.197%	1.04%	0.680%	0.162%
first fit AO	1.14%	0.693%	0.197%	1.05%	0.685%	0.162%
first fit AO 8K	1.14%	0.692%	0.197%	1.05%	0.685%	0.162%
first fit AO def AO	1.14%	0.693%	0.197%	1.05%	0.684%	0.162%
first fit AO def FIFO	1.17%	0.726%	0.204%	1.08%	0.716%	0.166%
first fit AO def LIFO	1.14%	0.692%	0.197%	1.05%	0.683%	0.163%
first fit AO no-foot	1.13%	0.688%	0.193%	1.05%	0.681%	0.160%
first fit FIFO	1.16%	0.722%	0.208%	1.07%	0.715%	0.168%
first fit FIFO no-foot	1.14%	0.714%	0.207%	1.06%	0.707%	0.169%
first fit LIFO	1.14%	0.700%	0.205%	1.06%	0.692%	0.168%

Table B.19: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Hyper (Part 1)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
first fit LIFO def LIFO	1.13%	0.693%	0.198%	1.05%	0.685%	0.163%
first fit LIFO no-foot	1.13%	0.694%	0.201%	1.05%	0.687%	0.166%
first fit LIFO split-14	1.14%	0.700%	0.203%	1.06%	0.692%	0.169%
first fit LIFO split-7	1.14%	0.700%	0.203%	1.06%	0.692%	0.169%
half fit	1.14%	0.692%	0.197%	1.05%	0.683%	0.163%
Lea 2.5.1	1.15%	0.697%	0.199%	1.05%	0.688%	0.163%
Lea 2.5.1 no-foot	1.13%	0.691%	0.196%	1.05%	0.683%	0.161%
Lea 2.6.1	1.12%	0.688%	0.194%	1.05%	0.681%	0.160%
multi-fit max	1.13%	0.692%	0.198%	1.05%	0.684%	0.163%
multi-fit min	1.13%	0.692%	0.198%	1.05%	0.684%	0.163%
next fit AO	1.16%	0.723%	0.208%	1.08%	0.716%	0.168%
next fit AO 8K	1.15%	0.723%	0.216%	1.07%	0.715%	0.174%
next fit AO def AO	1.14%	0.693%	0.197%	1.05%	0.684%	0.162%
next fit AO def FIFO	1.17%	0.726%	0.204%	1.08%	0.718%	0.166%
next fit AO def LIFO	1.14%	0.693%	0.198%	1.05%	0.685%	0.163%
next fit AO no-foot	1.14%	0.716%	0.207%	1.06%	0.709%	0.168%
next fit FIFO	1.16%	0.722%	0.208%	1.07%	0.714%	0.168%
next fit FIFO no-foot	1.14%	0.714%	0.208%	1.06%	0.707%	0.169%
next fit LIFO	1.14%	0.700%	0.206%	1.06%	0.693%	0.168%
next fit LIFO def LIFO	1.14%	0.694%	0.198%	1.05%	0.685%	0.163%
next fit LIFO no-foot	1.13%	0.695%	0.202%	1.05%	0.687%	0.167%
next fit LIFO split-14	1.14%	0.700%	0.203%	1.06%	0.693%	0.169%
next fit LIFO split-7	1.14%	0.700%	0.203%	1.06%	0.693%	0.169%
next fit LIFO WPH	1.13%	0.700%	0.205%	1.05%	0.692%	0.168%
simple seg 2^N	1.14%	0.690%	0.197%	1.05%	0.678%	0.162%
simple seg $3 * 2^N$	1.14%	0.688%	0.197%	1.05%	0.678%	0.162%

Table B.20: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Hyper (Part 2)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
binary buddy	1.31%	0.408%	0.0645%	1.20%	0.357%	0.0516%
best fit AO	1.44%	0.451%	0.0584%	1.39%	0.428%	0.0533%
best fit AO 8K	1.45%	0.455%	0.0588%	1.40%	0.431%	0.0533%
best fit AO def AO	1.50%	0.490%	0.0614%	1.46%	0.460%	0.0538%
best fit AO def FIFO	1.65%	0.595%	0.0625%	1.60%	0.574%	0.0537%
best fit AO def LIFO	1.52%	0.494%	0.0621%	1.47%	0.461%	0.0547%
best fit AO no-foot	1.34%	0.389%	0.0492%	1.29%	0.368%	0.0456%
best fit FIFO	1.44%	0.450%	0.0584%	1.39%	0.428%	0.0533%
best fit FIFO no-foot	1.34%	0.387%	0.0487%	1.29%	0.366%	0.0456%
best fit LIFO	1.43%	0.448%	0.0587%	1.39%	0.428%	0.0534%
best fit LIFO def AO	1.52%	0.497%	0.0619%	1.48%	0.465%	0.0542%
best fit LIFO def FIFO	1.64%	0.598%	0.0621%	1.60%	0.574%	0.0538%
best fit LIFO def LIFO	1.53%	0.495%	0.0611%	1.48%	0.461%	0.0544%
best fit LIFO no-foot	1.33%	0.389%	0.0487%	1.29%	0.367%	0.0454%
best fit LIFO split-14	1.46%	0.454%	0.0594%	1.40%	0.431%	0.0534%
best fit LIFO split-7	1.45%	0.455%	0.0594%	1.40%	0.432%	0.0534%
double buddy 10K	1.32%	0.363%	0.0485%	1.26%	0.336%	0.0458%
double buddy 5K	1.32%	0.361%	0.0483%	1.26%	0.335%	0.0458%
first fit AO	1.43%	0.445%	0.0584%	1.39%	0.432%	0.0532%
first fit AO 8K	1.43%	0.445%	0.0585%	1.39%	0.432%	0.0532%
first fit AO def AO	1.51%	0.484%	0.0617%	1.46%	0.454%	0.0543%
first fit AO def FIFO	1.64%	0.589%	0.0626%	1.59%	0.573%	0.0535%
first fit AO def LIFO	1.52%	0.489%	0.0632%	1.48%	0.462%	0.0547%
first fit AO no-foot	1.33%	0.387%	0.0489%	1.30%	0.374%	0.0458%
first fit FIFO	1.62%	0.619%	0.0613%	1.57%	0.607%	0.0531%
first fit FIFO no-foot	1.54%	0.545%	0.0507%	1.48%	0.525%	0.0457%
first fit LIFO	1.44%	0.464%	0.0659%	1.40%	0.440%	0.0585%

Table B.21: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for P2C (Part 1)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
first fit LIFO def LIFO	1.54%	0.499%	0.0631%	1.49%	0.468%	0.0557%
first fit LIFO no-foot	1.37%	0.418%	0.0533%	1.33%	0.392%	0.0494%
first fit LIFO split-14	1.46%	0.470%	0.0665%	1.42%	0.446%	0.0589%
first fit LIFO split-7	1.46%	0.470%	0.0664%	1.42%	0.446%	0.0589%
half fit	1.53%	0.499%	0.0629%	1.48%	0.463%	0.0543%
Lea 2.5.1	1.58%	0.539%	0.0698%	1.53%	0.510%	0.0587%
Lea 2.5.1 no-foot	1.55%	0.498%	0.0576%	1.49%	0.461%	0.0511%
Lea 2.6.1	1.35%	0.395%	0.0490%	1.30%	0.369%	0.0455%
multi-fit max	1.51%	0.496%	0.0640%	1.46%	0.458%	0.0549%
multi-fit min	1.53%	0.497%	0.0629%	1.49%	0.463%	0.0551%
next fit AO	1.67%	0.629%	0.0682%	1.60%	0.611%	0.0577%
next fit AO 8K	1.67%	0.632%	0.0704%	1.60%	0.613%	0.0579%
next fit AO def AO	1.52%	0.496%	0.0617%	1.48%	0.467%	0.0555%
next fit AO def FIFO	1.64%	0.600%	0.0647%	1.60%	0.581%	0.0557%
next fit AO def LIFO	1.52%	0.493%	0.0645%	1.48%	0.468%	0.0560%
next fit AO no-foot	1.58%	0.567%	0.0588%	1.52%	0.545%	0.0489%
next fit FIFO	1.69%	0.645%	0.0699%	1.61%	0.632%	0.0587%
next fit FIFO no-foot	1.60%	0.576%	0.0566%	1.56%	0.568%	0.0495%
next fit LIFO	1.46%	0.483%	0.0715%	1.42%	0.456%	0.0639%
next fit LIFO def LIFO	1.51%	0.494%	0.0652%	1.47%	0.462%	0.0568%
next fit LIFO no-foot	1.37%	0.427%	0.0583%	1.32%	0.402%	0.0535%
next fit LIFO split-14	1.48%	0.484%	0.0723%	1.43%	0.456%	0.0636%
next fit LIFO split-7	1.48%	0.484%	0.0723%	1.43%	0.456%	0.0637%
next fit LIFO WPH	1.47%	0.485%	0.0662%	1.43%	0.461%	0.0583%
simple seg 2^N	1.71%	0.602%	0.1014%	1.60%	0.563%	0.0914%
simple seg $3 * 2^N$	1.58%	0.542%	0.0758%	1.53%	0.492%	0.0587%

Table B.22: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for P2C (Part 2)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
binary buddy	1.11%	.0038%	.00327%	0.65%	.0038%	.00327%
best fit AO	0.99%	.0037%	.00326%	0.70%	.0038%	.00326%
best fit AO 8K	1.03%	.0037%	.00325%	0.72%	.0038%	.00326%
best fit AO def AO	0.98%	.0038%	.00328%	0.71%	.0038%	.00328%
best fit AO def FIFO	1.67%	.0038%	.00328%	1.05%	.0038%	.00328%
best fit AO def LIFO	0.93%	.0038%	.00328%	0.73%	.0038%	.00329%
best fit AO no-foot	1.01%	.0037%	.00319%	0.67%	.0037%	.00319%
best fit FIFO	0.99%	.0038%	.00326%	0.70%	.0038%	.00326%
best fit FIFO no-foot	1.03%	.0037%	.00319%	0.67%	.0037%	.00319%
best fit LIFO	0.99%	.0037%	.00326%	0.72%	.0037%	.00326%
best fit LIFO def AO	1.09%	.0038%	.00328%	0.73%	.0038%	.00328%
best fit LIFO def FIFO	1.57%	.0038%	.00329%	1.06%	.0038%	.00328%
best fit LIFO def LIFO	0.97%	.0038%	.00328%	0.74%	.0038%	.00328%
best fit LIFO no-foot	1.00%	.0037%	.00320%	0.69%	.0037%	.00320%
best fit LIFO split-14	0.97%	.0038%	.00327%	0.70%	.0037%	.00326%
best fit LIFO split-7	0.96%	.0038%	.00327%	0.71%	.0037%	.00326%
double buddy 10K	0.83%	.0038%	.00326%	0.71%	.0038%	.00326%
double buddy 5K	0.77%	.0038%	.00326%	0.71%	.0038%	.00325%
first fit AO	0.93%	.0038%	.00326%	0.71%	.0038%	.00326%
first fit AO 8K	1.01%	.0038%	.00326%	0.71%	.0038%	.00326%
first fit AO def AO	1.07%	.0038%	.00328%	0.72%	.0038%	.00328%
first fit AO def FIFO	1.50%	.0038%	.00328%	1.03%	.0038%	.00328%
first fit AO def LIFO	1.02%	.0038%	.00328%	0.73%	.0038%	.00328%
first fit AO no-foot	0.97%	.0037%	.00319%	0.68%	.0037%	.00319%
first fit FIFO	1.66%	.0038%	.00330%	1.11%	.0038%	.00330%
first fit FIFO no-foot	1.53%	.0037%	.00320%	1.08%	.0037%	.00324%
first fit LIFO	1.06%	.0038%	.00331%	0.73%	.0038%	.00330%

Table B.23: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Perl (Part 1)

	8-way set-associative			fully-associative		
Allocator Name	16K	64K	256K	16K	64K	256K
first fit LIFO def LIFO	1.04%	.0038%	.00328%	0.76%	.0038%	.00328%
first fit LIFO no-foot	0.92%	.0037%	.00324%	0.68%	.0037%	.00324%
first fit LIFO split-14	1.05%	.0038%	.00329%	0.72%	.0038%	.00329%
first fit LIFO split-7	1.06%	.0038%	.00331%	0.73%	.0038%	.00330%
half fit	1.09%	.0038%	.00328%	0.74%	.0038%	.00328%
Lea 2.5.1	1.40%	.0038%	.00328%	0.76%	.0038%	.00328%
Lea 2.5.1 no-foot	0.98%	.0037%	.00321%	0.74%	.0037%	.00320%
Lea 2.6.1	1.22%	.0037%	.00319%	0.68%	.0037%	.00319%
multi-fit max	1.30%	.0038%	.00330%	0.77%	.0038%	.00330%
multi-fit min	1.43%	.0038%	.00330%	0.77%	.0038%	.00330%
next fit AO	1.31%	.0038%	.00330%	0.87%	.0038%	.00330%
next fit AO 8K	1.35%	.0039%	.00338%	0.89%	.0039%	.00337%
next fit AO def AO	1.07%	.0038%	.00329%	0.74%	.0038%	.00329%
next fit AO def FIFO	1.66%	.0038%	.00329%	1.14%	.0038%	.00329%
next fit AO def LIFO	1.11%	.0038%	.00330%	0.76%	.0038%	.00330%
next fit AO no-foot	1.19%	.0037%	.00325%	0.84%	.0037%	.00324%
next fit FIFO	1.63%	.0039%	.00330%	0.97%	.0038%	.00330%
next fit FIFO no-foot	1.51%	.0038%	.00325%	0.92%	.0038%	.00325%
next fit LIFO	0.96%	.0038%	.00330%	0.75%	.0038%	.00330%
next fit LIFO def LIFO	1.23%	.0038%	.00328%	0.74%	.0038%	.00328%
next fit LIFO no-foot	0.76%	.0038%	.00330%	0.70%	.0038%	.00330%
next fit LIFO split-14	1.02%	.0039%	.00335%	0.75%	.0039%	.00335%
next fit LIFO split-7	0.96%	.0038%	.00330%	0.75%	.0038%	.00330%
next fit LIFO WPH	1.04%	.0039%	.00328%	0.74%	.0038%	.00328%
simple seg 2^N	1.58%	.0043%	.00360%	0.90%	.0042%	.00361%
simple seg $3 * 2^N$	1.42%	.0039%	.00333%	0.77%	.0039%	.00333%

Table B.24: Cache miss rate for given set-associative cache size (32 byte line size, 8-way set-associative) for Perl (Part 2)

Appendix C

Actual Fragmentation Plots for Selected Allocators

The pictures that follow are fragmentation plots of the form first shown in Section 2.14. These pictures show the heap use across time for each of the selected allocators, *when all overheads have been removed.* As was described in Section 2.14, darker pixels indicate a larger percentage of memory in use.

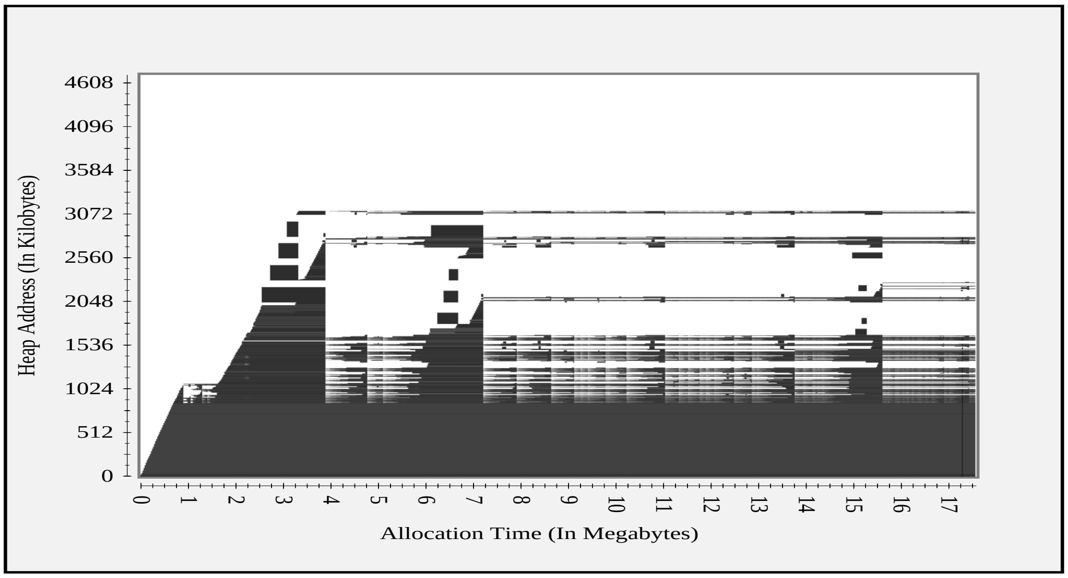

Figure C.1: Fragmentation plot for GCC using the binary-buddy allocator (accounting for all overheads)

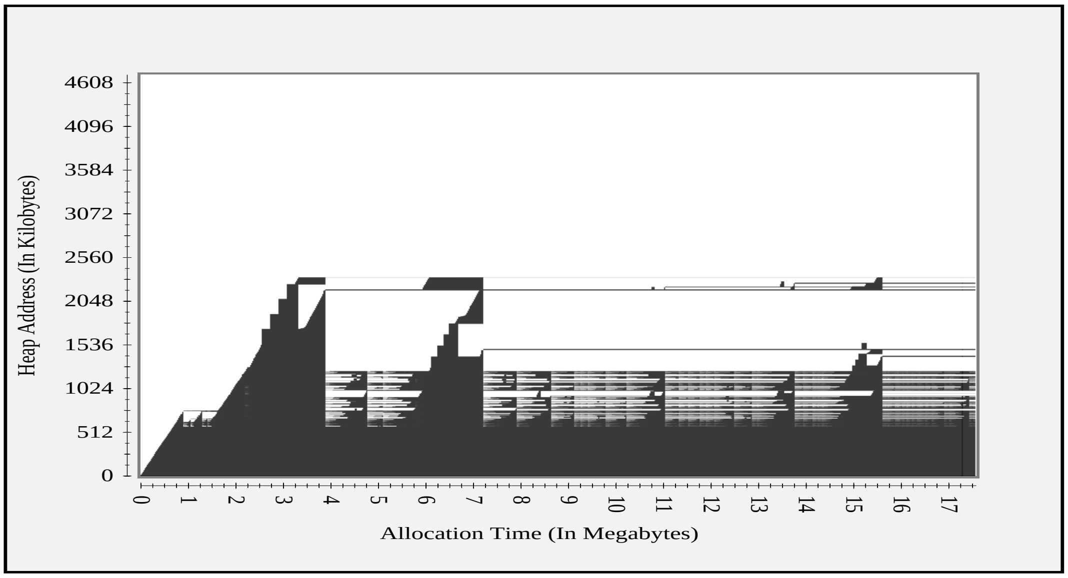

Figure C.2: Fragmentation plot for GCC using the best-fit LIFO no footer allocator (accounting for all overheads)

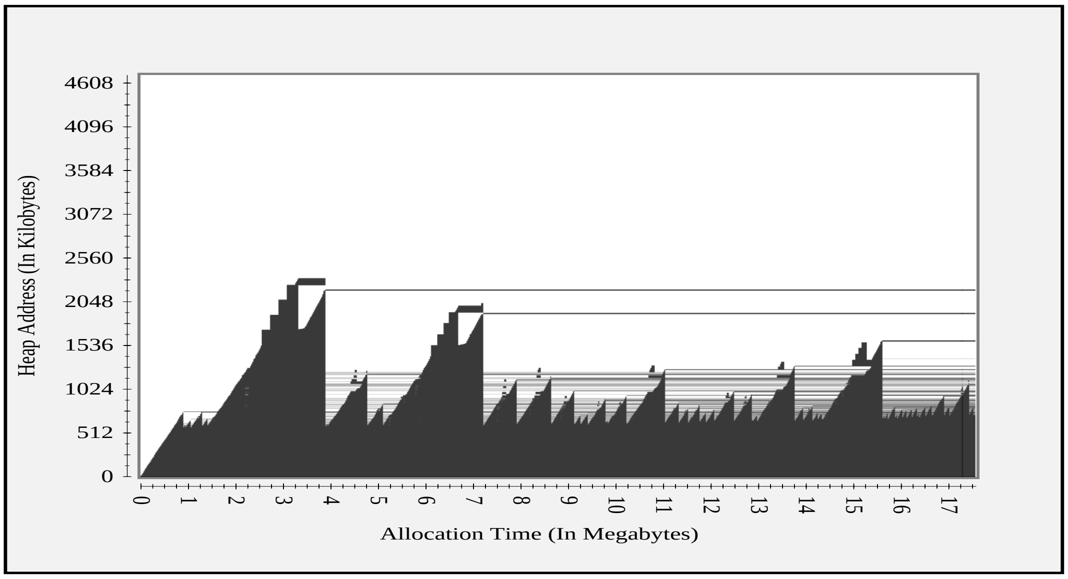

Figure C.3: Fragmentation plot for GCC using the first-fit address-ordered no-footer allocator (accounting for all overheads)

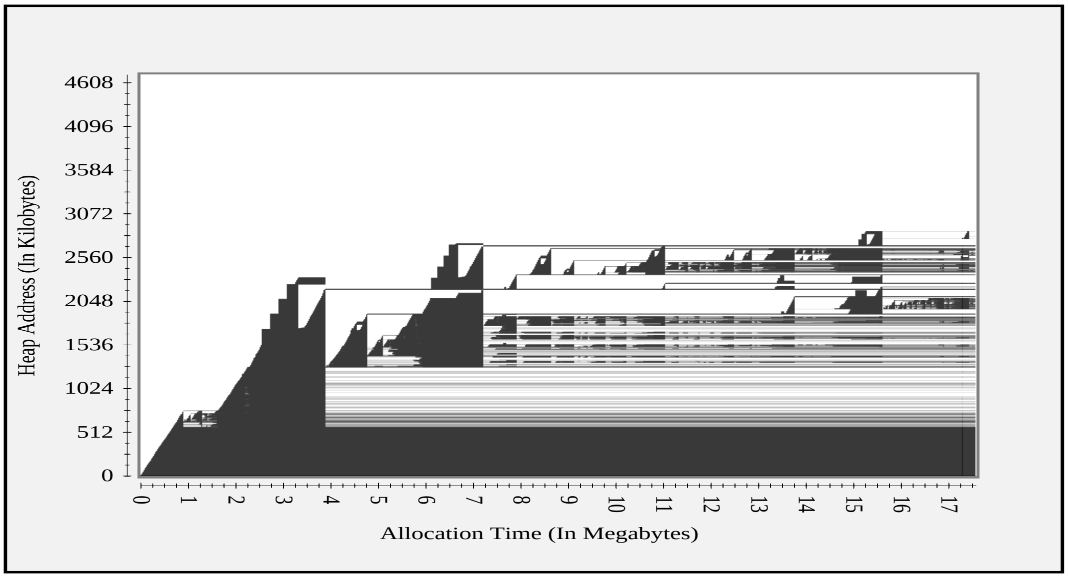

Figure C.4: Fragmentation plot for GCC using the first-fit LIFO no footer allocator (accounting for all overheads)

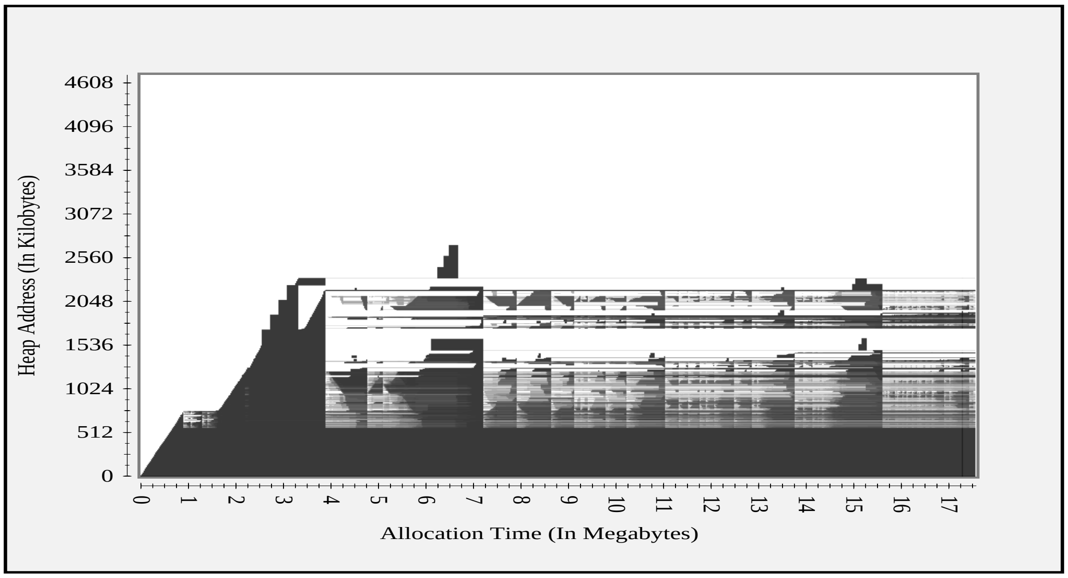

Figure C.5: Fragmentation plot for GCC using the half-fit allocator (accounting for all overheads)

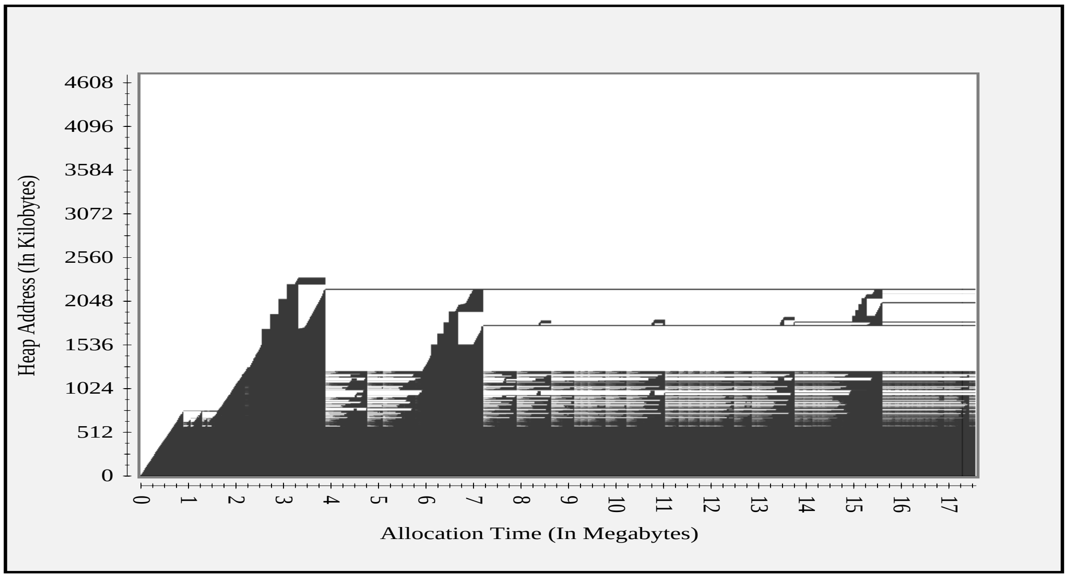

Figure C.6: Fragmentation plot for GCC using Lea's 2.6.1 allocator (accounting for all overheads)

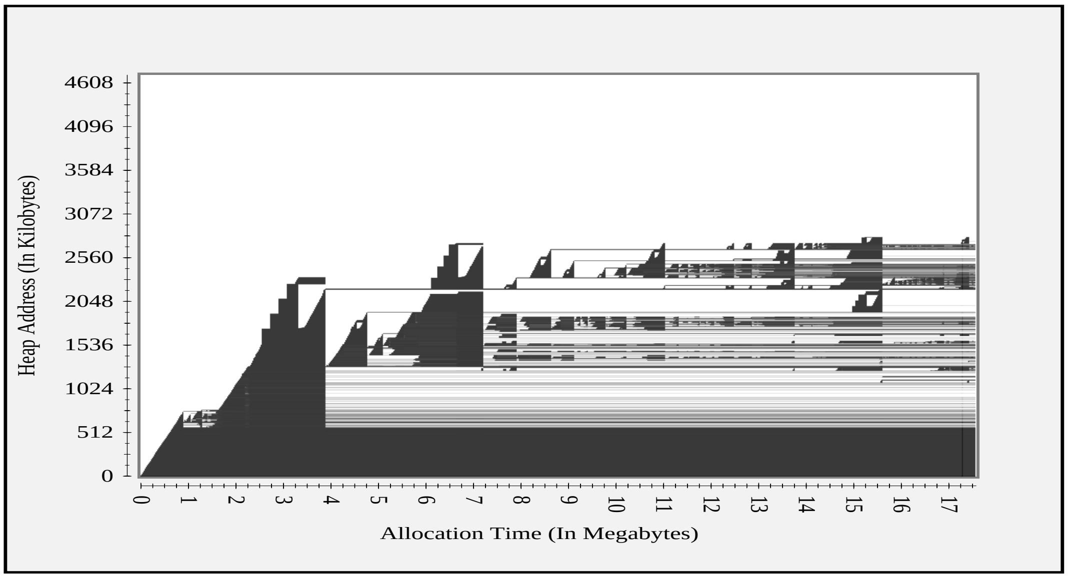

Figure C.7: Fragmentation plot for GCC using the next-fit LIFO no footer allocator (accounting for all overheads)

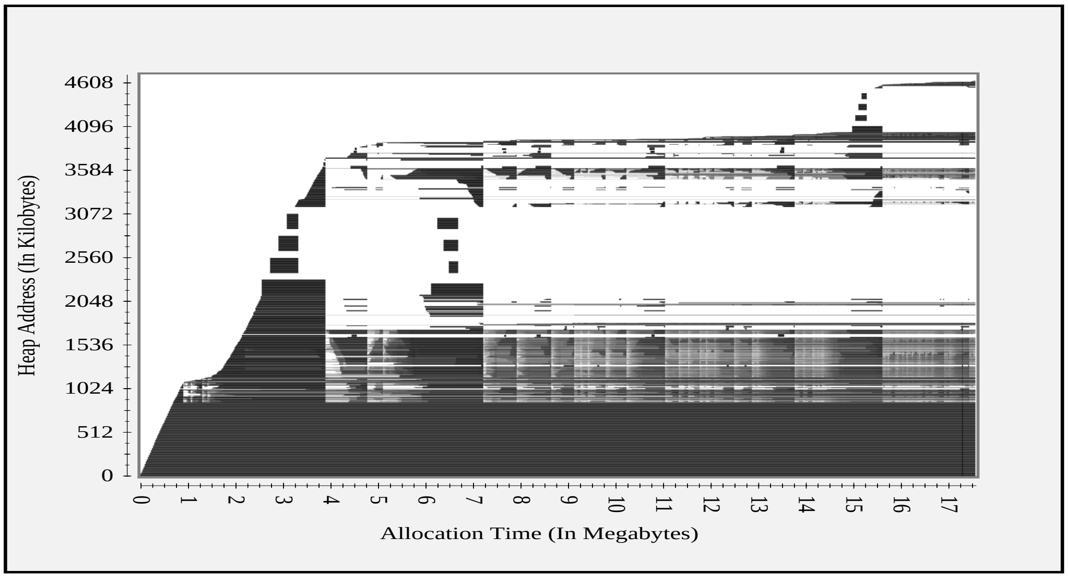

Figure C.8: Fragmentation plot for GCC using the simple segregated storage 2^N allocator (accounting for all overheads)

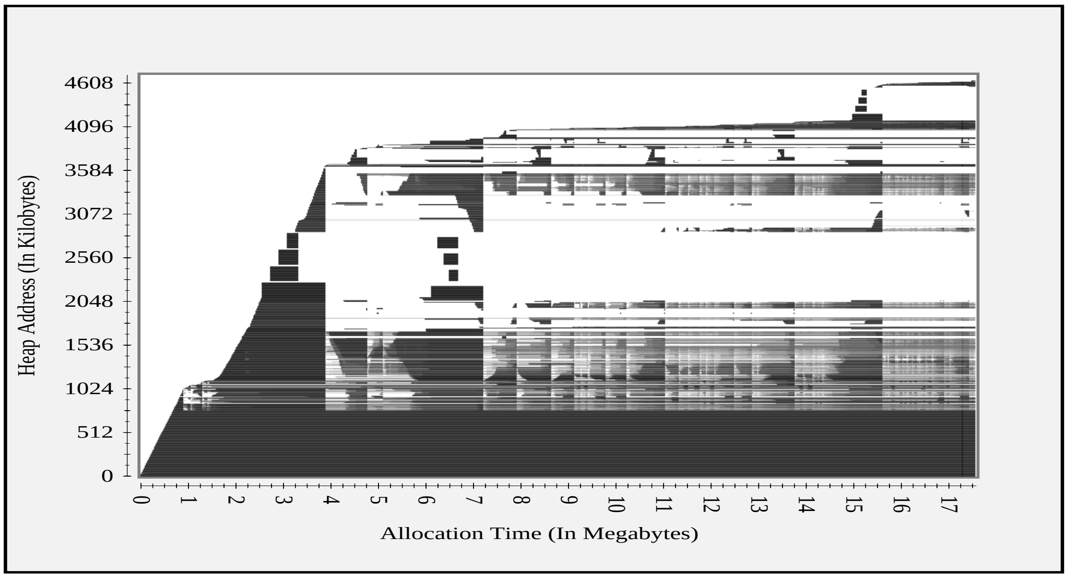

Figure C.9: Fragmentation plot for GCC using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

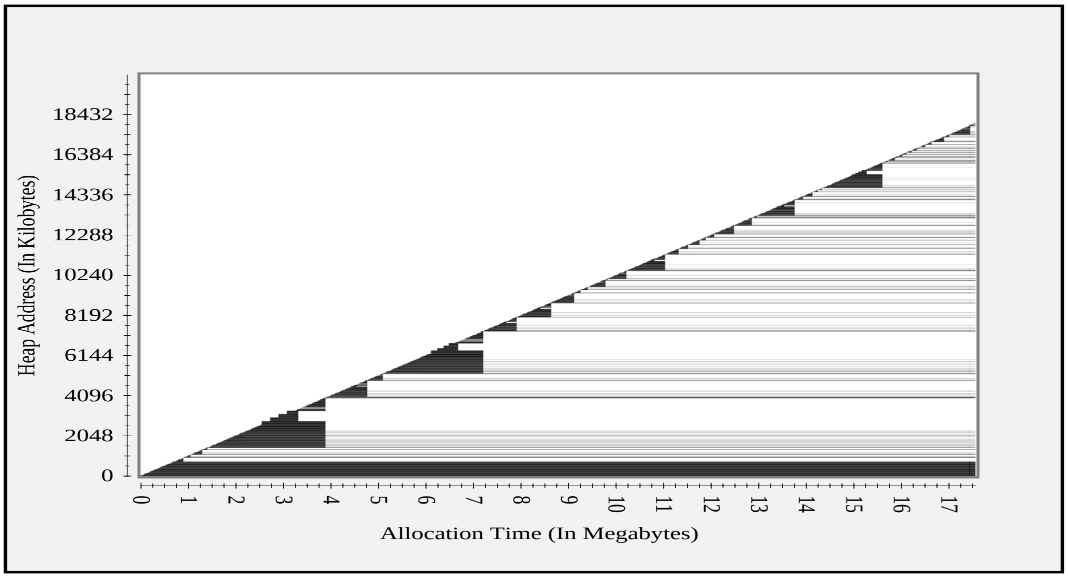

Figure C.10: Fragmentation plot for GCC using the linear allocator

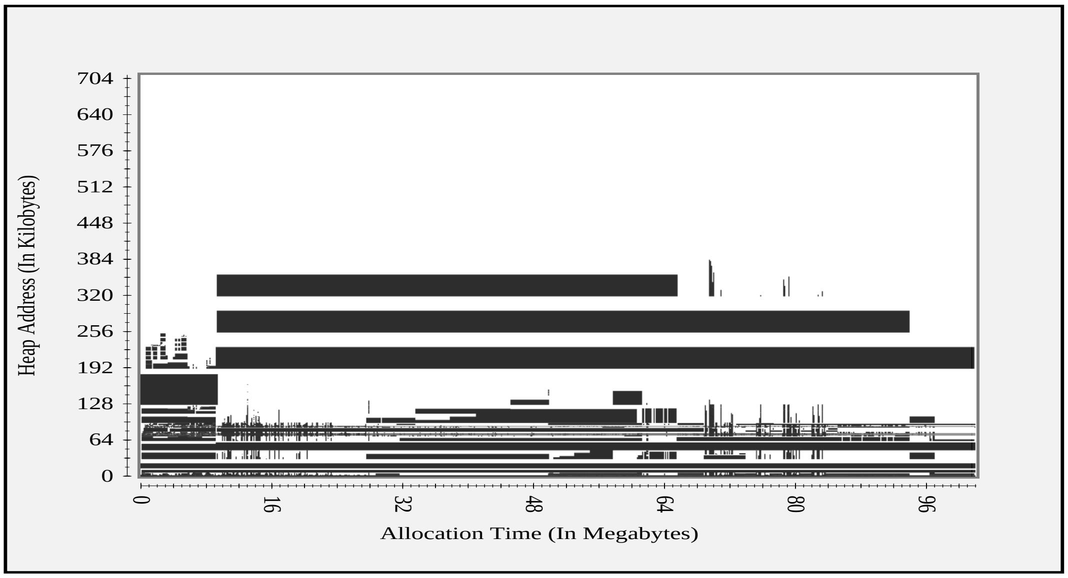

Figure C.11: Fragmentation plot for Espresso using the binary-buddy allocator (accounting for all overheads)

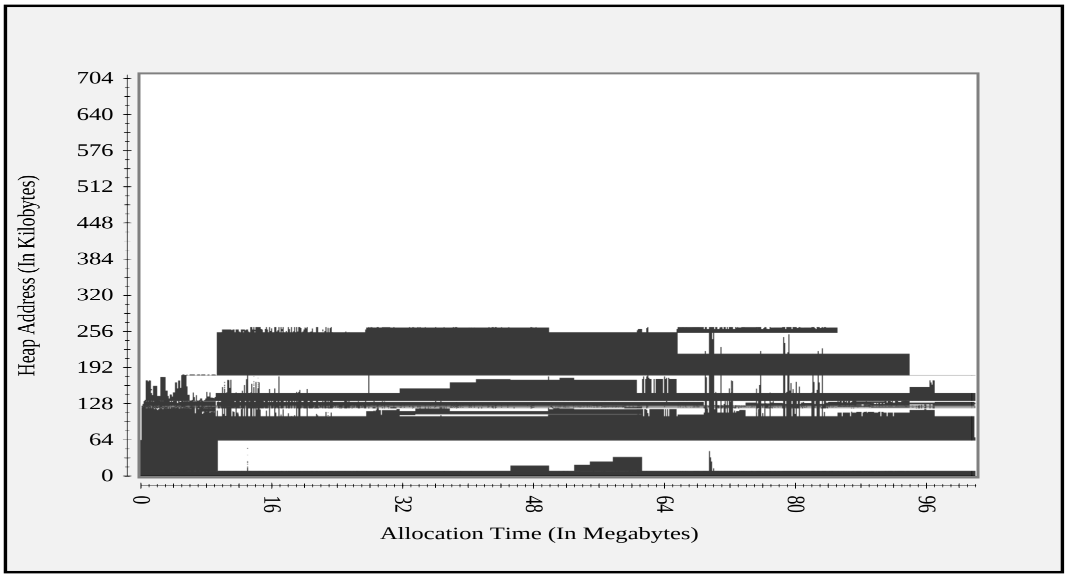

Figure C.12: Fragmentation plot for Espresso using the best-fit LIFO no-footer allocator (accounting for all overheads)

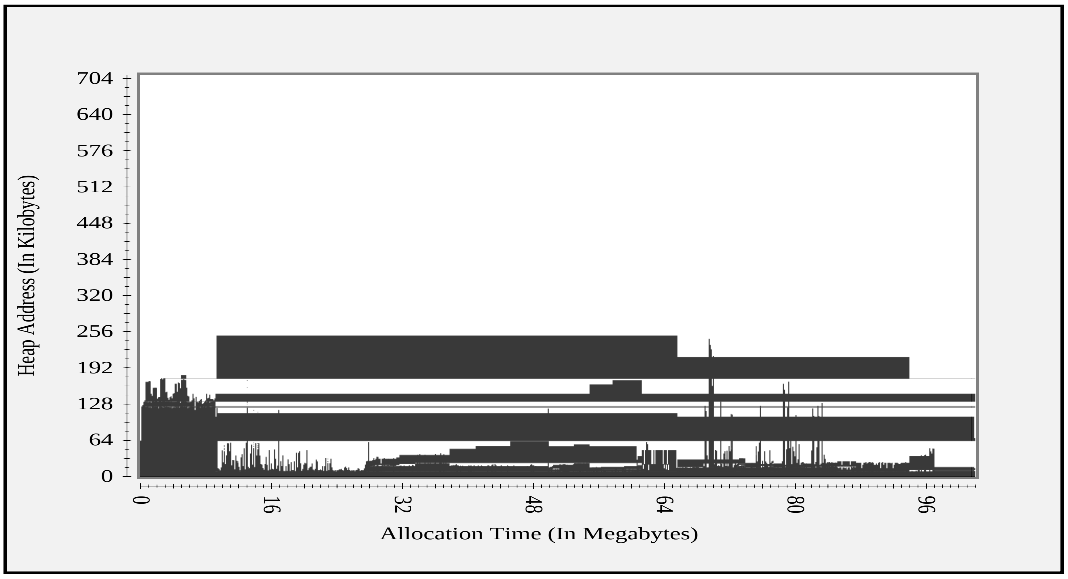

Figure C.13: Fragmentation plot for Espresso using the first-fit address-ordered no-footer allocator (accounting for all overheads)

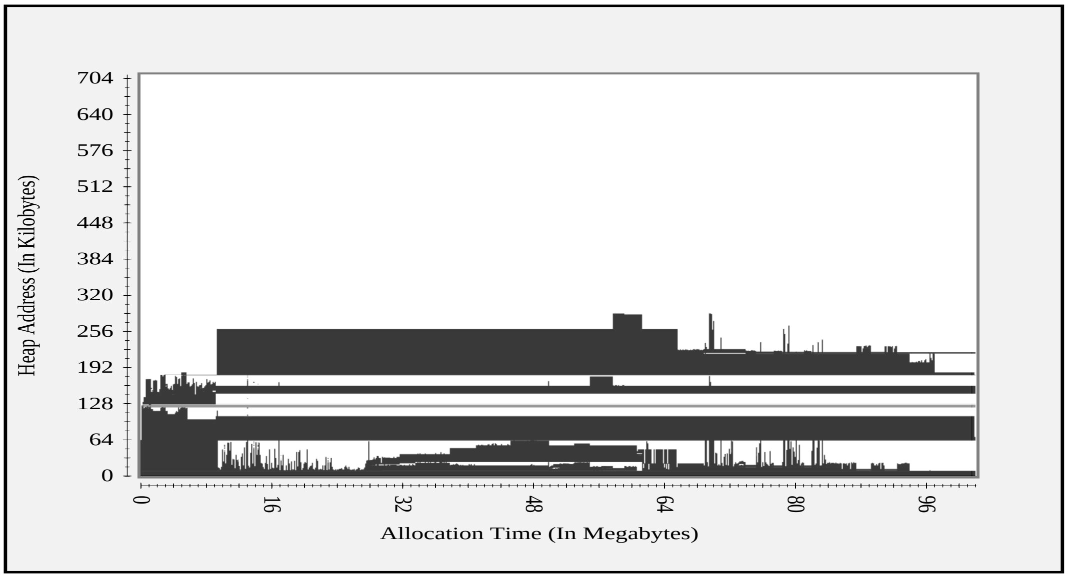

Figure C.14: Fragmentation plot for Espresso using the first-fit LIFO no-footer allocator (accounting for all overheads)

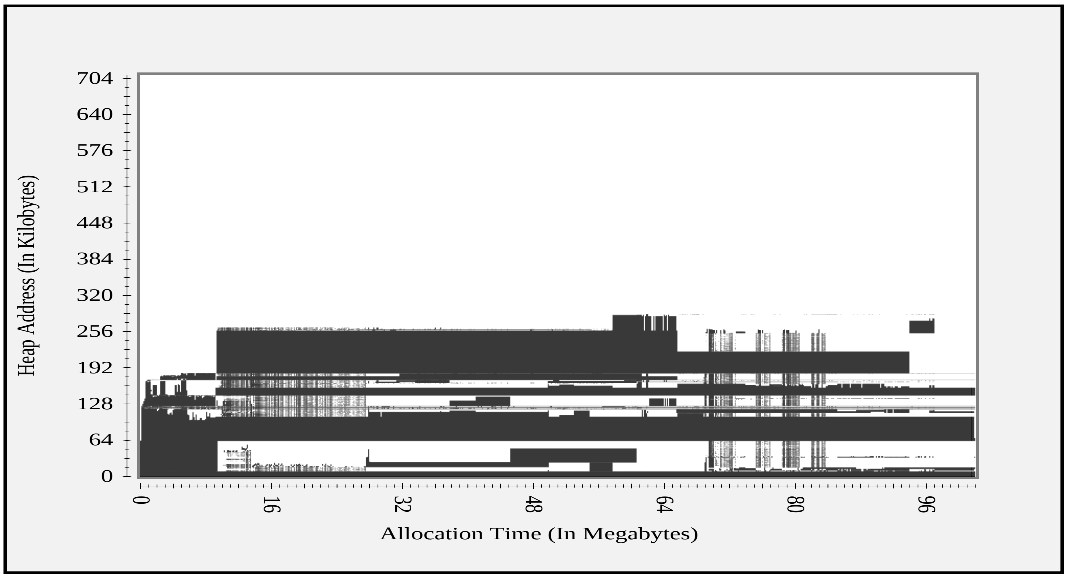

Figure C.15: Fragmentation plot for Espresso using the half-fit allocator (accounting for all overheads)

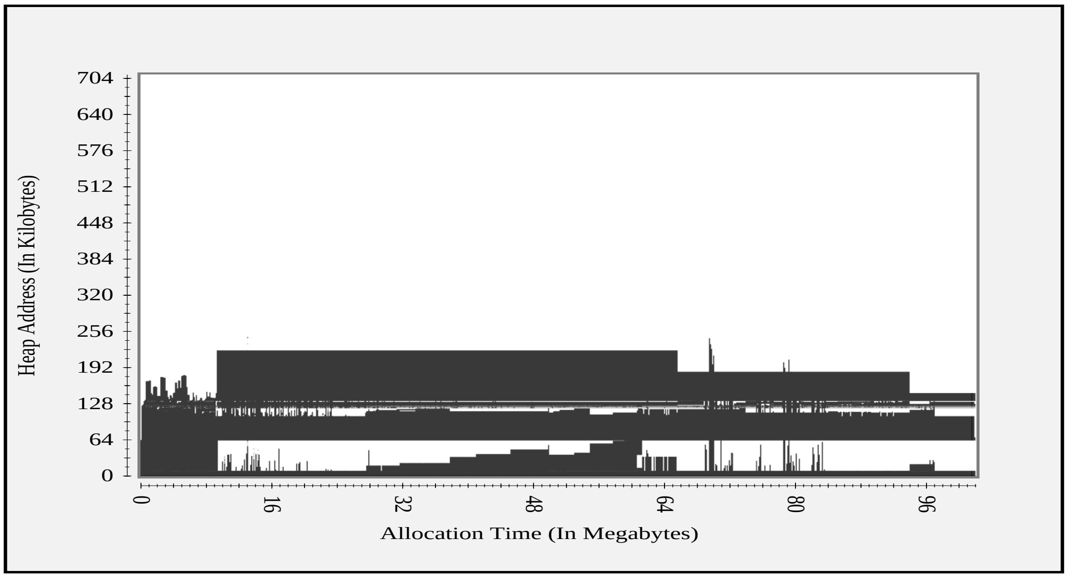

Figure C.16: Fragmentation plot for Espresso using Lea's 2.6.1 allocator (accounting for all overheads)

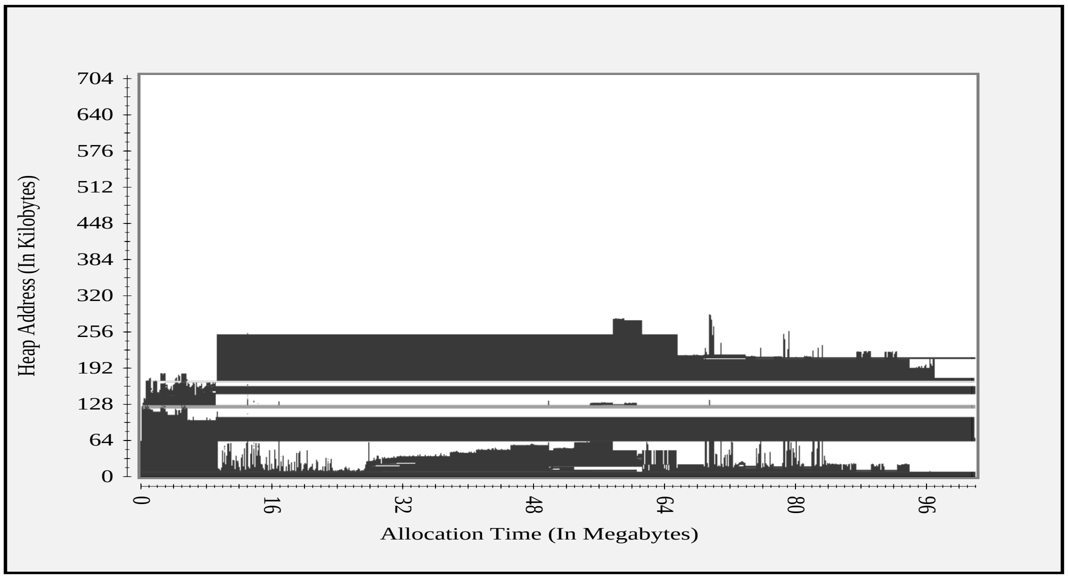

Figure C.17: Fragmentation plot for Espresso using the next-fit LIFO no-footer allocator (accounting for all overheads)

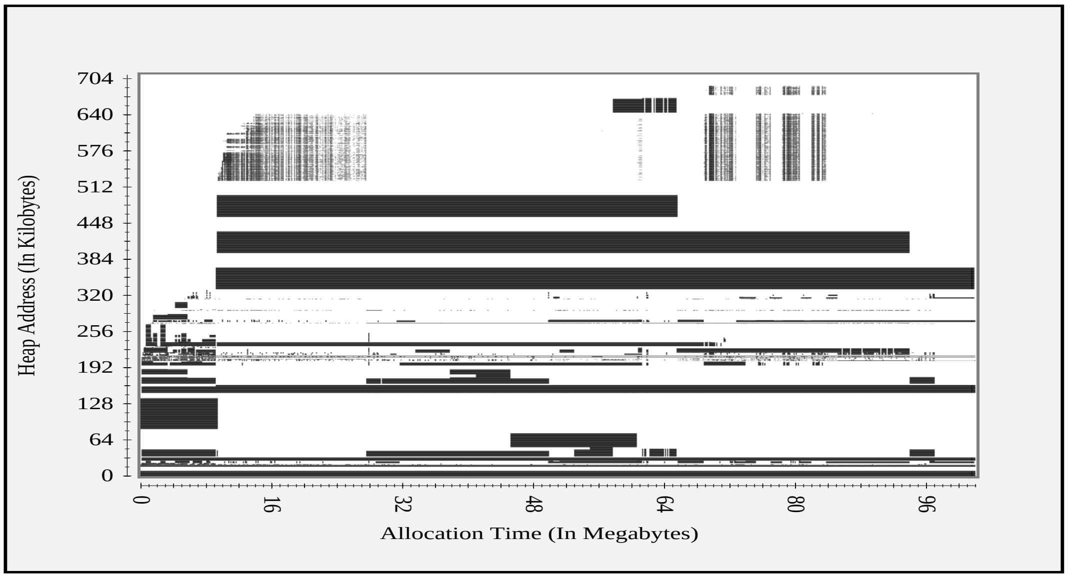

Figure C.18: Fragmentation plot for Espresso using the simple segregated storage 2^N allocator (accounting for all overheads)

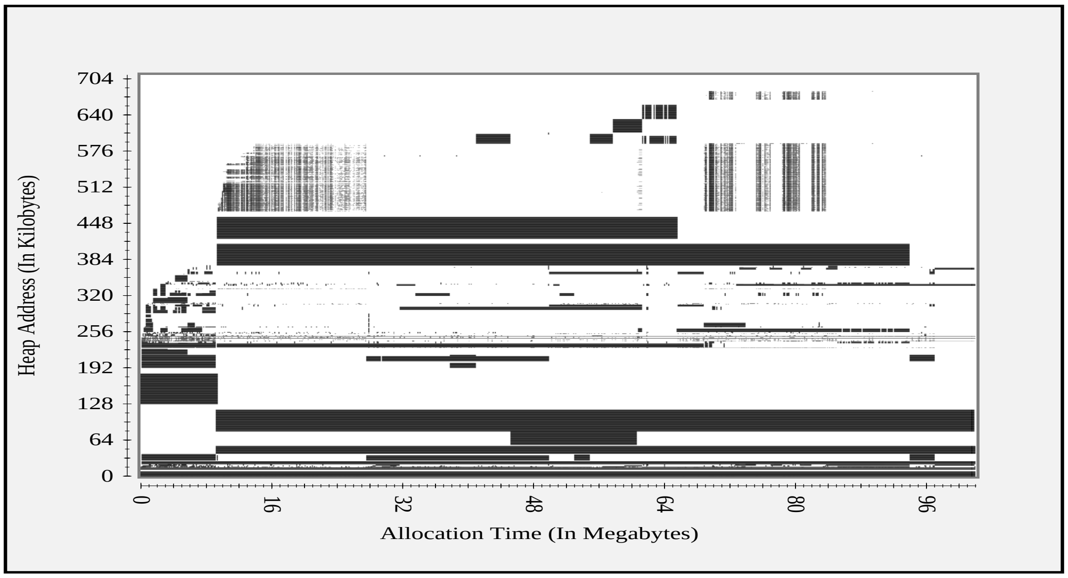

Figure C.19: Fragmentation plot for Espresso using the simple segregated storage 2^N & $3*2^N$ allocator (accounting for all overheads)

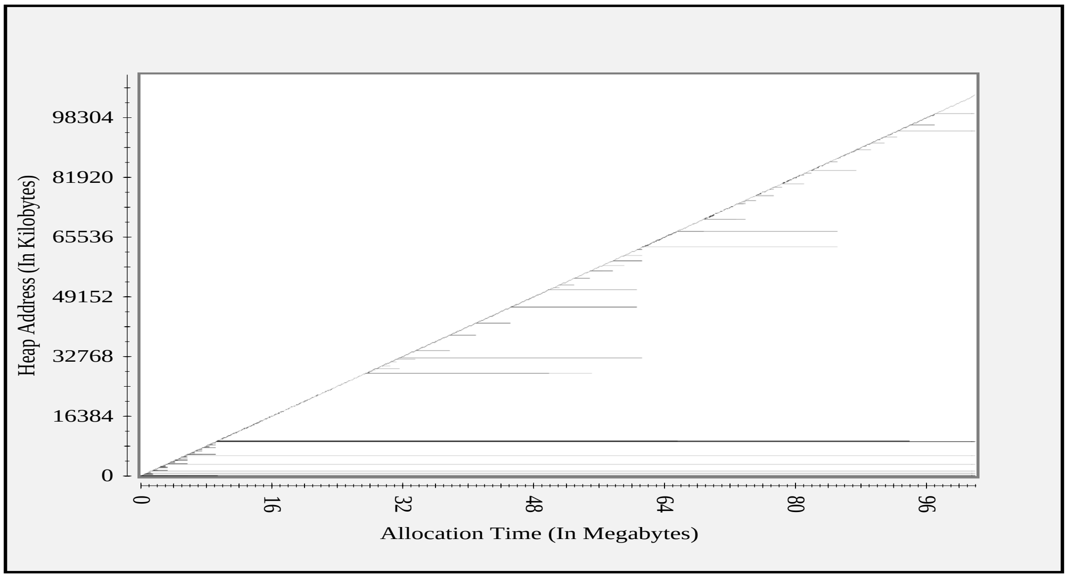

Figure C.20: Fragmentation plot for Espresso using the linear allocator

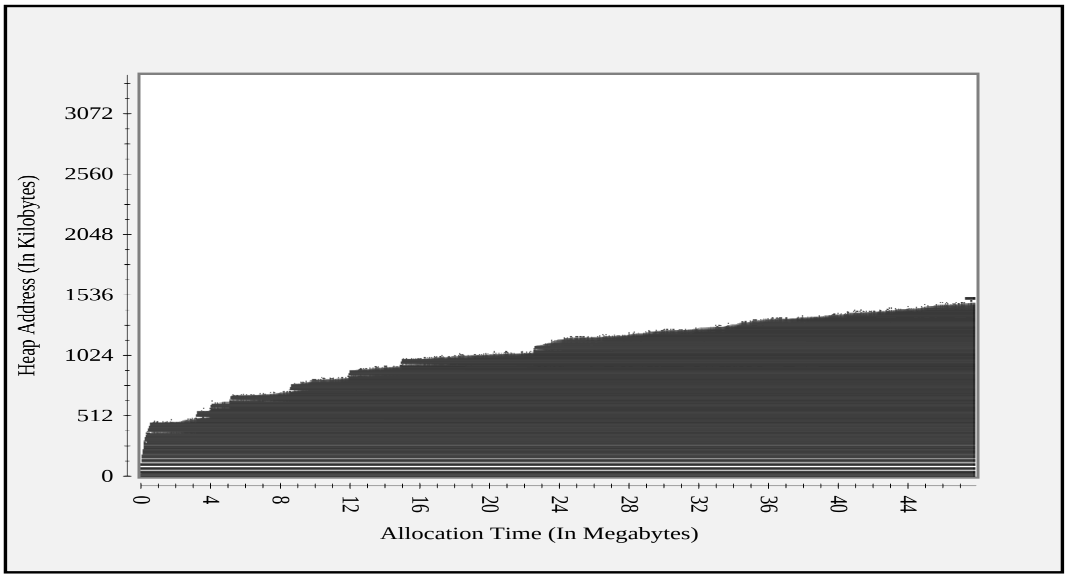

Figure C.21: Fragmentation plot for Ghostscript using the binary-buddy allocator (accounting for all overheads)

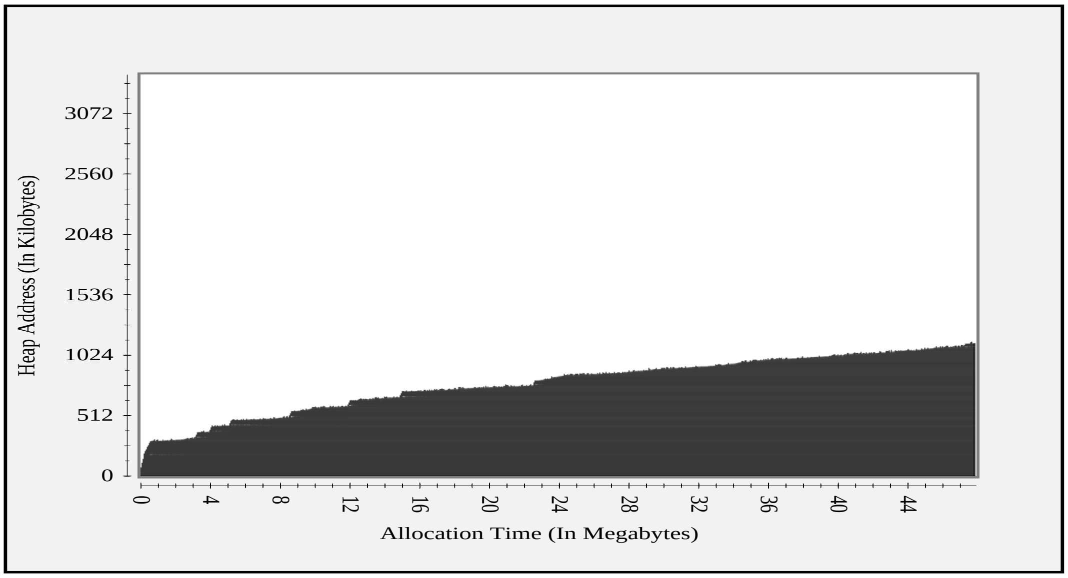

Figure C.22: Fragmentation plot for Ghostscript using the best-fit LIFO no-footer allocator (accounting for all overheads)

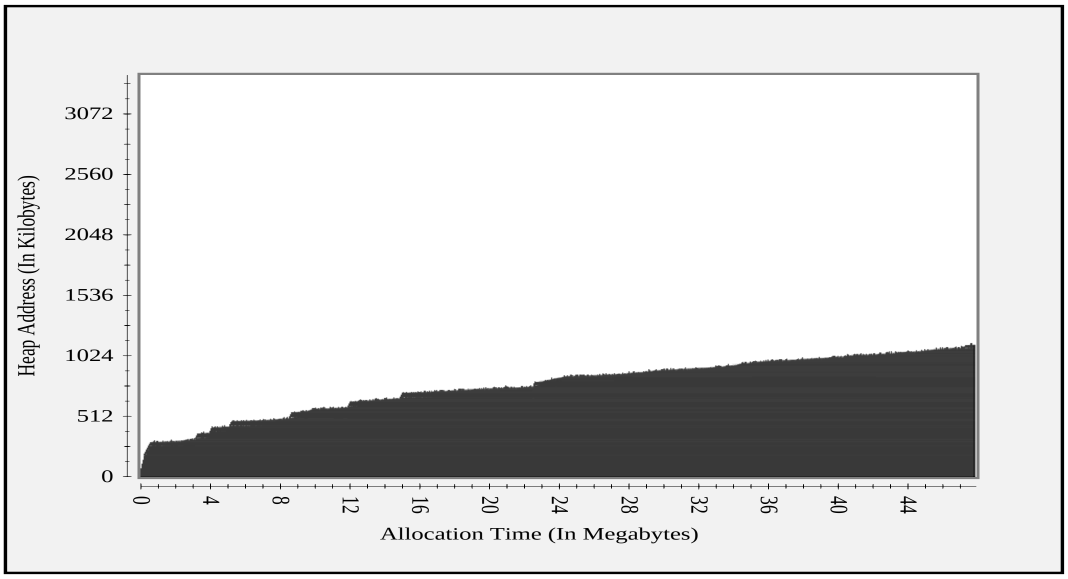

Figure C.23: Fragmentation plot for Ghostscript using the first-fit address-ordered no-footer allocator (accounting for all overheads)

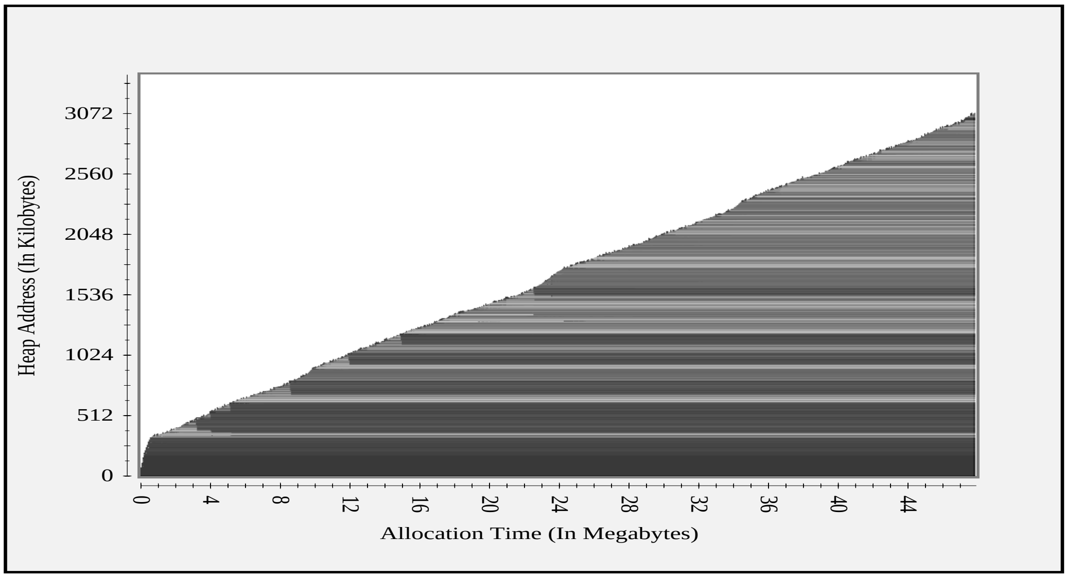

Figure C.24: Fragmentation plot for Ghostscript using the first-fit LIFO no-footer allocator (accounting for all overheads)

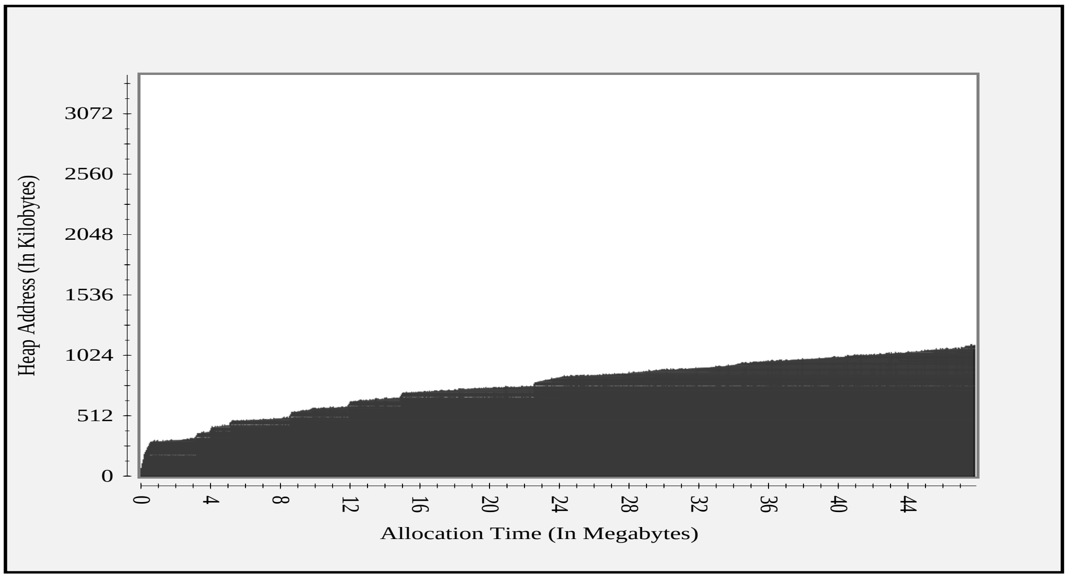

Figure C.25: Fragmentation plot for Ghostscript using the half-fit allocator (accounting for all overheads)

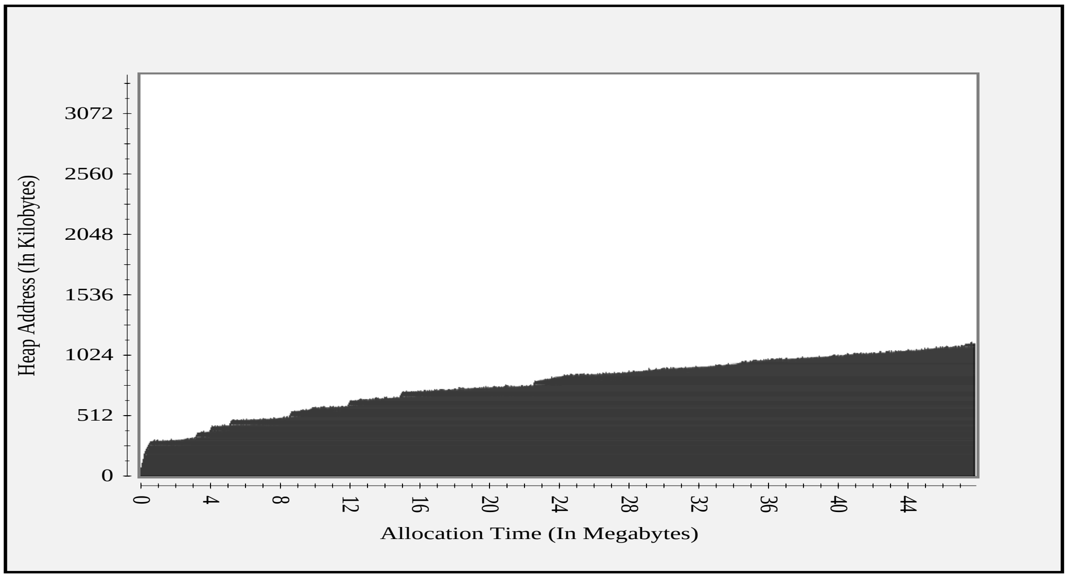

Figure C.26: Fragmentation plot for Ghostscript using Lea's 2.6.1 allocator (accounting for all overheads)

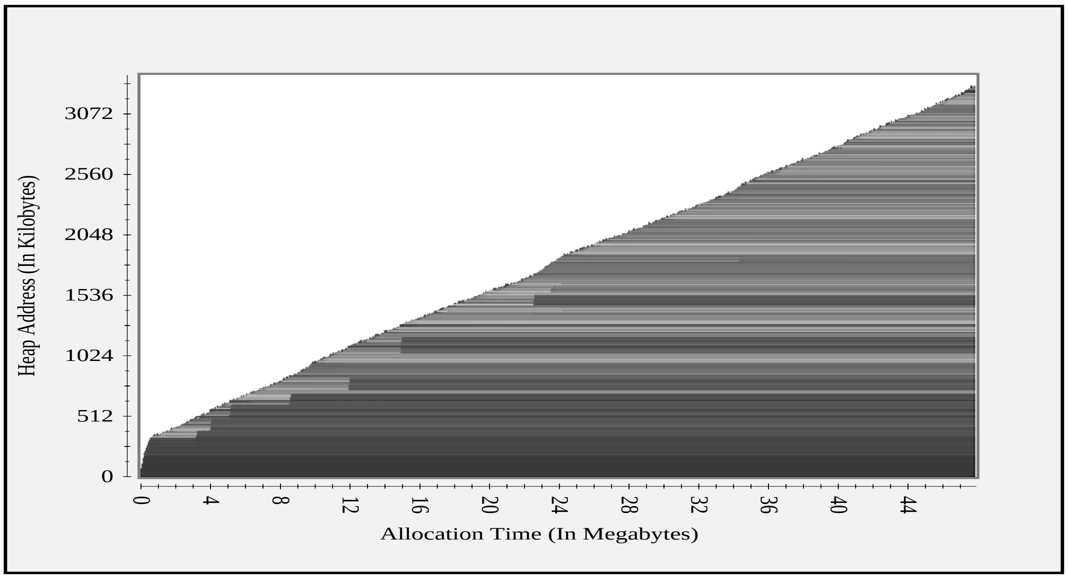

Figure C.27: Fragmentation plot for Ghostscript using the next-fit LIFO no-footer allocator (accounting for all overheads)

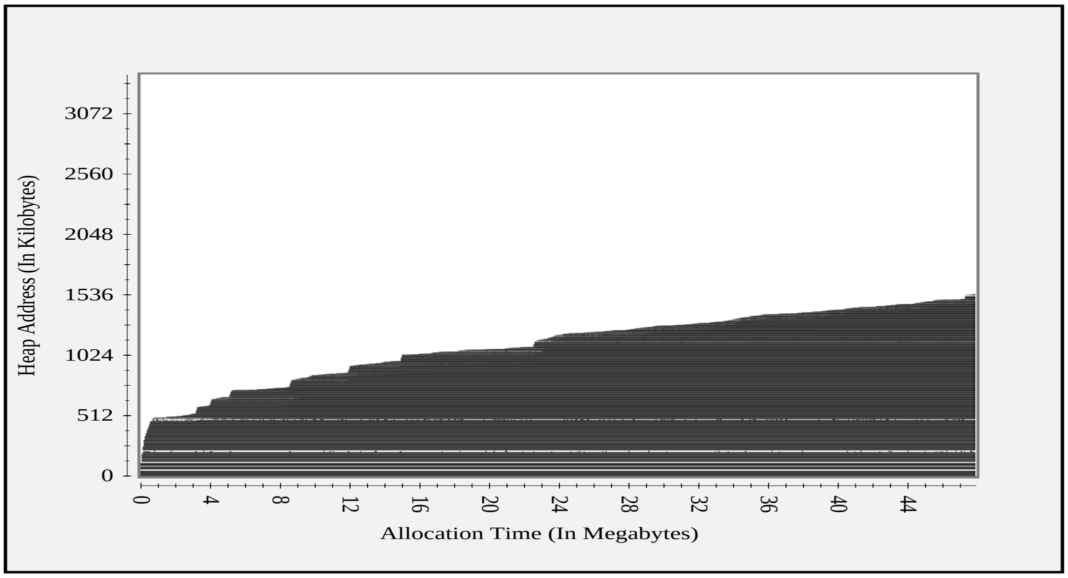

Figure C.28: Fragmentation plot for Ghostscript using the simple segregated storage 2^N allocator (accounting for all overheads)

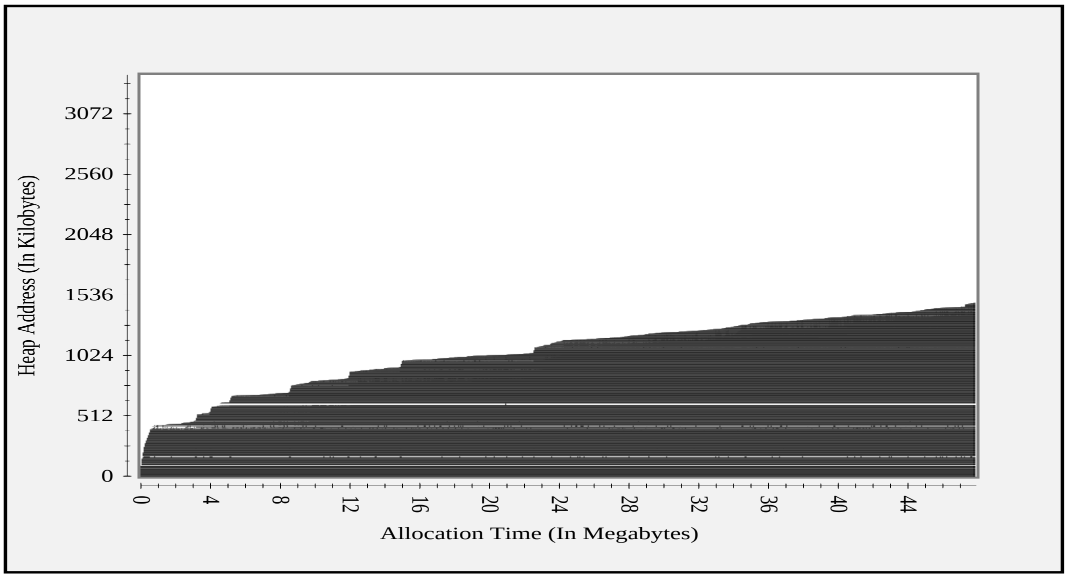

Figure C.29: Fragmentation plot for Ghostscript using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

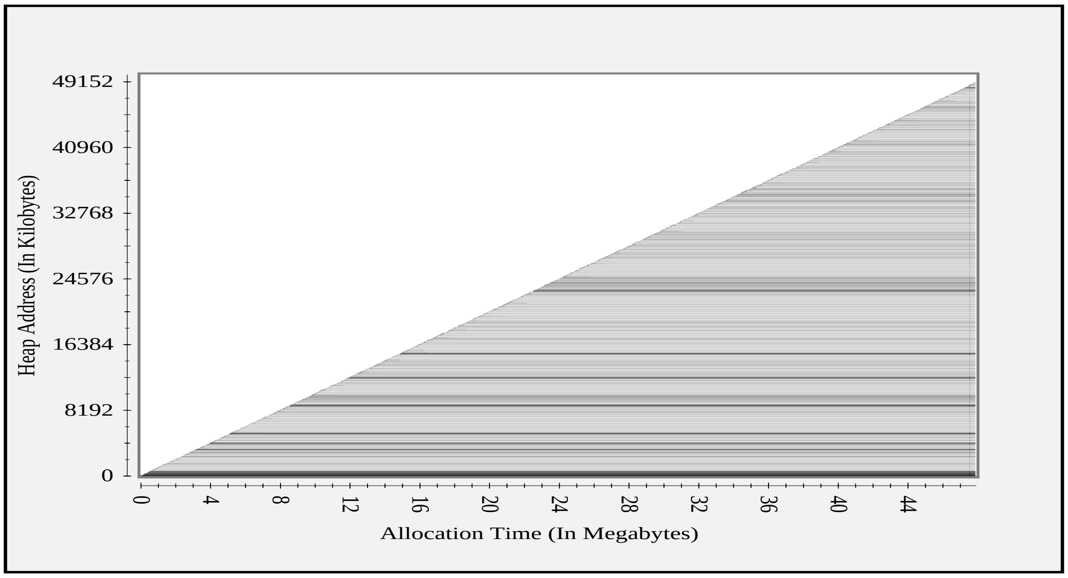

Figure C.30: Fragmentation plot for Ghostscript using the linear allocator

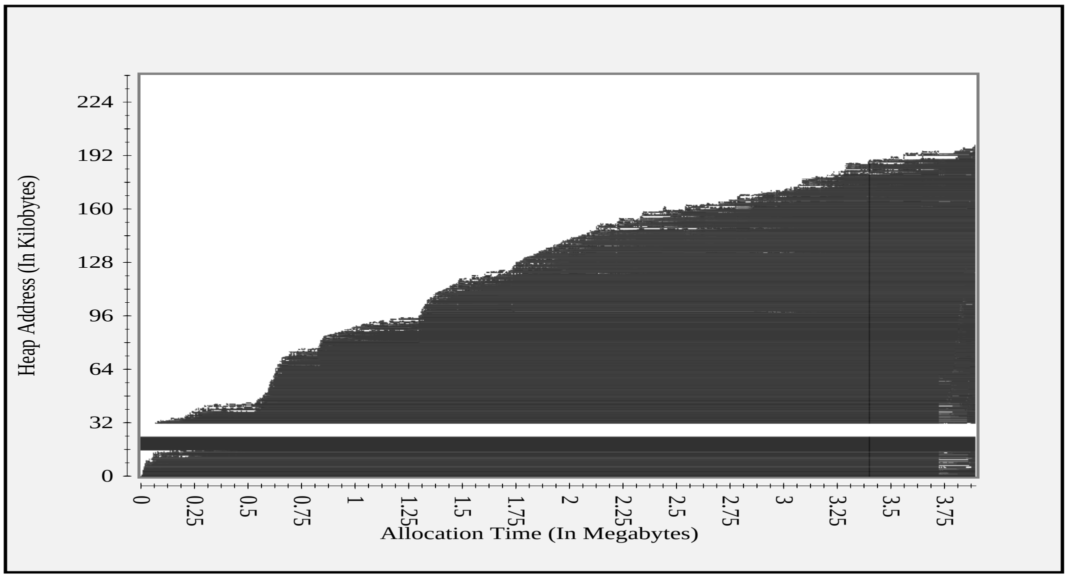

Figure C.31: Fragmentation plot for Grobner using the binary-buddy allocator (accounting for all overheads)

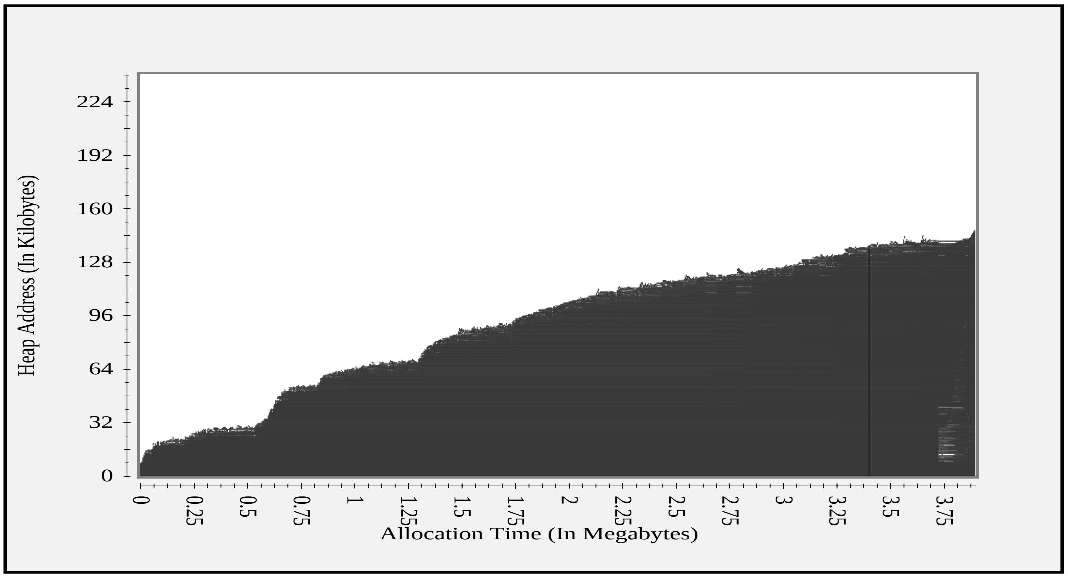

Figure C.32: Fragmentation plot for Grobner using the best-fit LIFO no-footer allocator (accounting for all overheads)

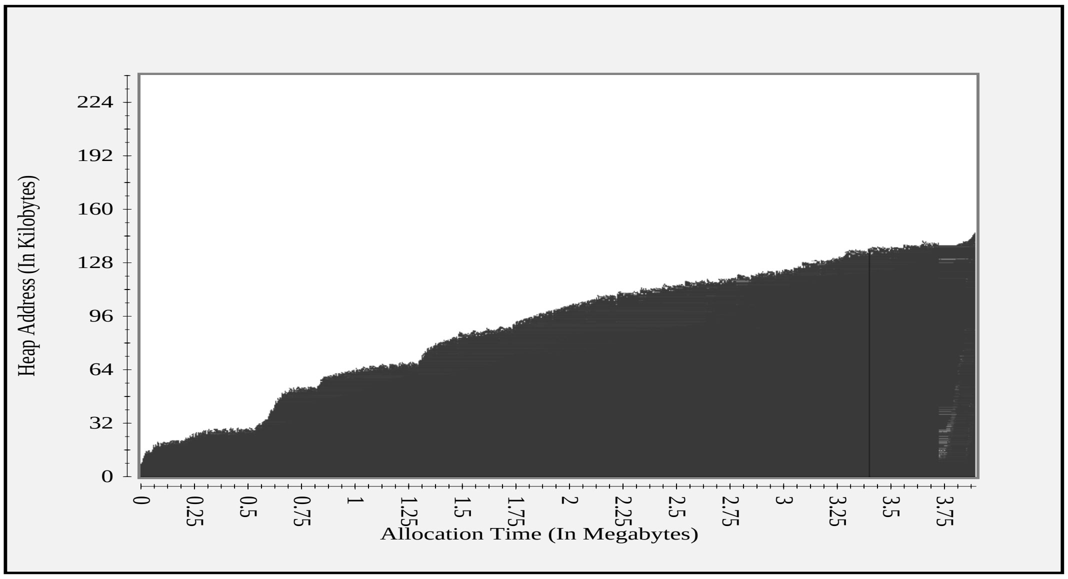

Figure C.33: Fragmentation plot for Grobner using the first-fit address-ordered no-footer allocator (accounting for all overheads)

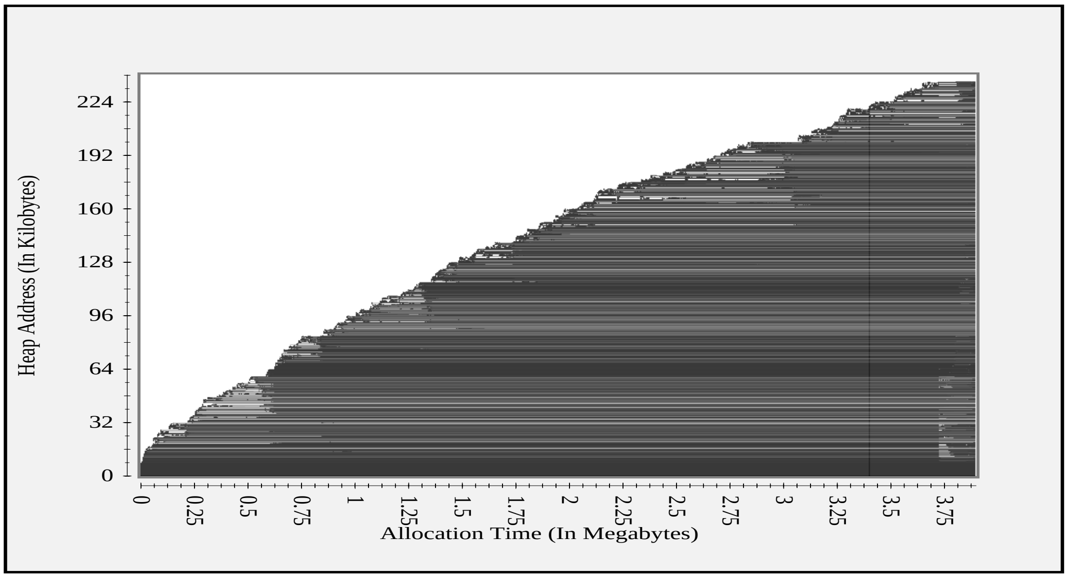

Figure C.34: Fragmentation plot for Grobner using the first-fit LIFO no-footer allocator (accounting for all overheads)

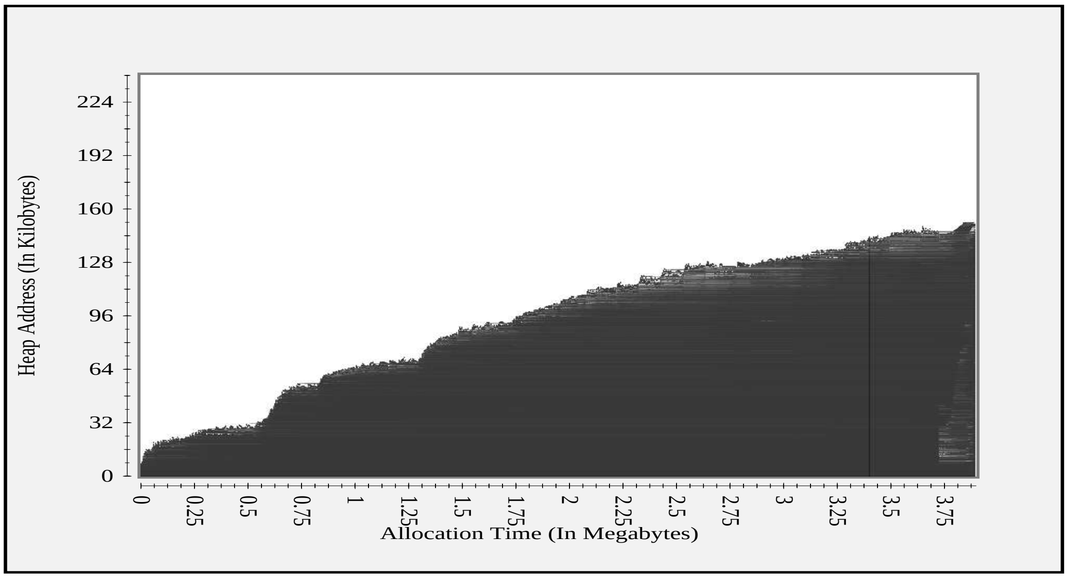

Figure C.35: Fragmentation plot for Grobner using the half-fit allocator (accounting for all overheads)

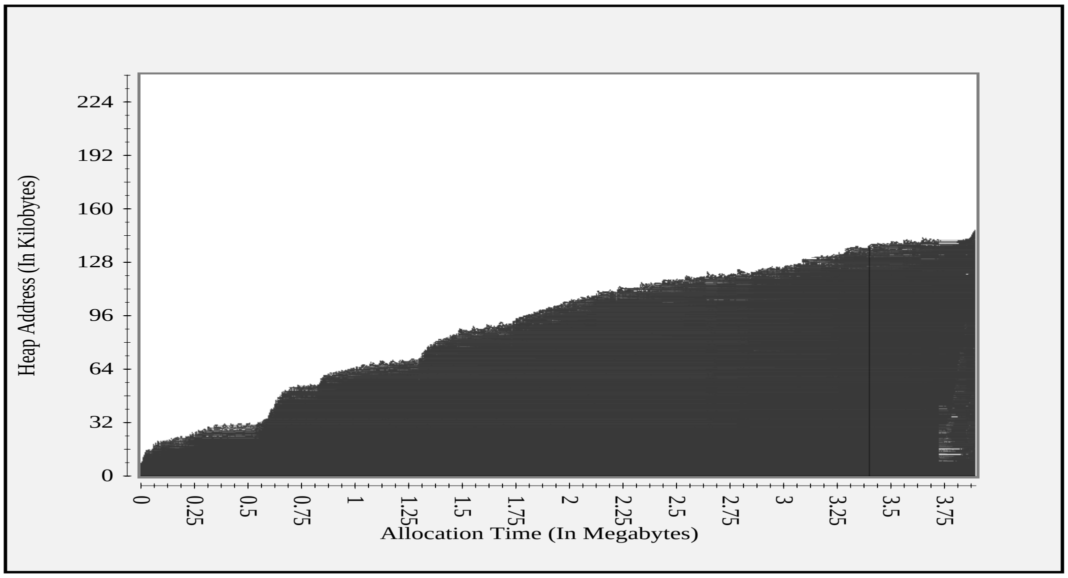

Figure C.36: Fragmentation plot for Grobner using Lea's 2.6.1 allocator (accounting for all overheads)

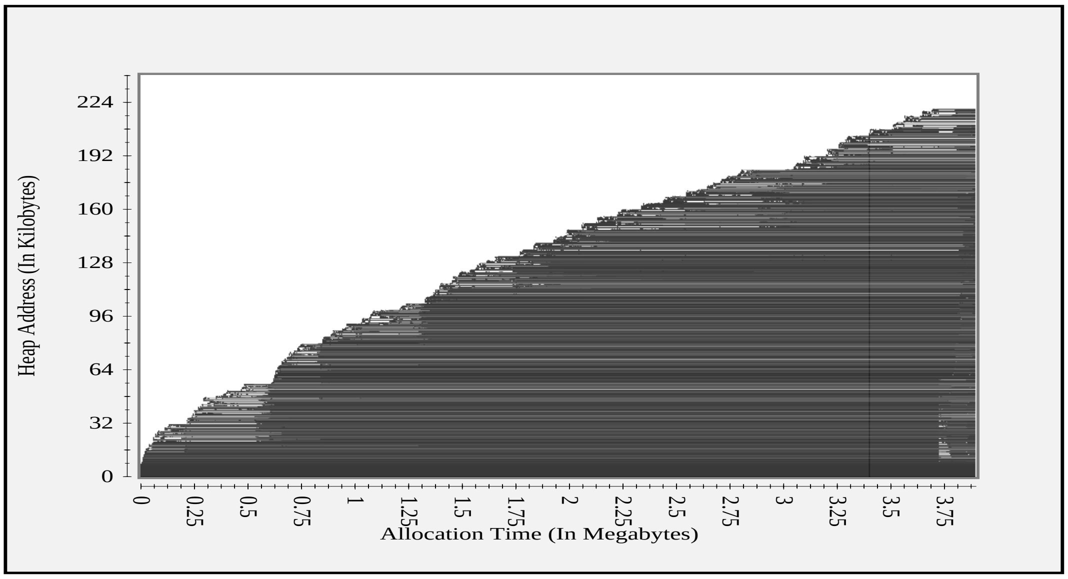

Figure C.37: Fragmentation plot for Grobner using the next-fit LIFO no-footer allocator (accounting for all overheads)

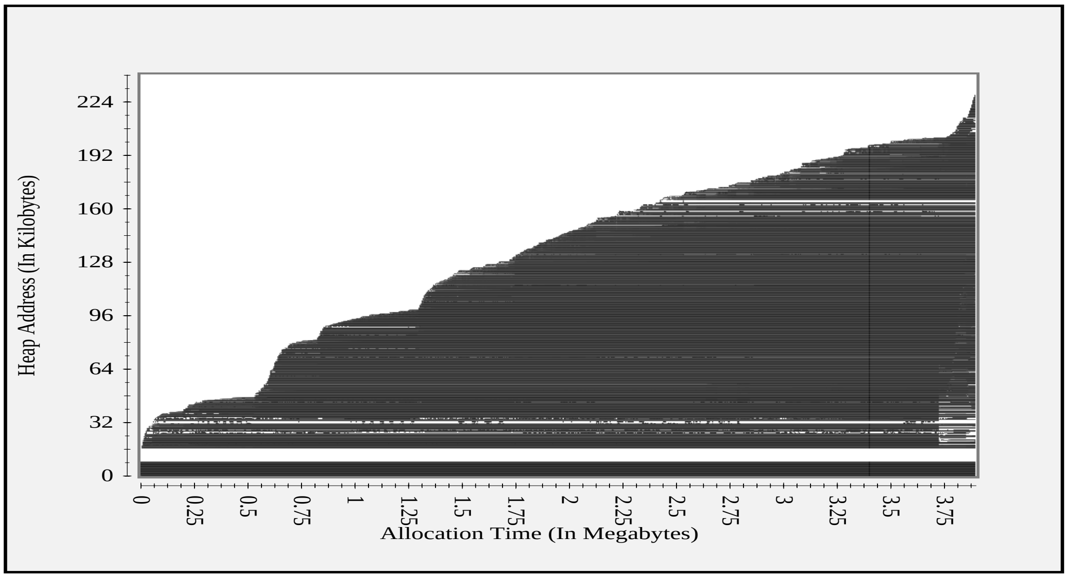

Figure C.38: Fragmentation plot for Grobner using the simple segregated storage 2^N allocator (accounting for all overheads)

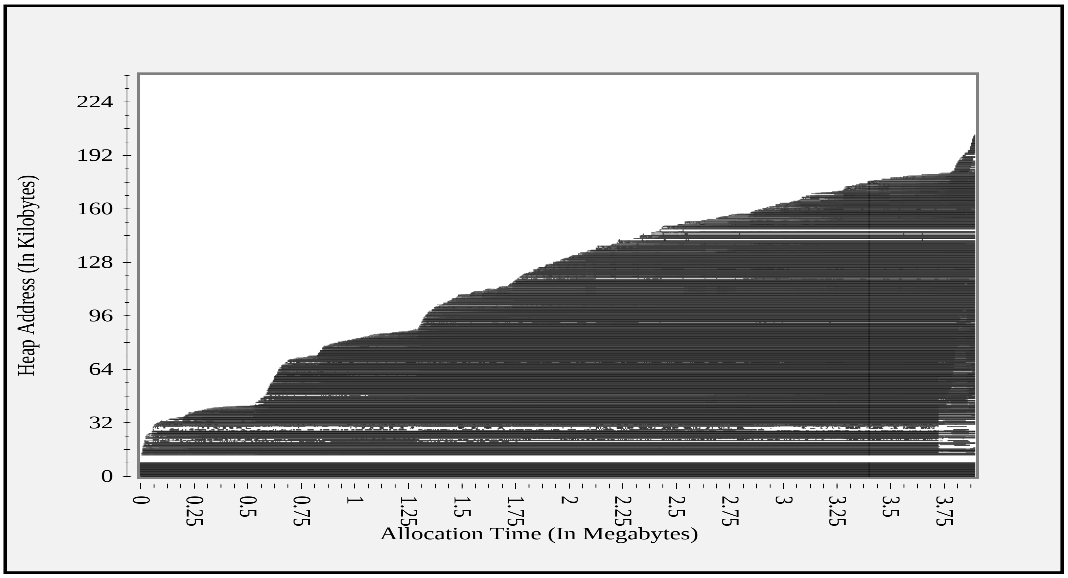

Figure C.39: Fragmentation plot for Grobner using the simple segregated storage 2^N & $3*2^N$ allocator (accounting for all overheads)

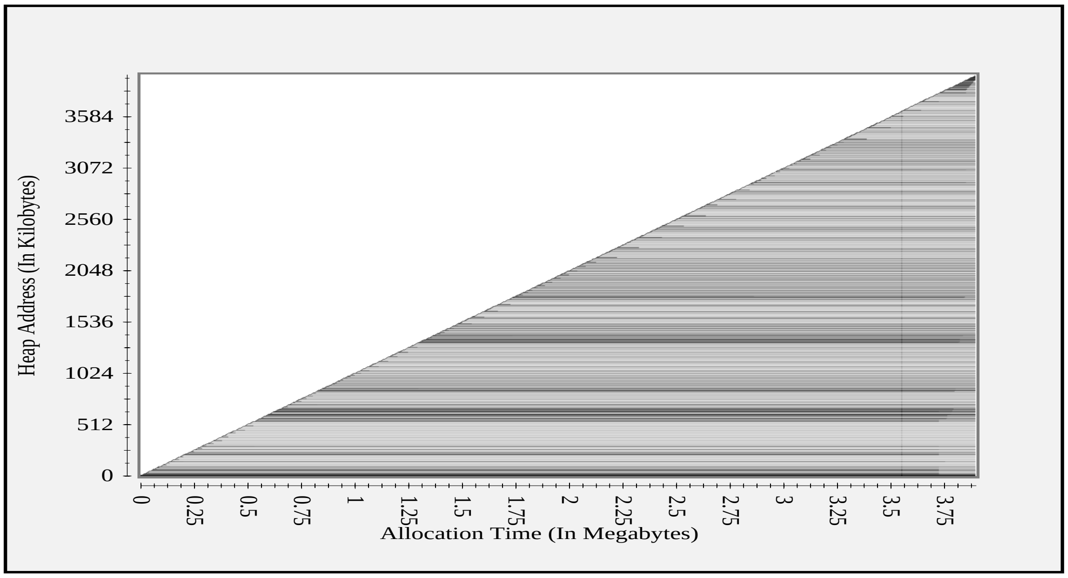

Figure C.40: Fragmentation plot for Grobner using the linear allocator

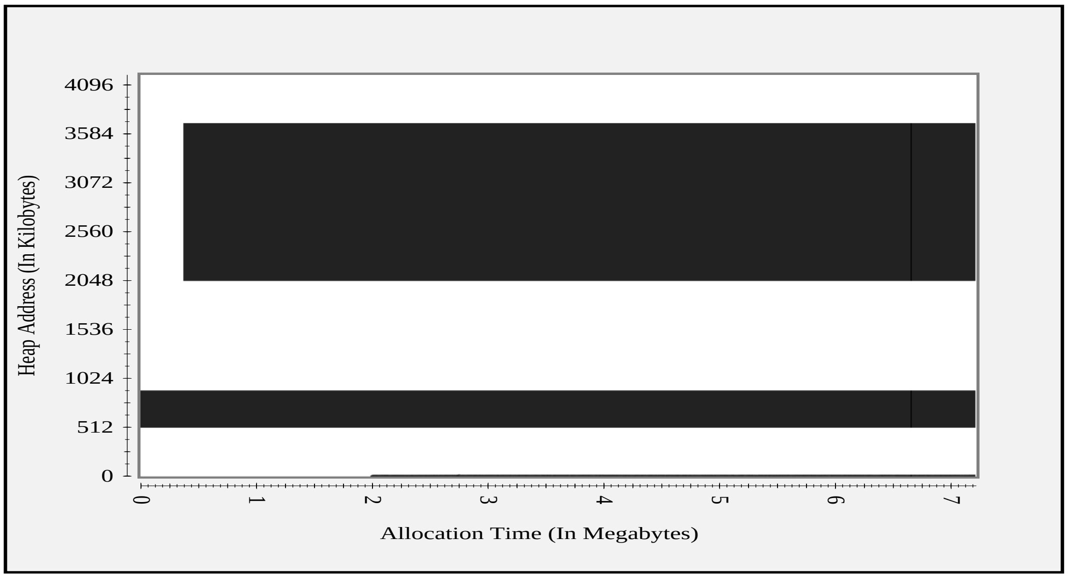

Figure C.41: Fragmentation plot for Hyper using the binary-buddy allocator (accounting for all overheads)

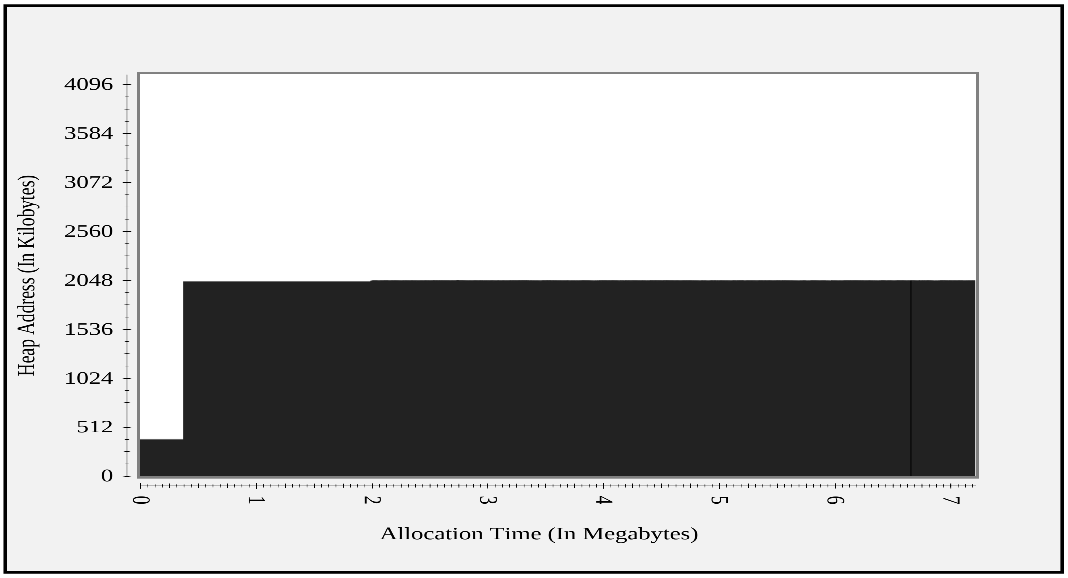

Figure C.42: Fragmentation plot for Hyper using the best-fit LIFO no-footer allocator (accounting for all overheads)

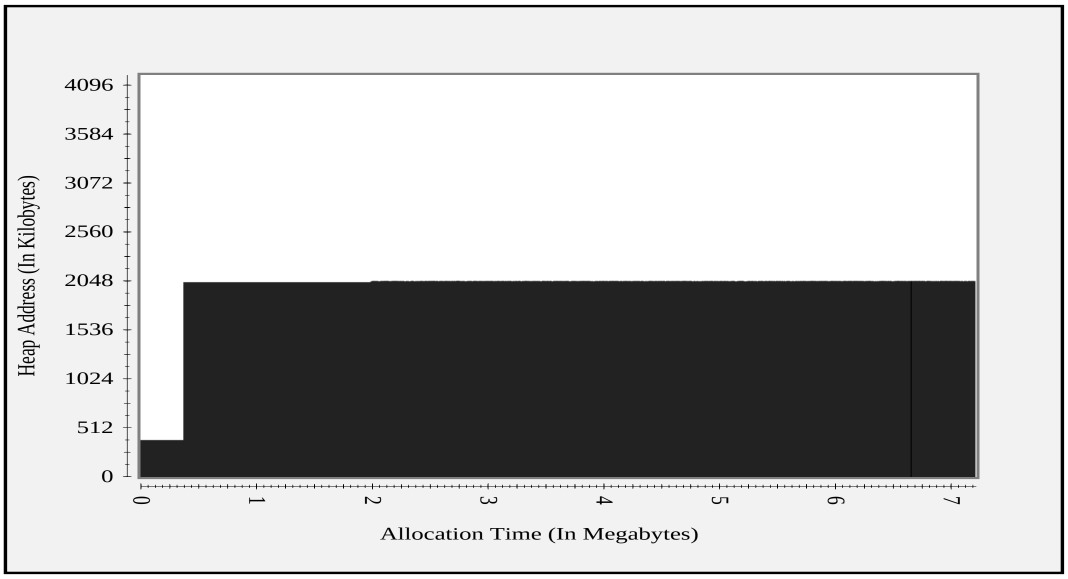

Figure C.43: Fragmentation plot for Hyper using the first-fit address-ordered no-footer allocator (accounting for all overheads)

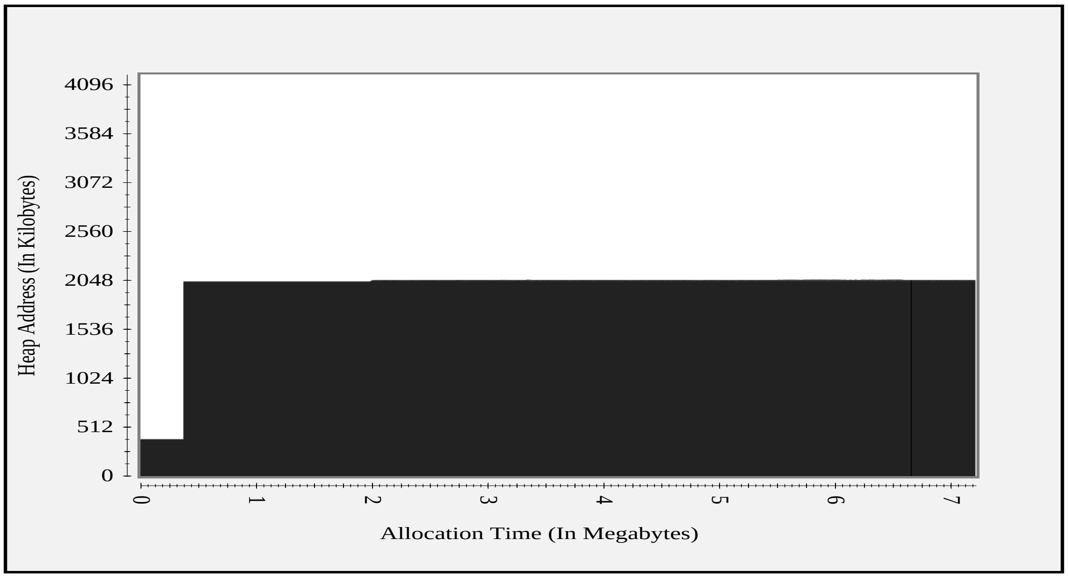

Figure C.44: Fragmentation plot for Hyper using the first-fit LIFO no-footer allocator (accounting for all overheads)

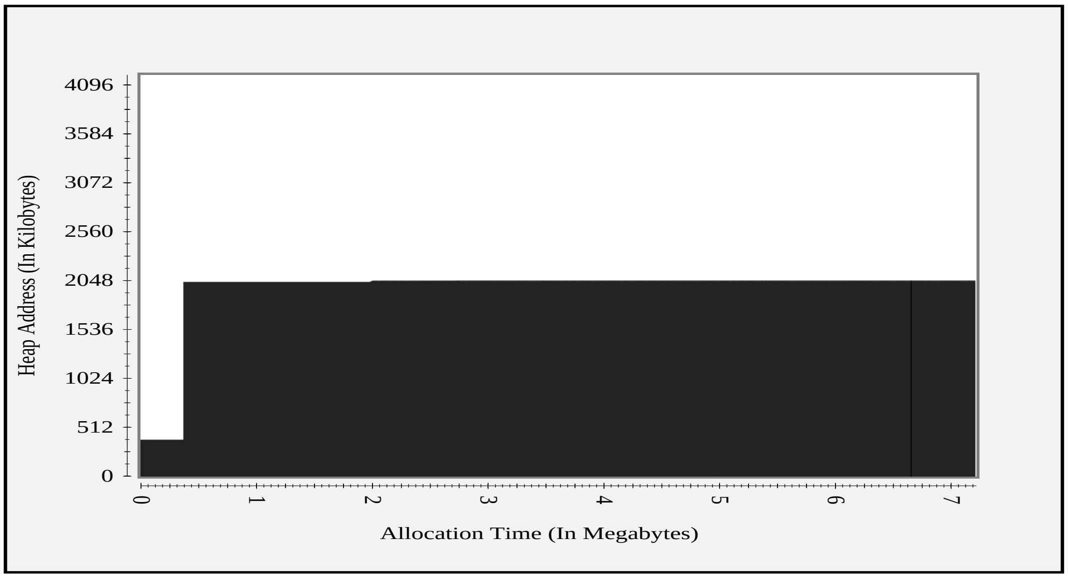

Figure C.45: Fragmentation plot for Hyper using the half-fit allocator (accounting for all overheads)

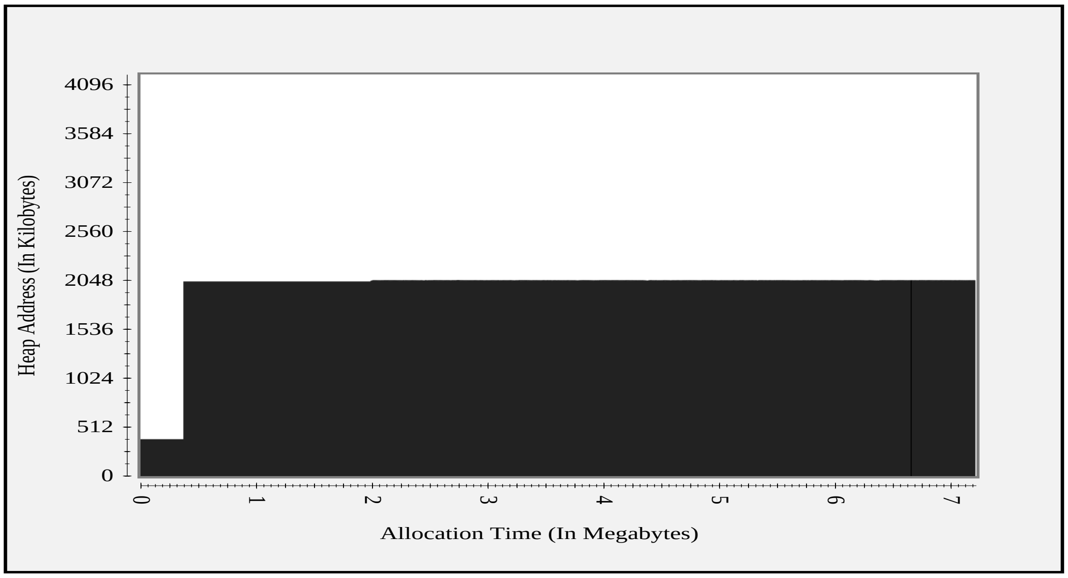

Figure C.46: Fragmentation plot for Hyper using Lea's 2.6.1 allocator (accounting for all overheads)

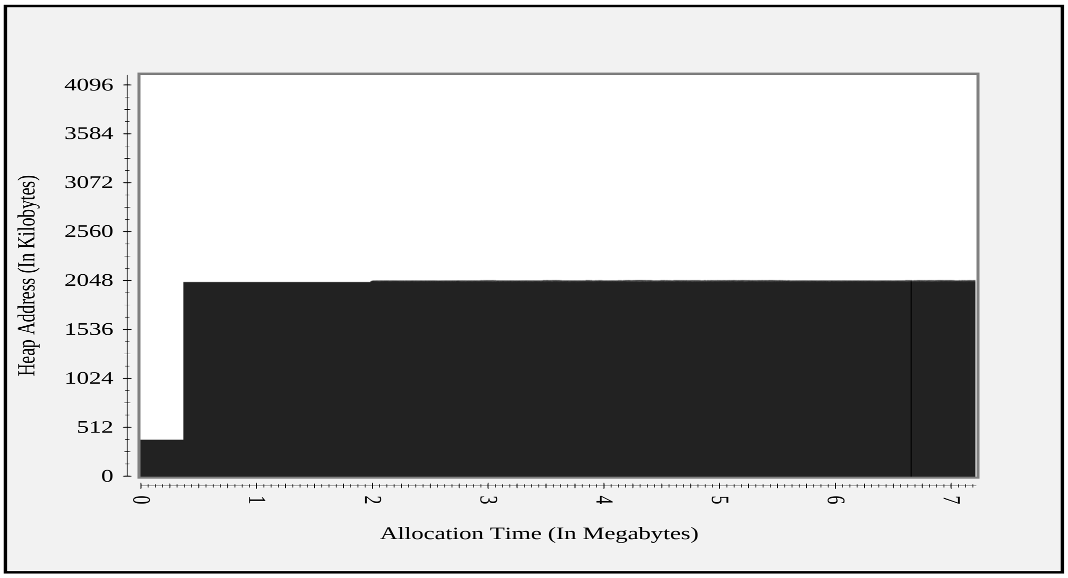

Figure C.47: Fragmentation plot for Hyper using the next-fit LIFO no-footer allocator (accounting for all overheads)

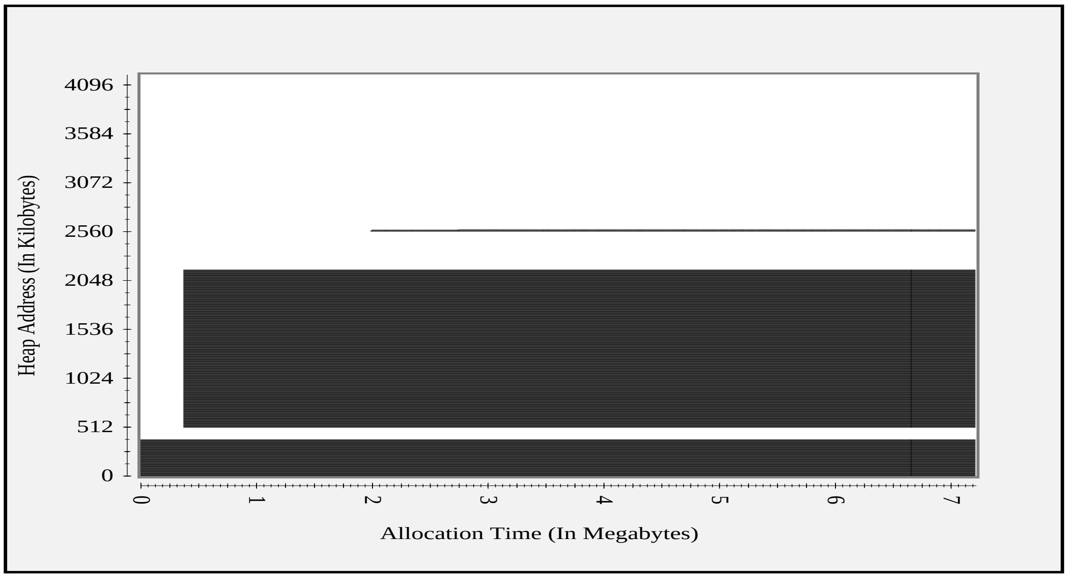

Figure C.48: Fragmentation plot for Hyper using the simple segregated storage 2^N allocator (accounting for all overheads)

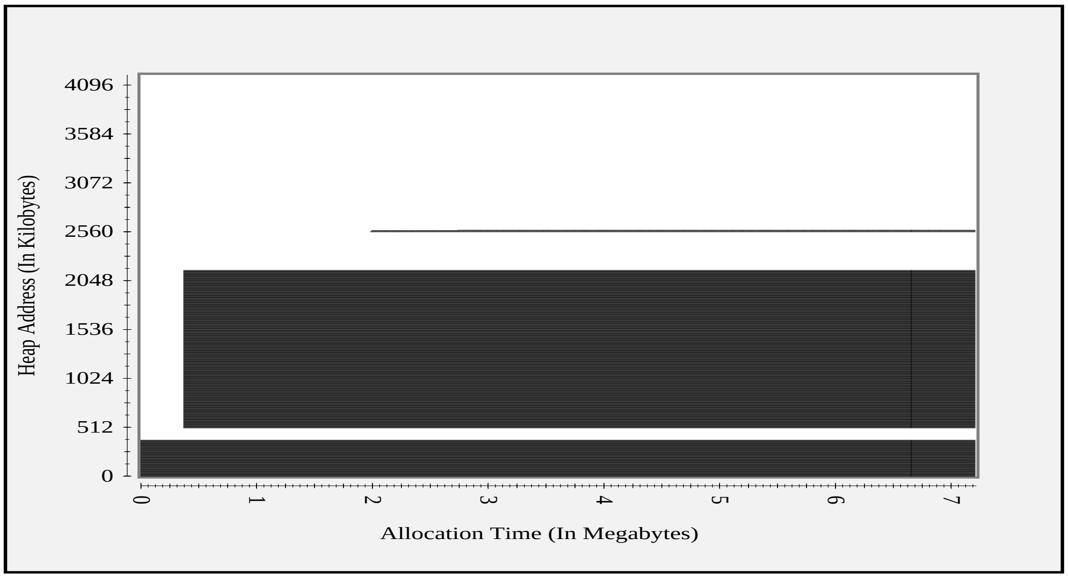

Figure C.49: Fragmentation plot for Hyper using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

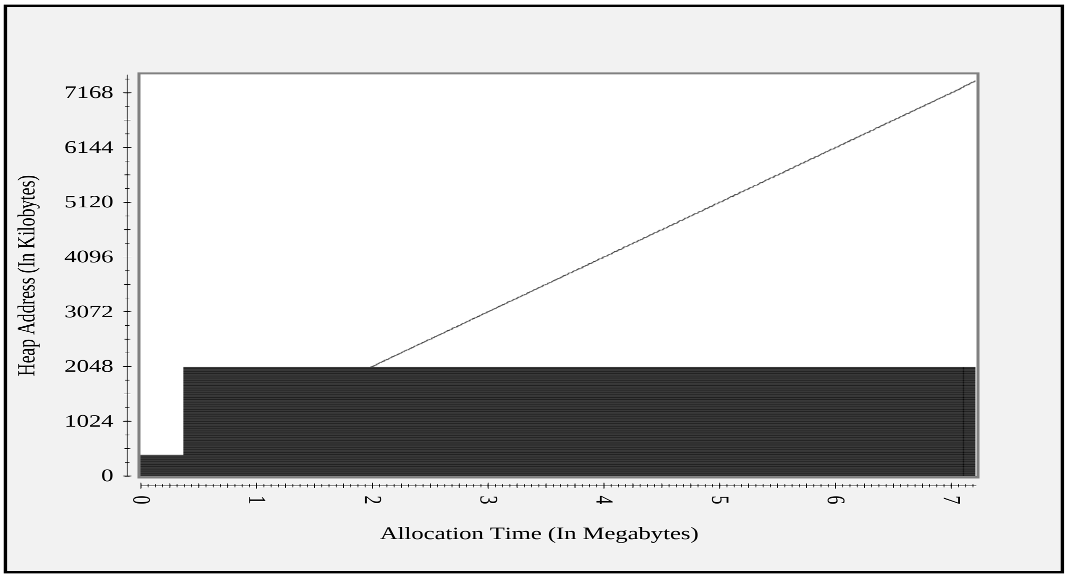

Figure C.50: Fragmentation plot for Hyper using the linear allocator

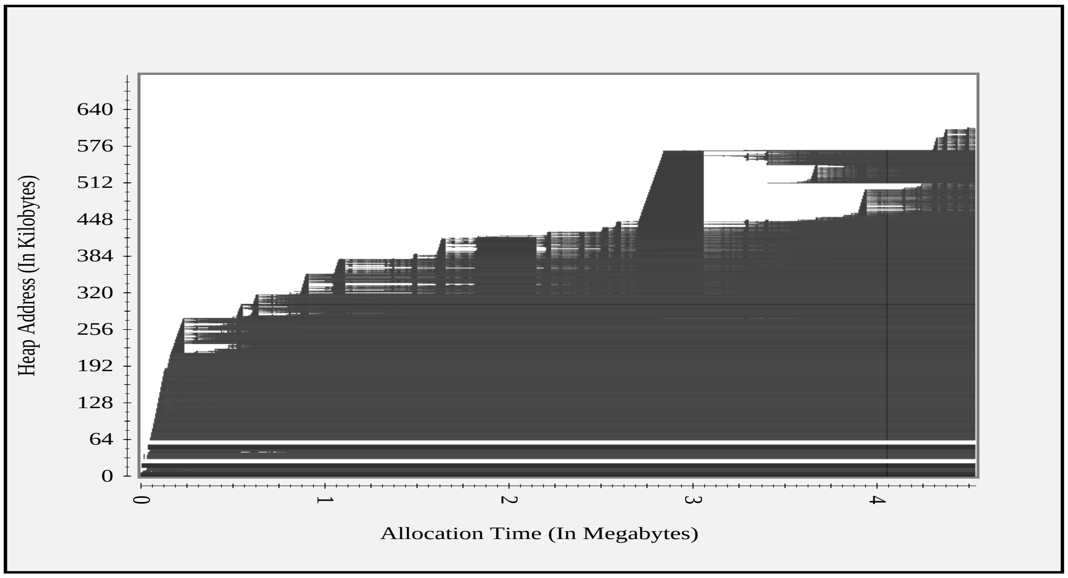

Figure C.51: Fragmentation plot for P2C using the binary-buddy allocator (accounting for all overheads)

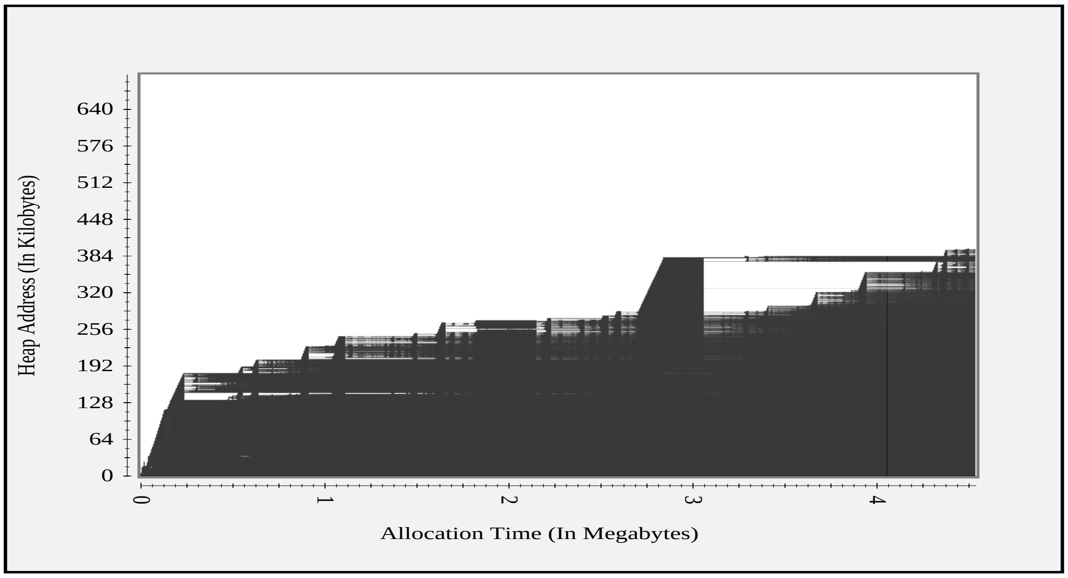

Figure C.52: Fragmentation plot for P2C using the best-fit LIFO no-footer allocator (accounting for all overheads)

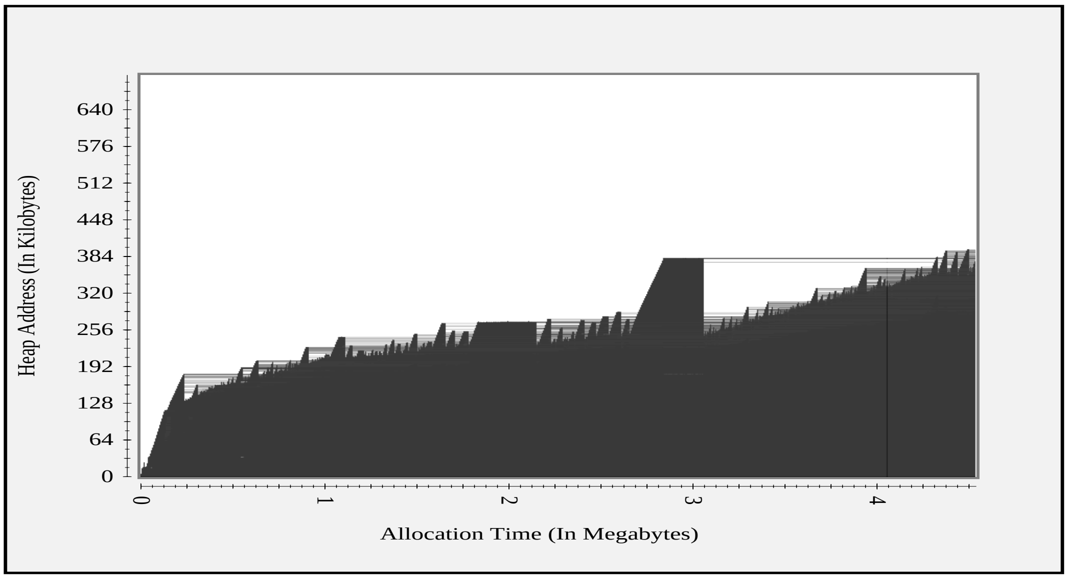

Figure C.53: Fragmentation plot for P2C using the first-fit address-ordered no-footer allocator (accounting for all overheads)

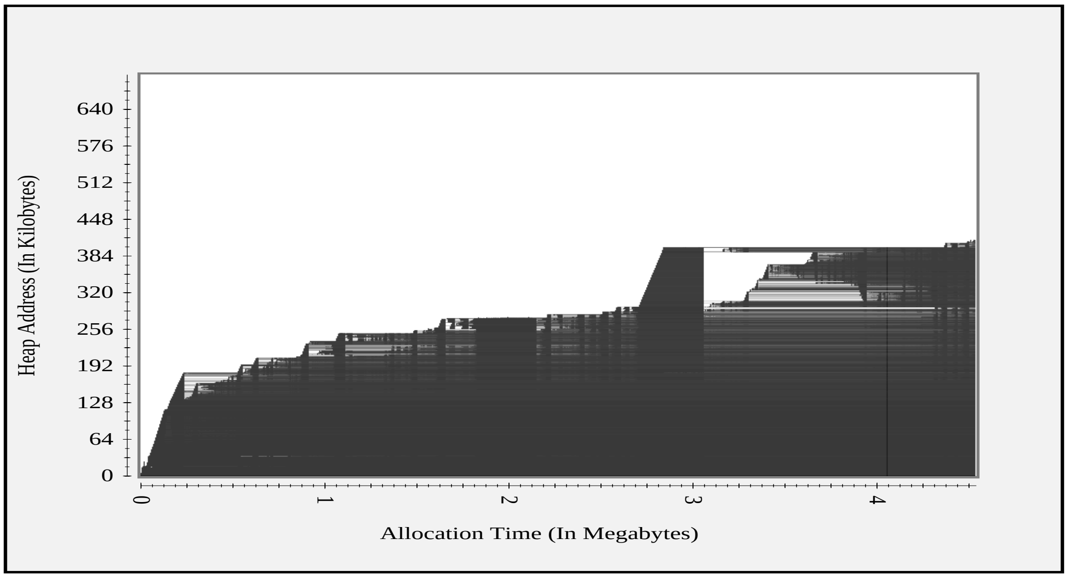

Figure C.54: Fragmentation plot for P2C using the first-fit LIFO no-footer allocator (accounting for all overheads)

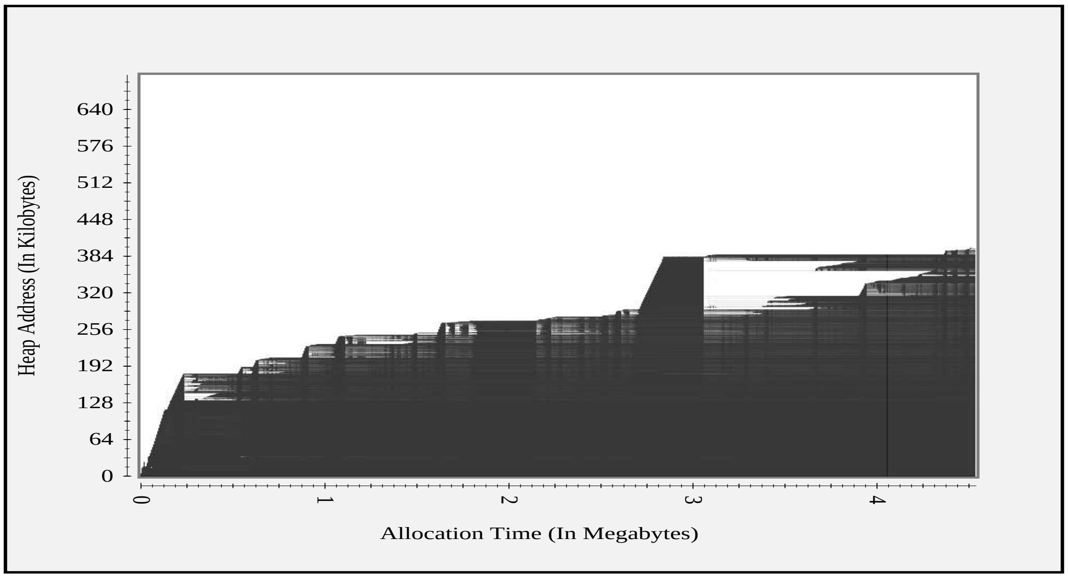

Figure C.55: Fragmentation plot for P2C using the half-fit allocator (accounting for all overheads)

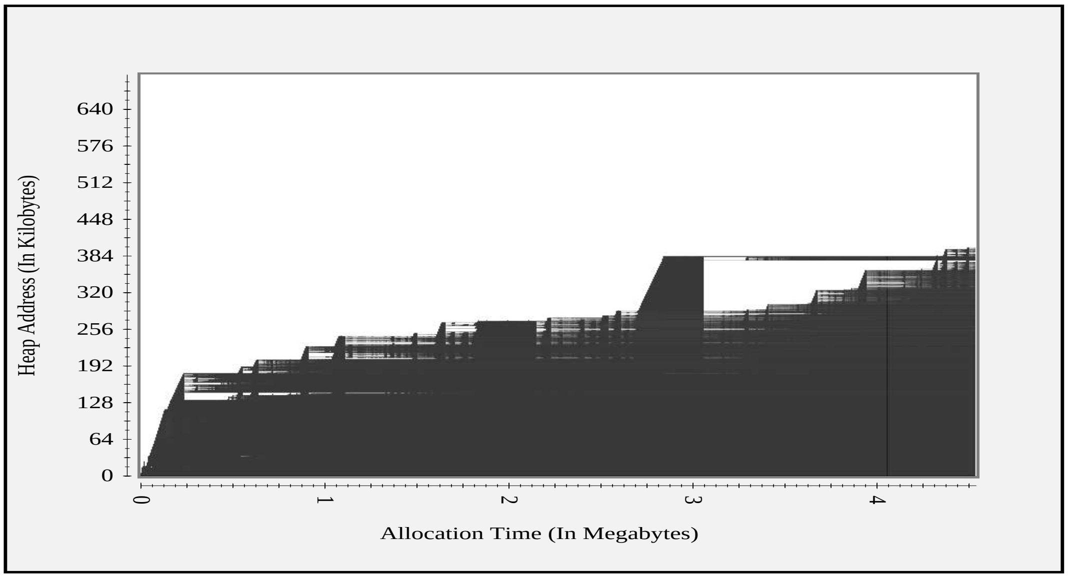

Figure C.56: Fragmentation plot for P2C using Lea's 2.6.1 allocator (accounting for all overheads)

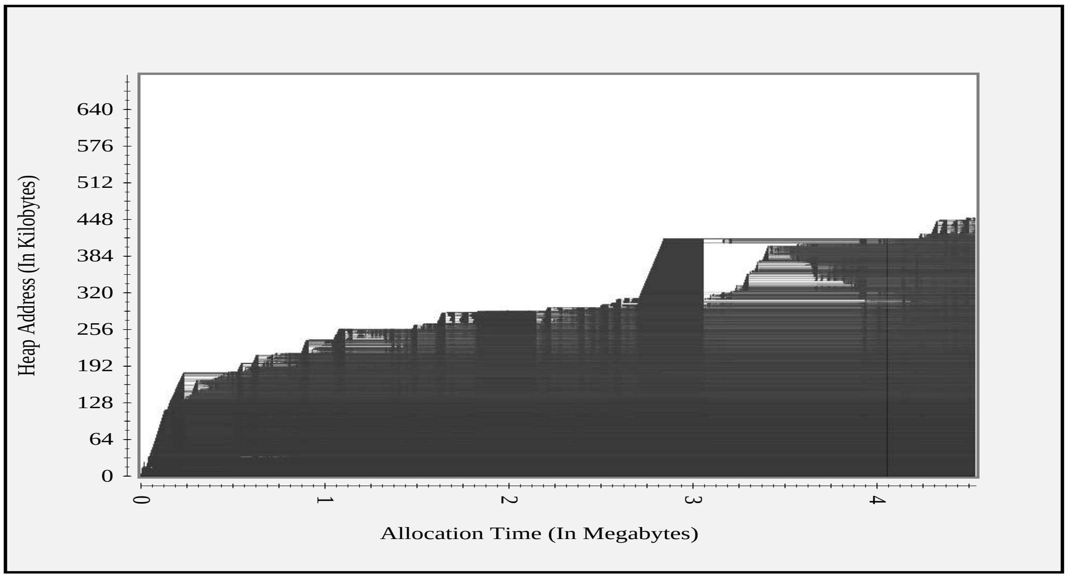

Figure C.57: Fragmentation plot for P2C using the next-fit LIFO no-footer allocator (accounting for all overheads)

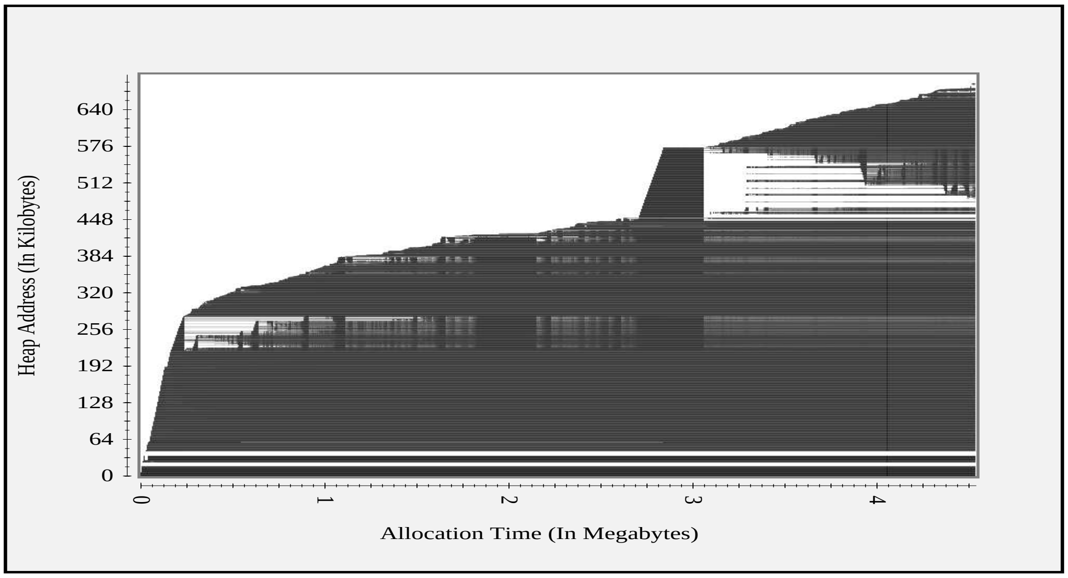

Figure C.58: Fragmentation plot for P2C using the simple segregated storage 2^N allocator (accounting for all overheads)

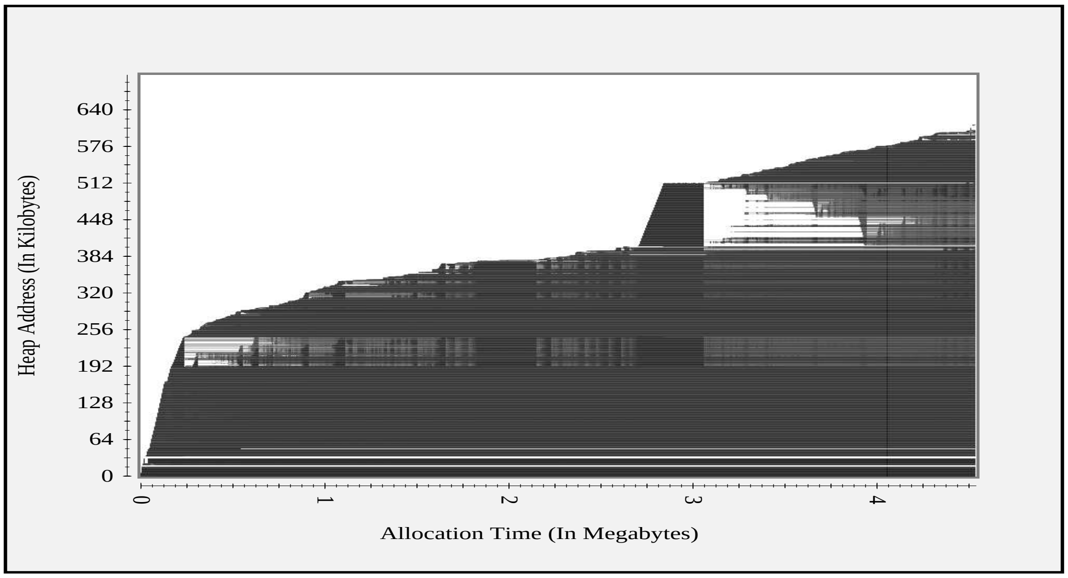

Figure C.59: Fragmentation plot for P2C using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

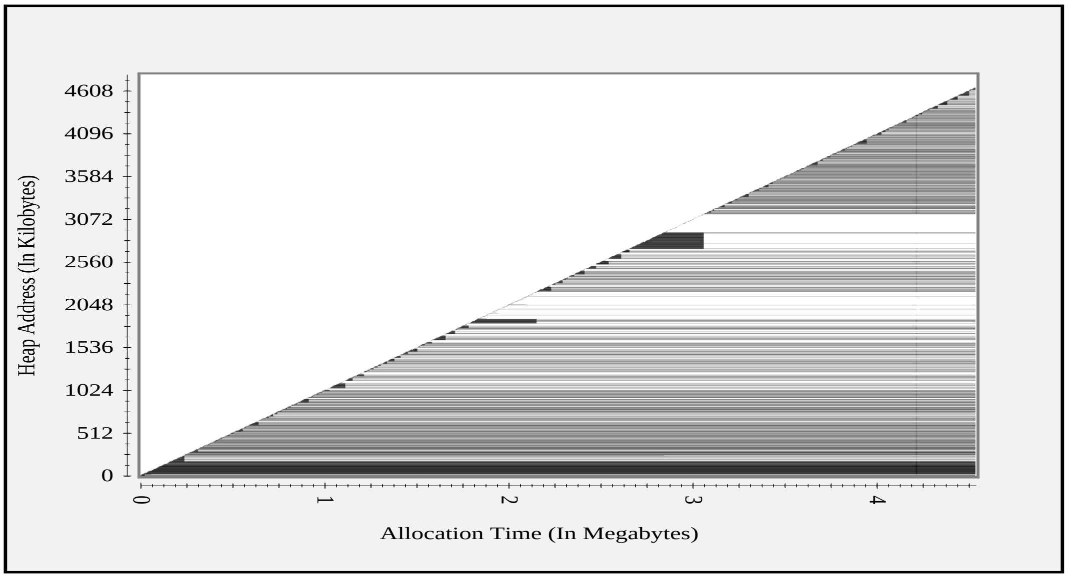

Figure C.60: Fragmentation plot for P2C using the linear allocator

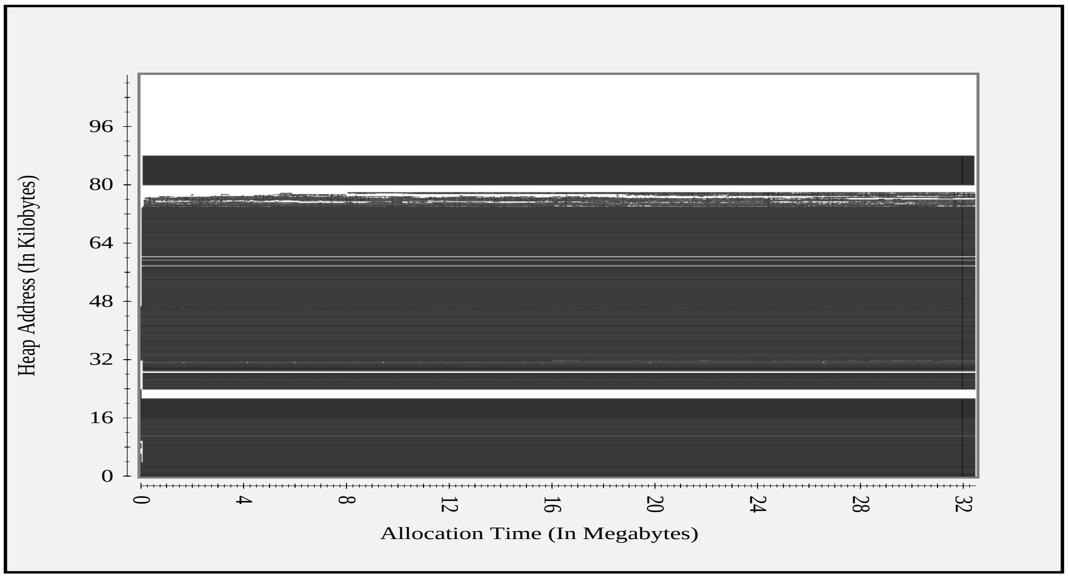

Figure C.61: Fragmentation plot for Perl using the binary-buddy allocator (accounting for all overheads)

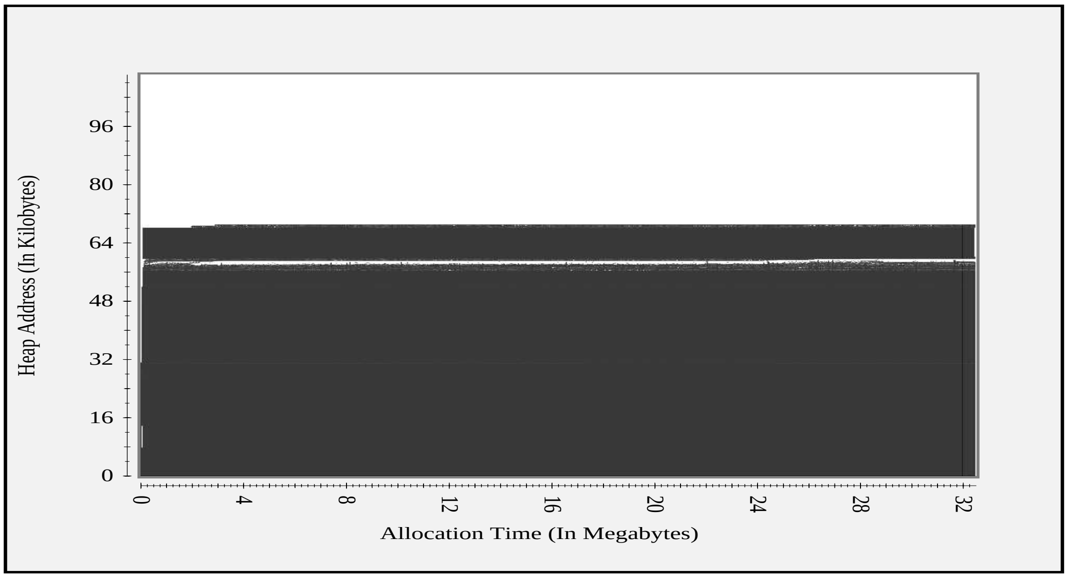

Figure C.62: Fragmentation plot for Perl using the best-fit LIFO no-footer allocator (accounting for all overheads)

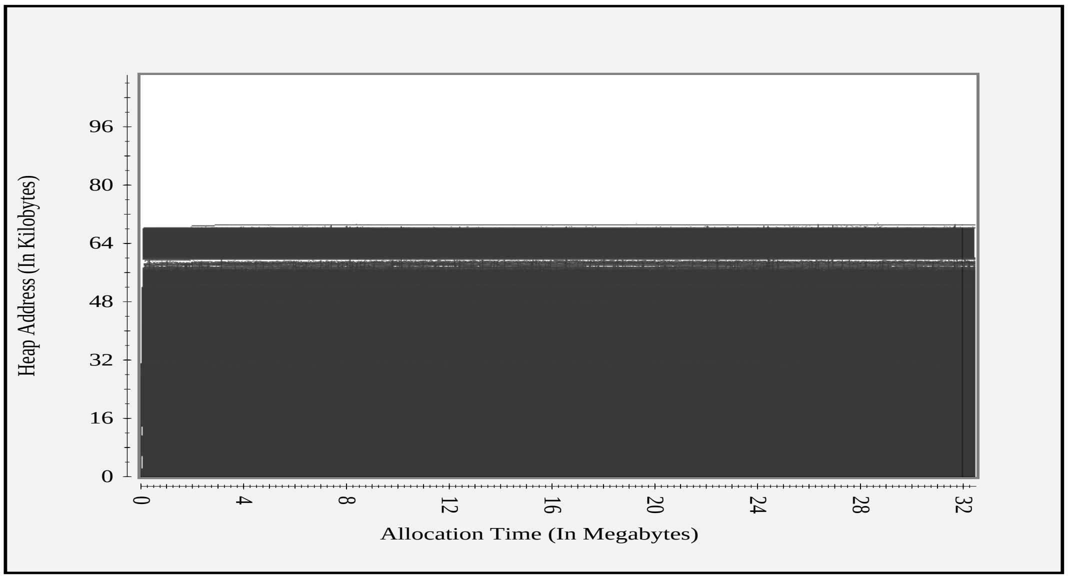

Figure C.63: Fragmentation plot for Perl using the first-fit address ordered no-footer allocator (accounting for all overheads)

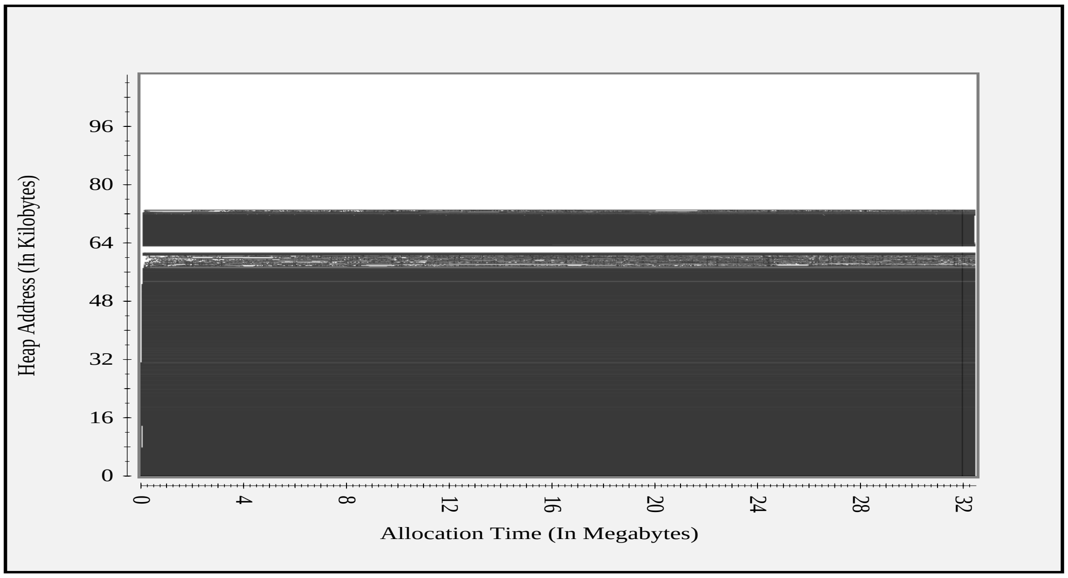

Figure C.64: Fragmentation plot for Perl using the first-fit LIFO no-footer allocator (accounting for all overheads)

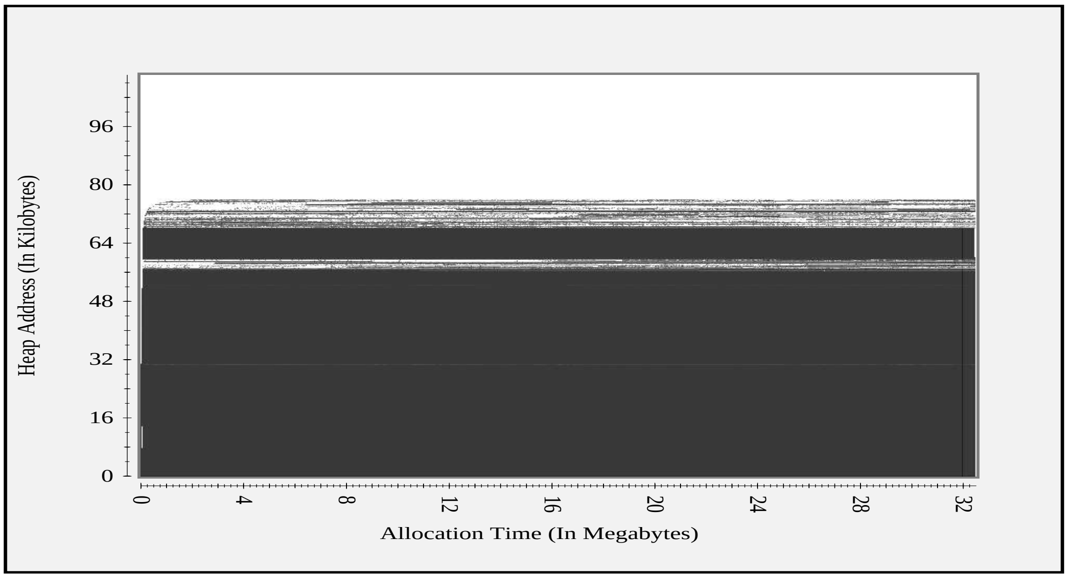

Figure C.65: Fragmentation plot for Perl using the half-fit allocator (accounting for all overheads)

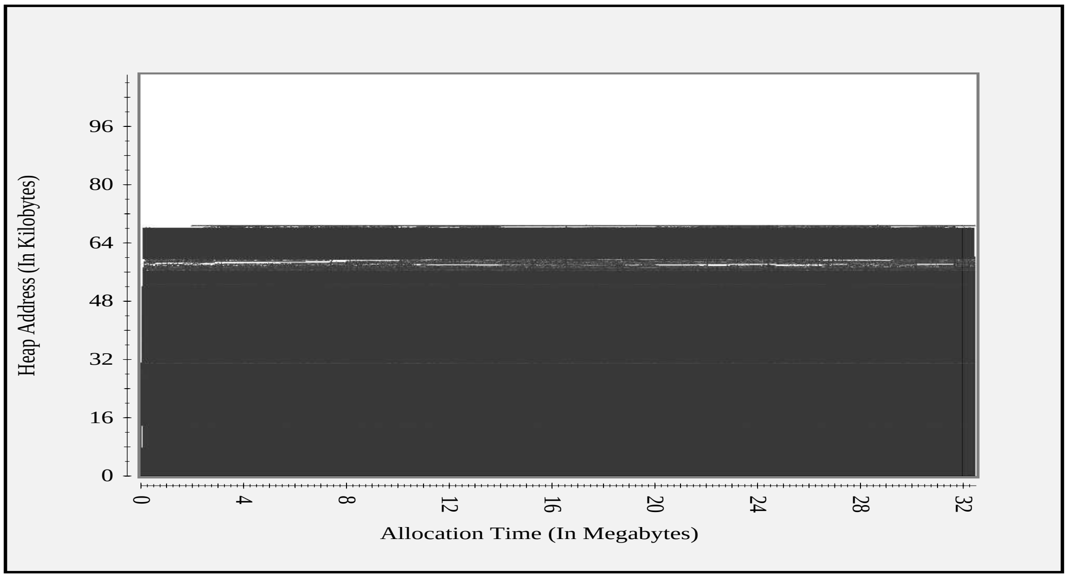

Figure C.66: Fragmentation plot for Perl using Lea's 2.6.1 allocator (accounting for all overheads)

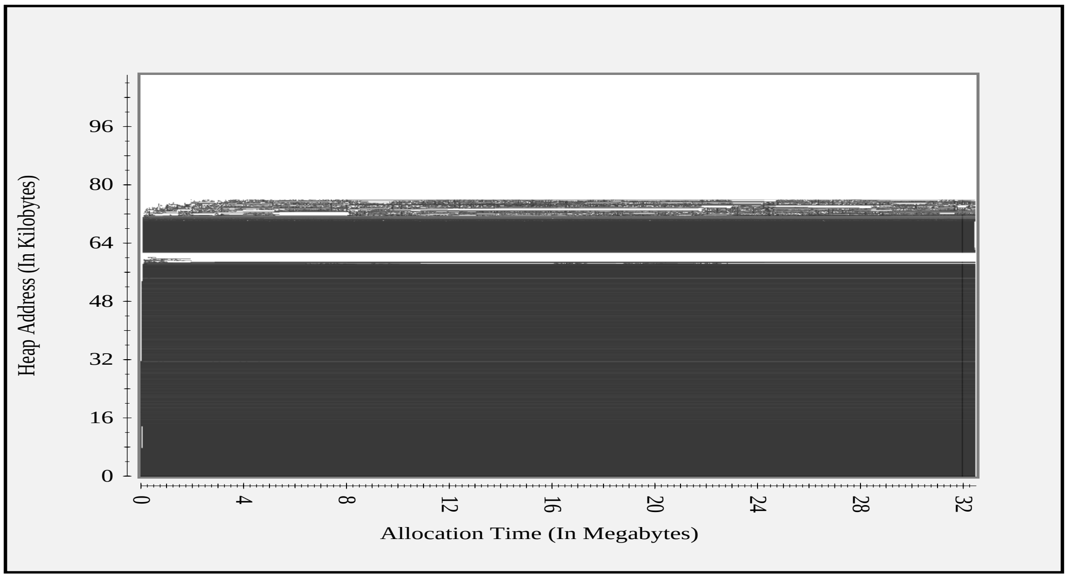

Figure C.67: Fragmentation plot for Perl using the next-fit LIFO no-footer allocator (accounting for all overheads)

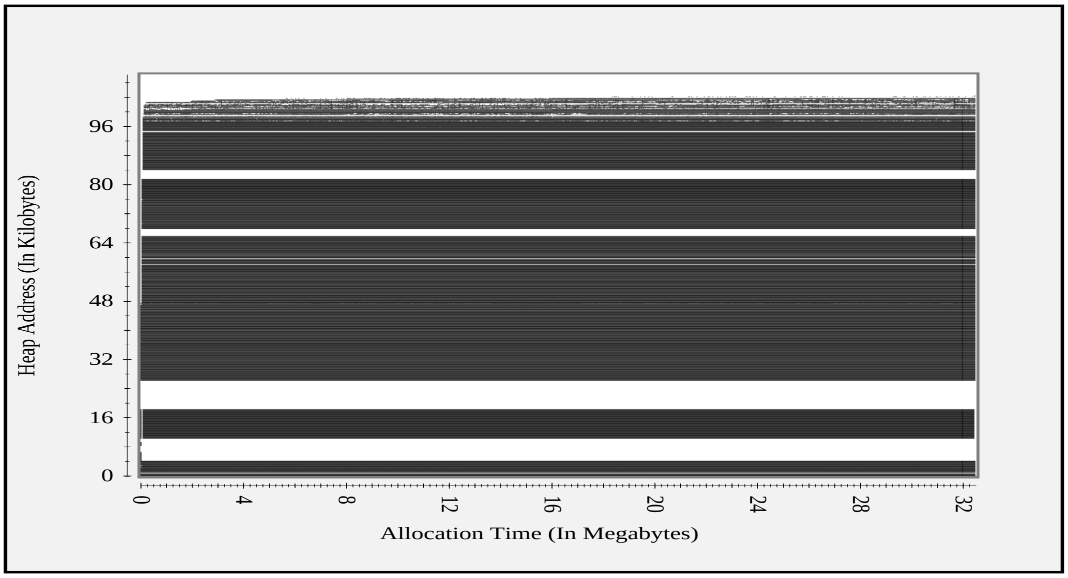

Figure C.68: Fragmentation plot for Perl using the simple segregated storage 2^N allocator (accounting for all overheads)

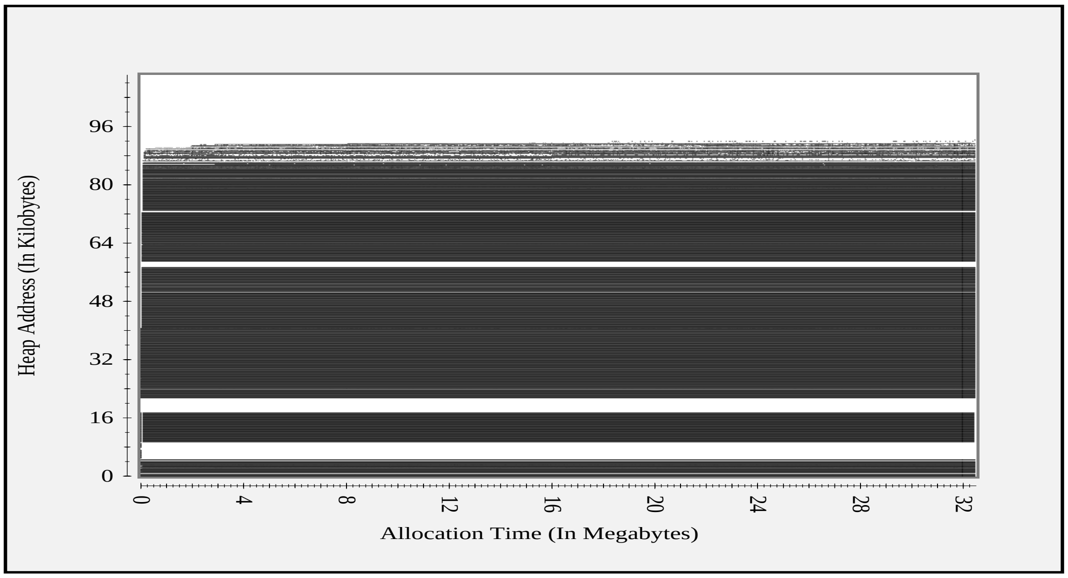

Figure C.69: Fragmentation plot for Perl using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

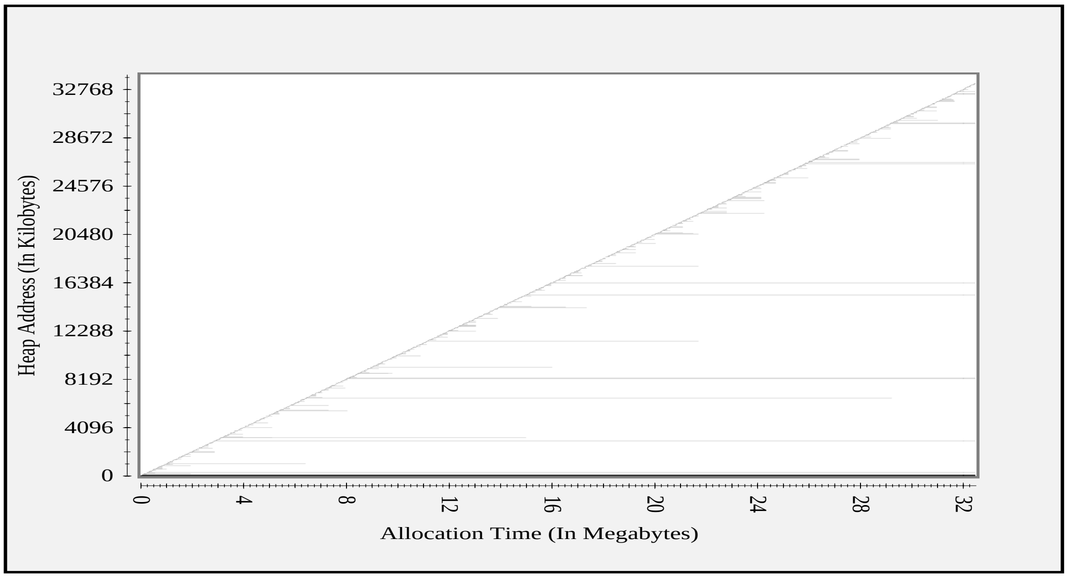

Figure C.70: Fragmentation plot for Perl using the linear allocator

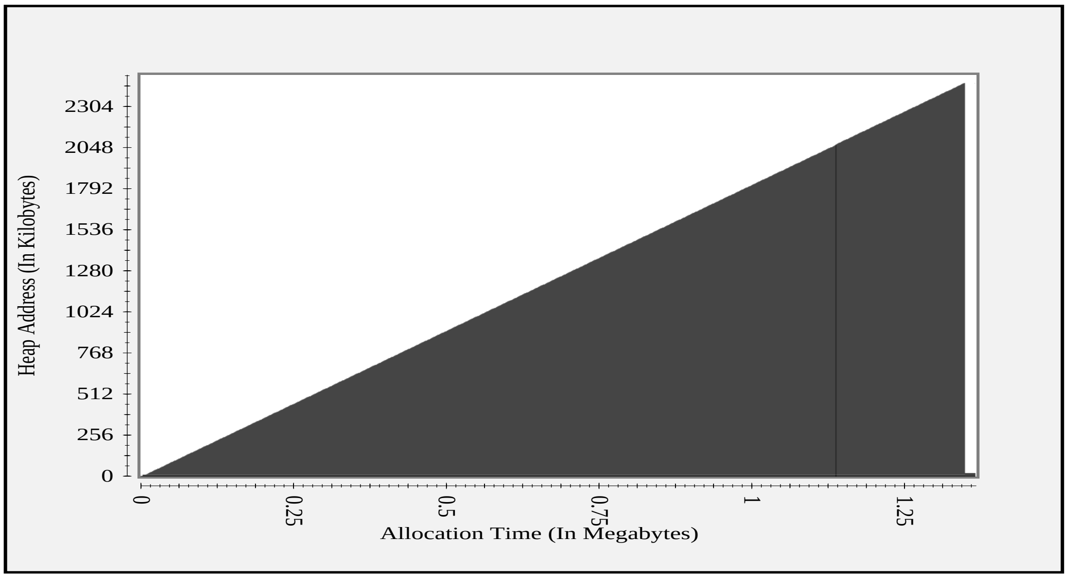

Figure C.71: Fragmentation plot for LRUsim using the binary-buddy allocator (accounting for all overheads)

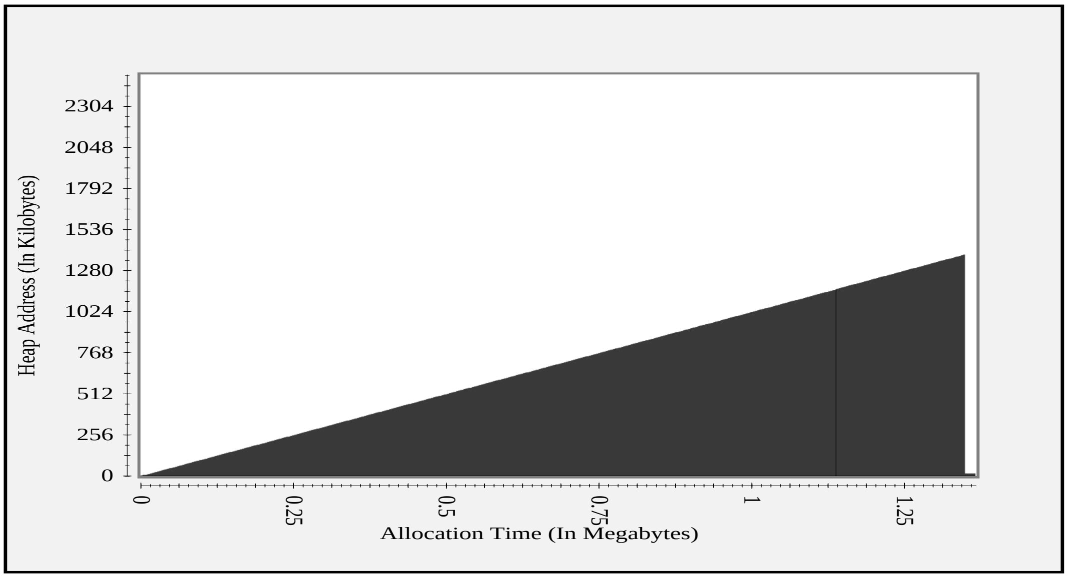

Figure C.72: Fragmentation plot for LRUsim using the best-fit LIFO no-footer allocator (accounting for all overheads)

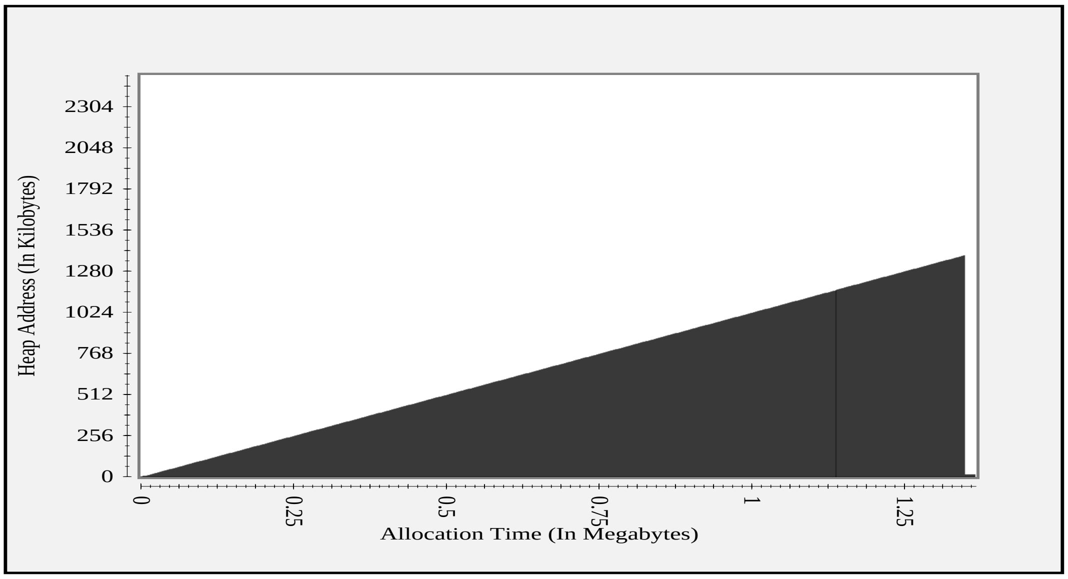

Figure C.73: Fragmentation plot for LRUsim using the first-fit address-ordered no-footer allocator (accounting for all overheads)

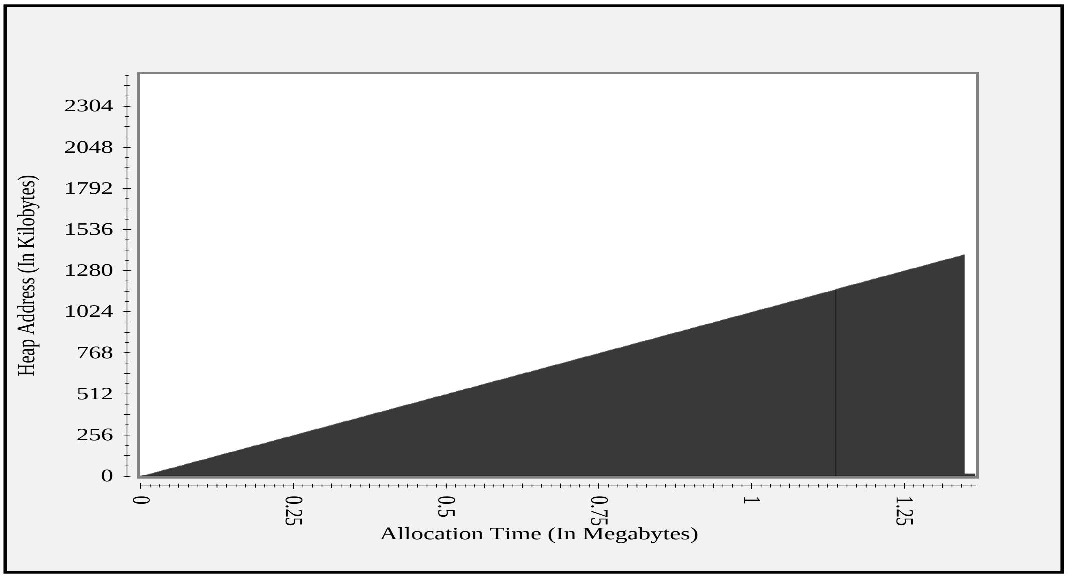

Figure C.74: Fragmentation plot for LRUsim using the first-fit LIFO no-footer allocator (accounting for all overheads)

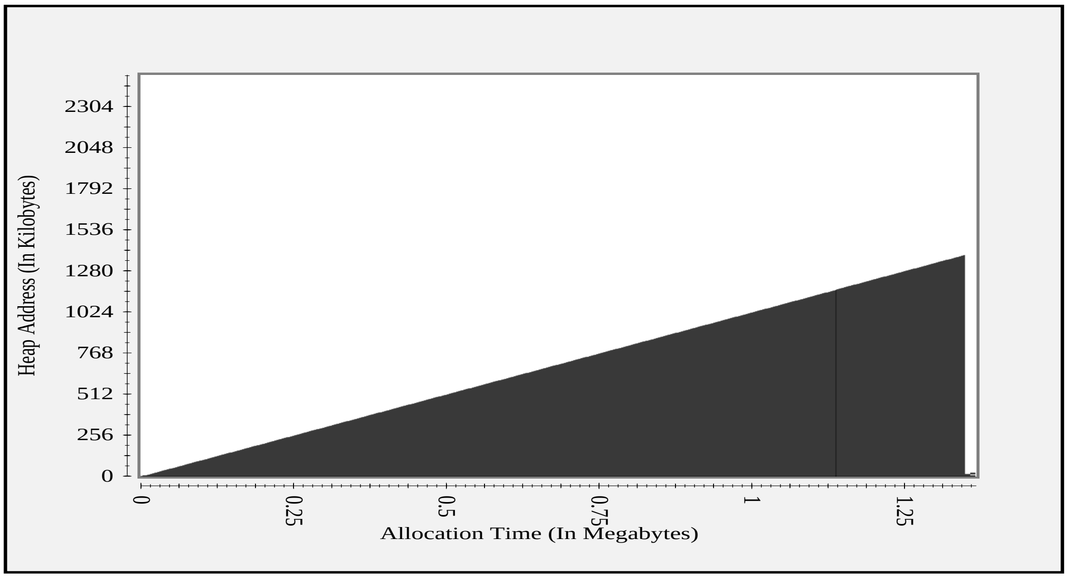

Figure C.75: Fragmentation plot for LRUsim using the half-fit allocator (accounting for all overheads)

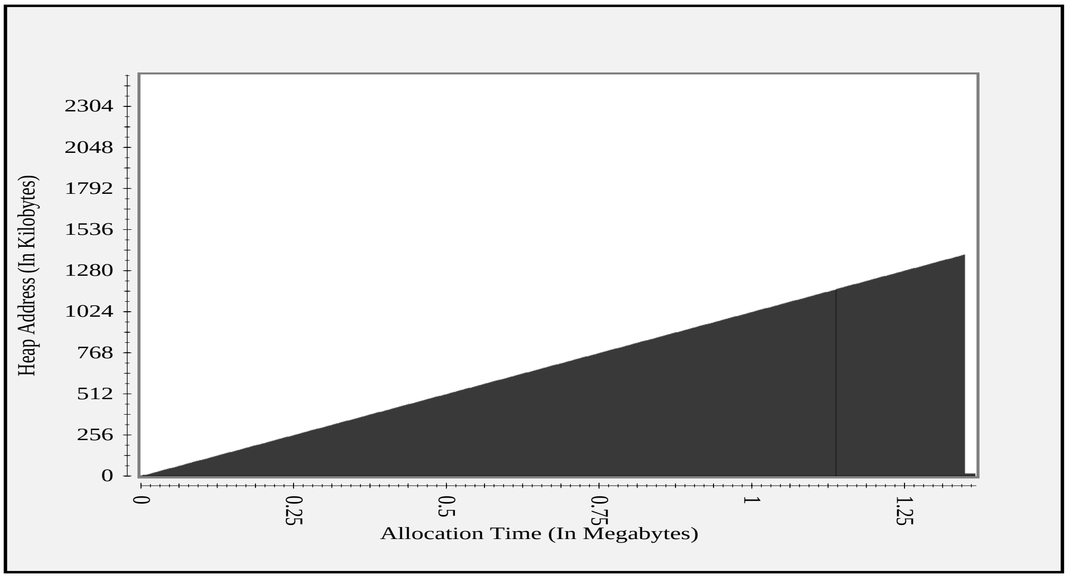

Figure C.76: Fragmentation plot for LRUsim using Lea's 2.6.1 allocator (accounting for all overheads)

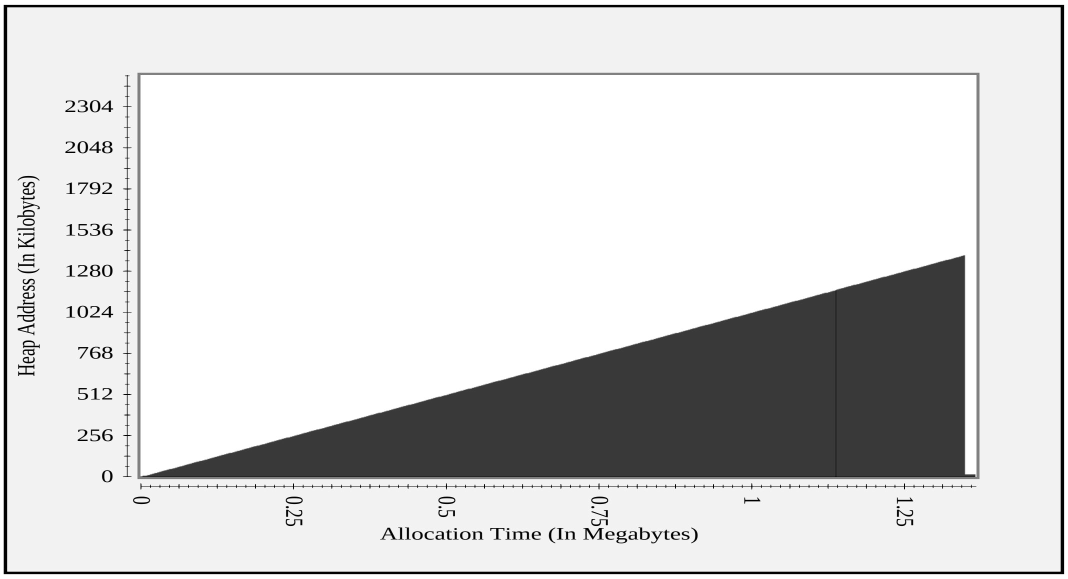

Figure C.77: Fragmentation plot for LRUsim using the next-fit LIFO no-footer allocator (accounting for all overheads)

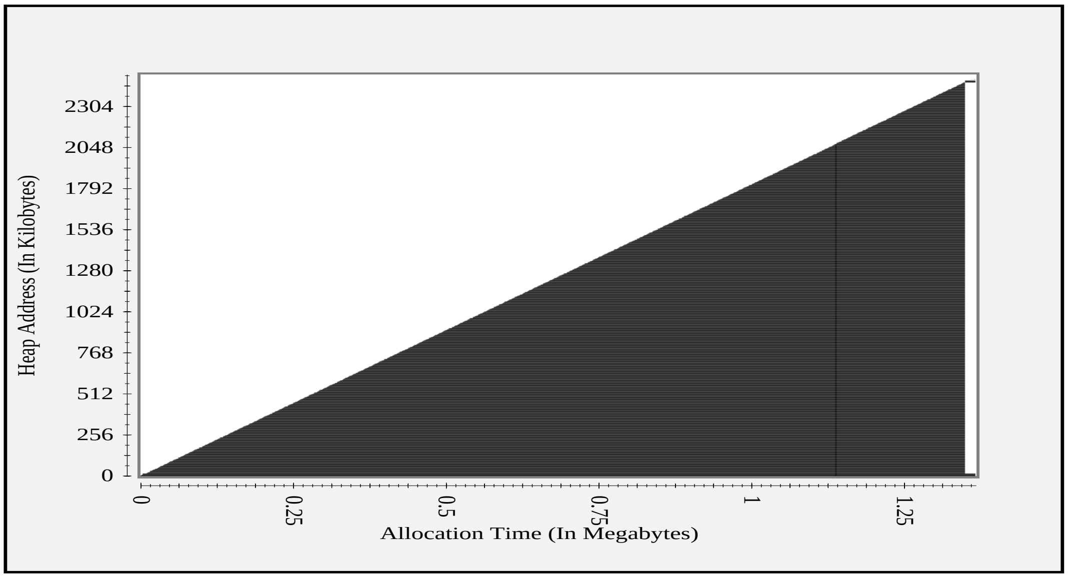

Figure C.78: Fragmentation plot for LRUsim using the simple segregated storage 2^N allocator (accounting for all overheads)

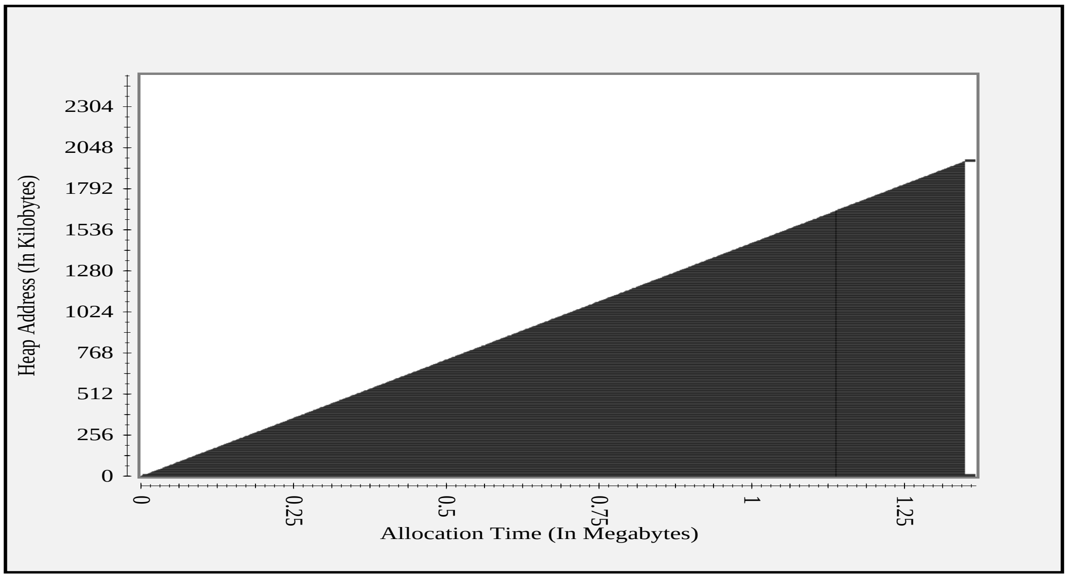

Figure C.79: Fragmentation plot for LRUsim using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

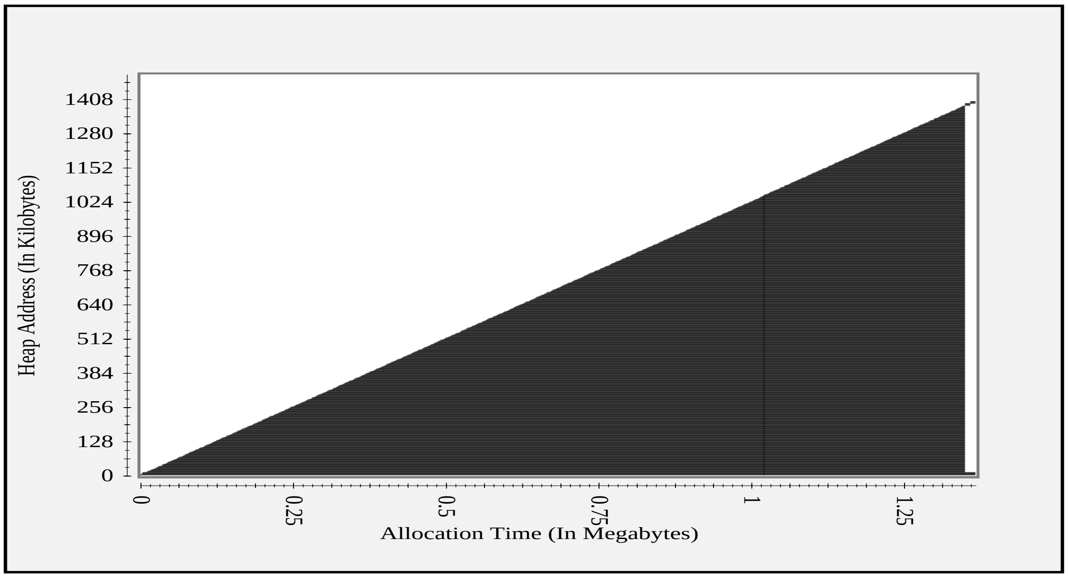

Figure C.80: Fragmentation plot for LRUsim using the linear allocator

Appendix D

Shuffled Heap Plots

The pictures that follow are fragmentation plots of the form first shown in Section 2.15.1. These pictures show the heap use for actual implementation of various selected allocators across time compared to the heap usage of the same allocator with a shuffled version of that trace. Note how strikingly different many of the pictures look. As was described in Section 2.14, darker pixels indicate a larger percentage of memory in use.

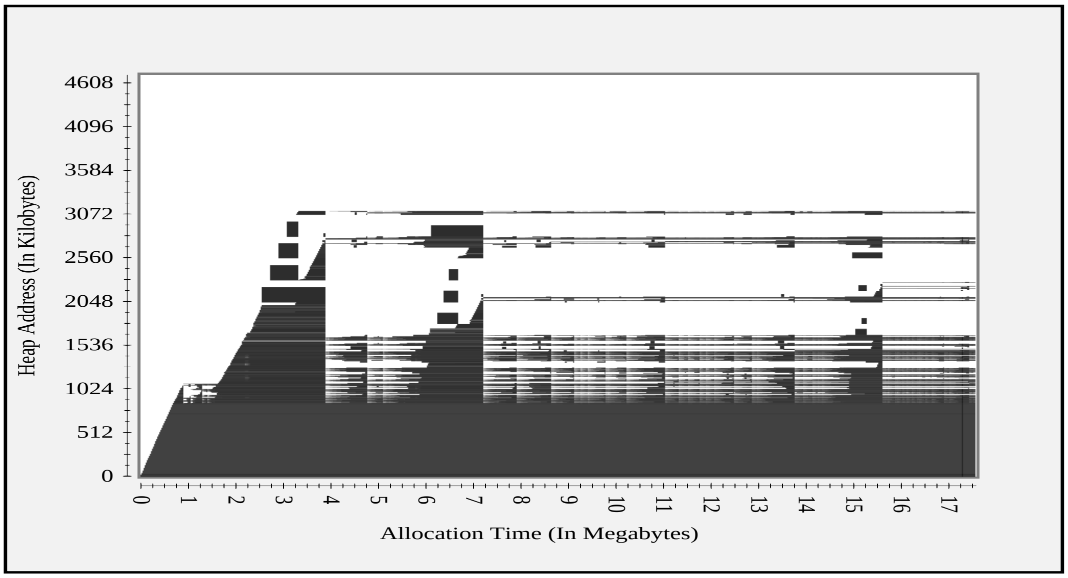

Figure D.1: Fragmentation plot for GCC using the binary-buddy allocator (accounting for all overheads)

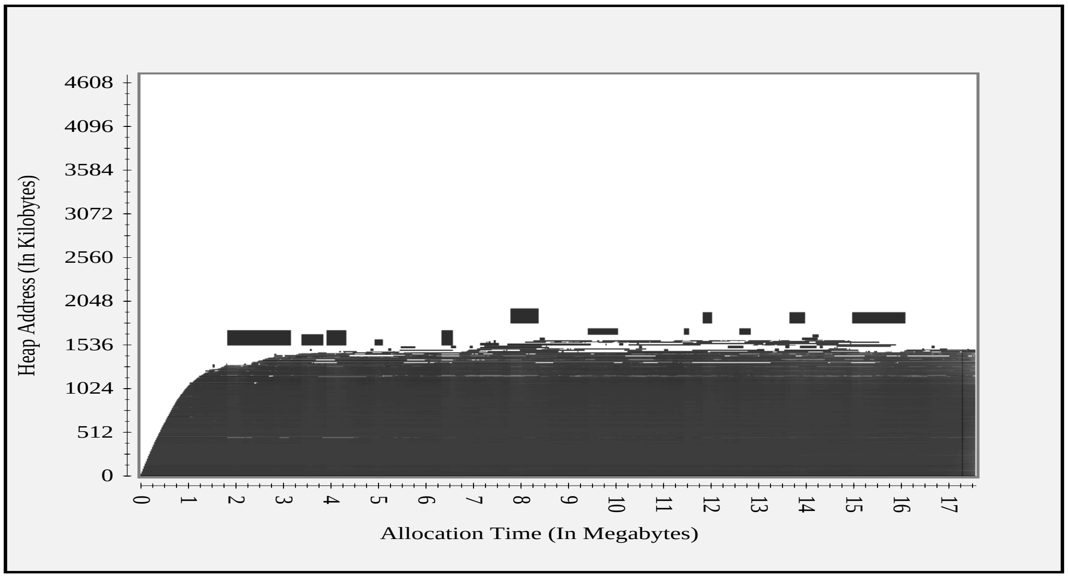

Figure D.2: Fragmentation plot for GCC *shuffled* using the binary-buddy allocator (accounting for all overheads)

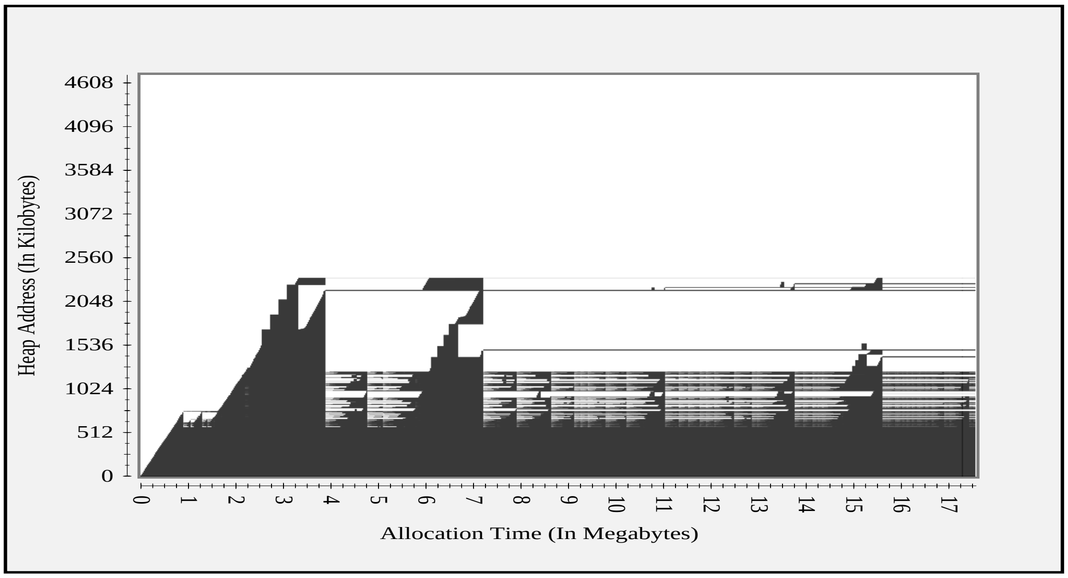

Figure D.3: Fragmentation plot for GCC using the best-fit LIFO no-footer allocator (accounting for all overheads)

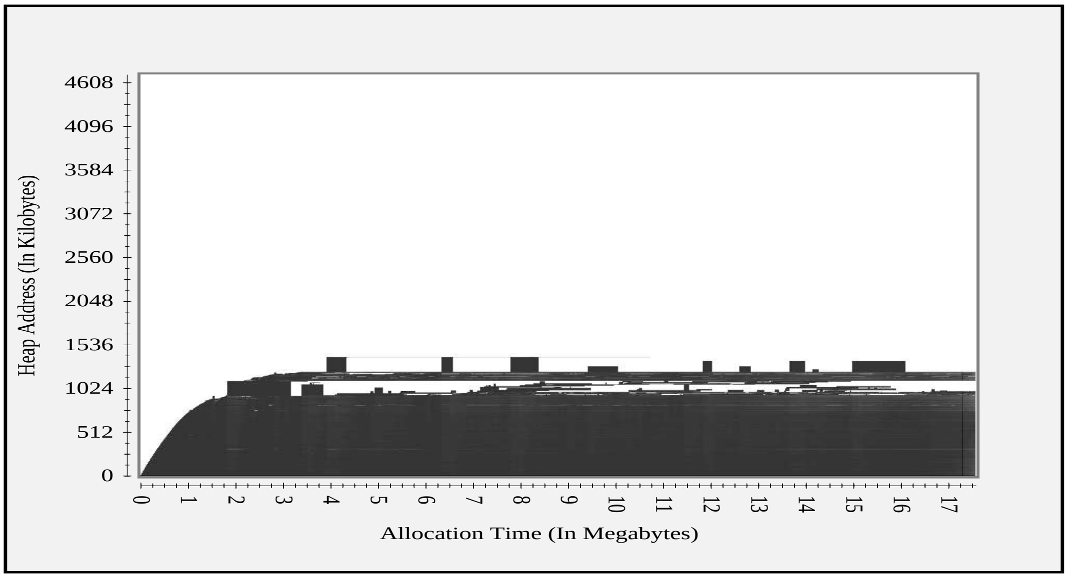

Figure D.4: Fragmentation plot for GCC *shuffled* using the best-fit LIFO no-footer allocator (accounting for all overheads)

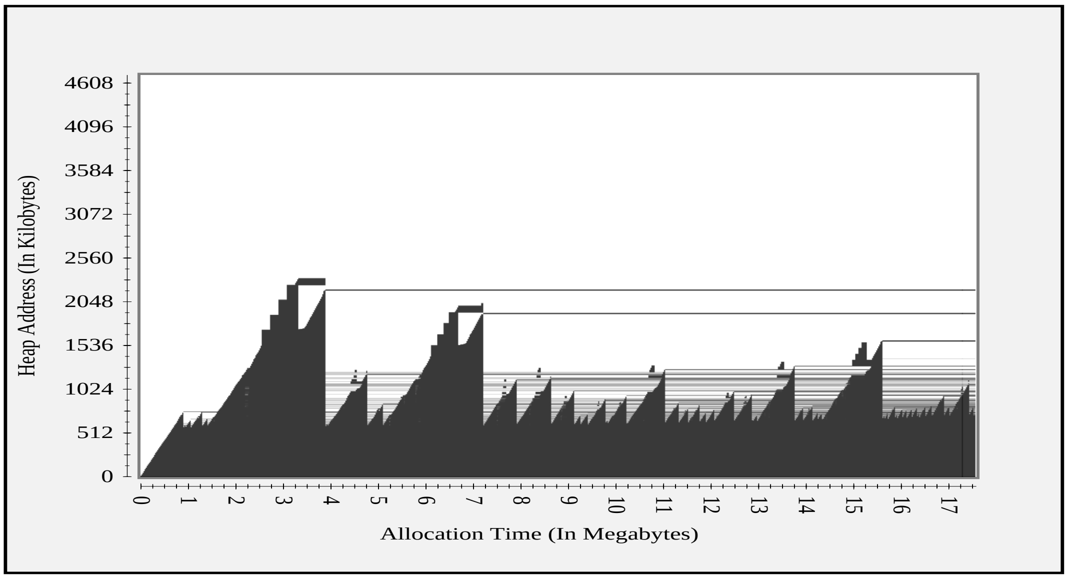

Figure D.5: Fragmentation plot for GCC using the first-fit address-ordered no-footer allocator (accounting for all overheads)

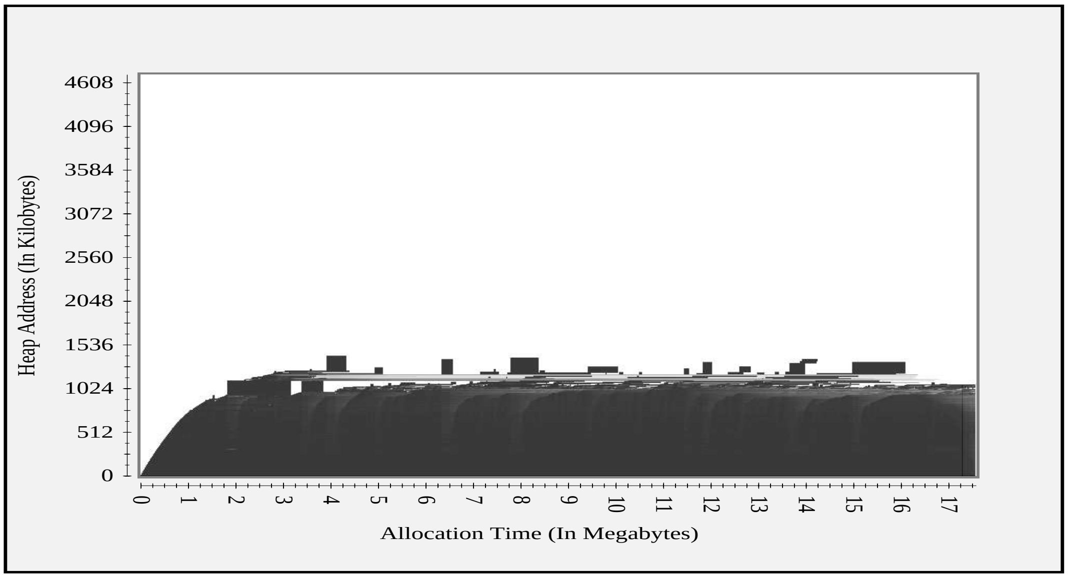

Figure D.6: Fragmentation plot for GCC *shuffled* using the first-fit address-ordered no-footer allocator (accounting for all overheads)

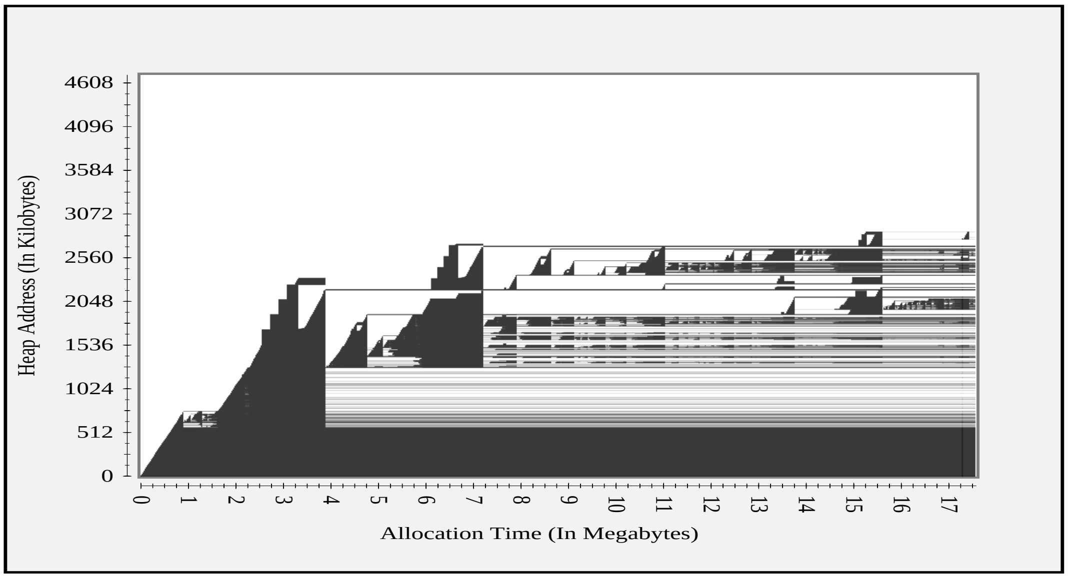

Figure D.7: Fragmentation plot for GCC using the first-fit LIFO no-footer allocator (accounting for all overheads)

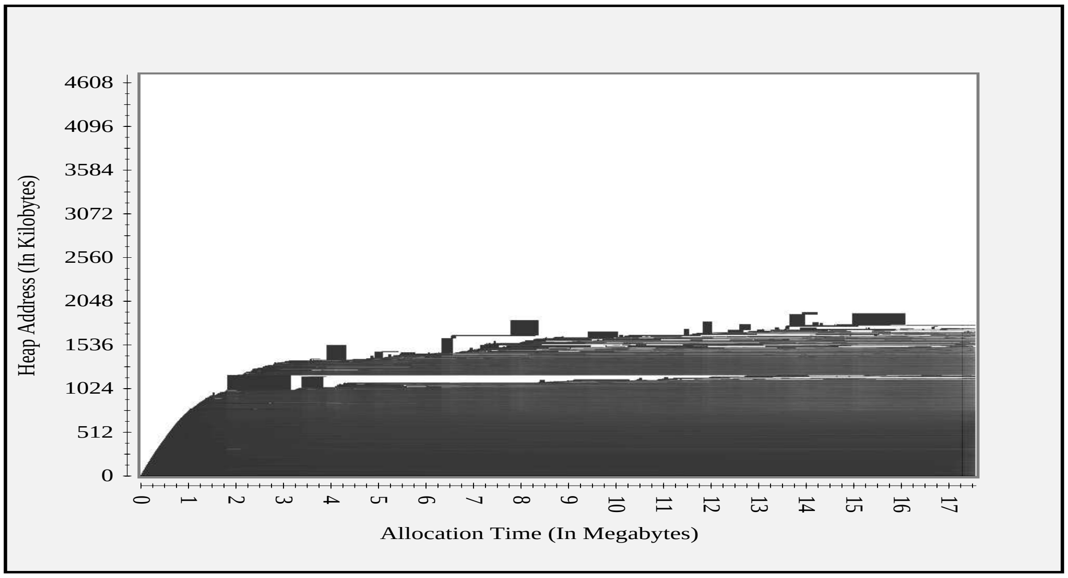

Figure D.8: Fragmentation plot for GCC *shuffled* using the first-fit LIFO no-footer allocator (accounting for all overheads)

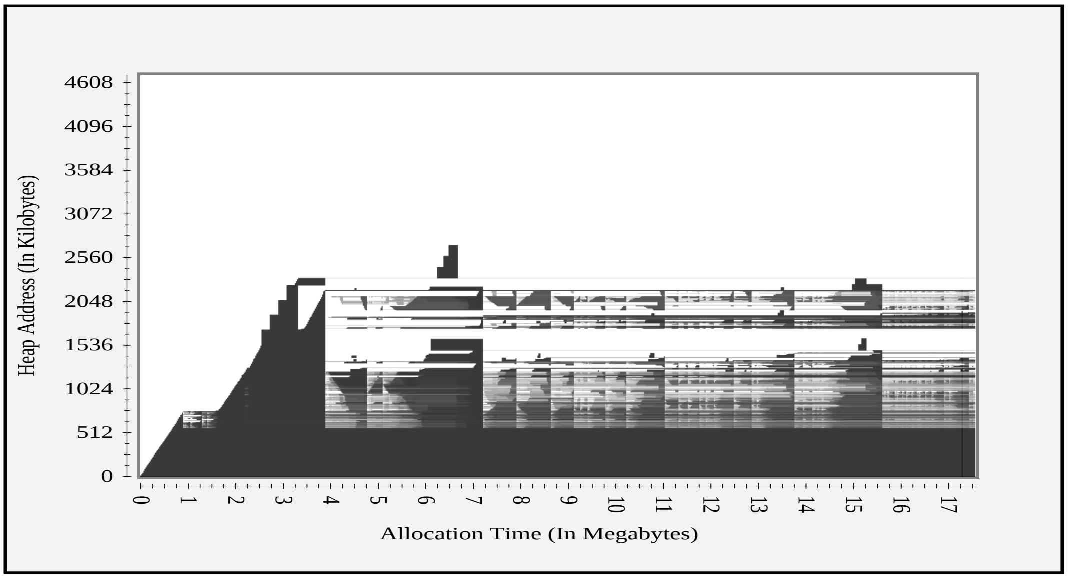

Figure D.9: Fragmentation plot for GCC using the half-fit allocator (accounting for all overheads)

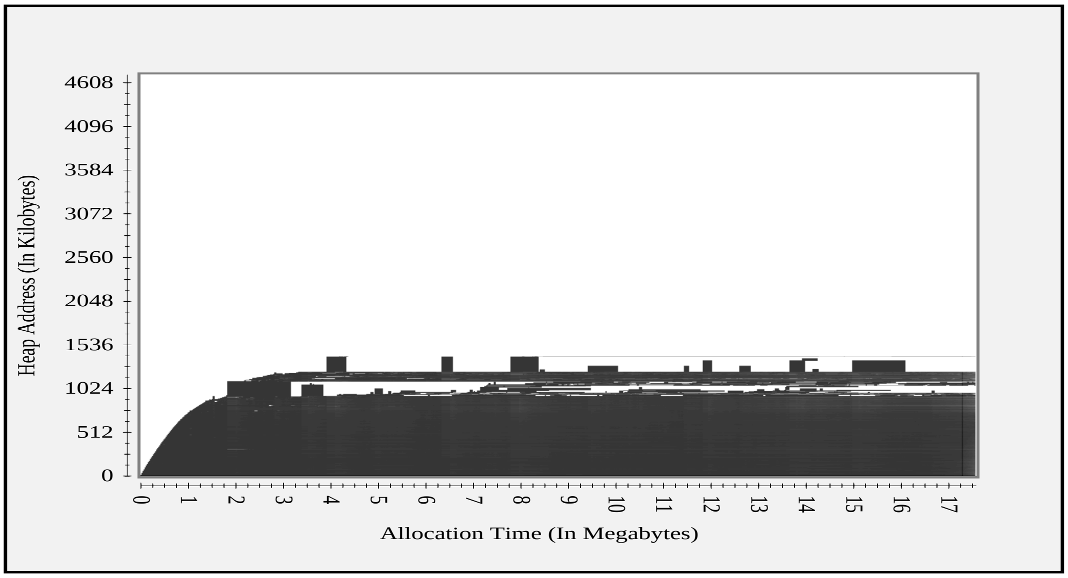

Figure D.10: Fragmentation plot for GCC *shuffled* using the half-fit allocator (accounting for all overheads)

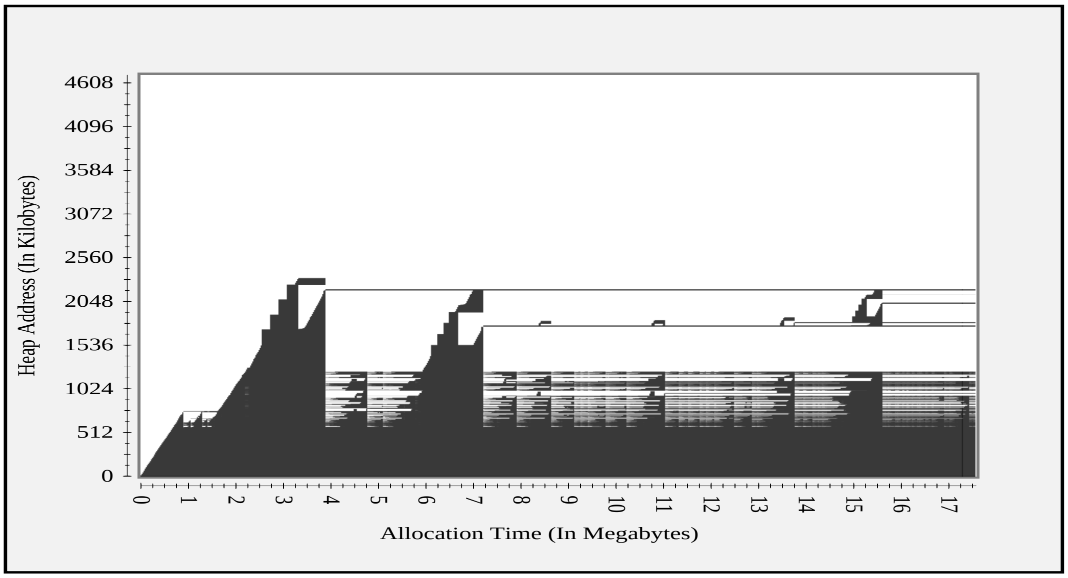

Figure D.11: Fragmentation plot for GCC using Lea's 2.6.1 allocator (accounting for all overheads)

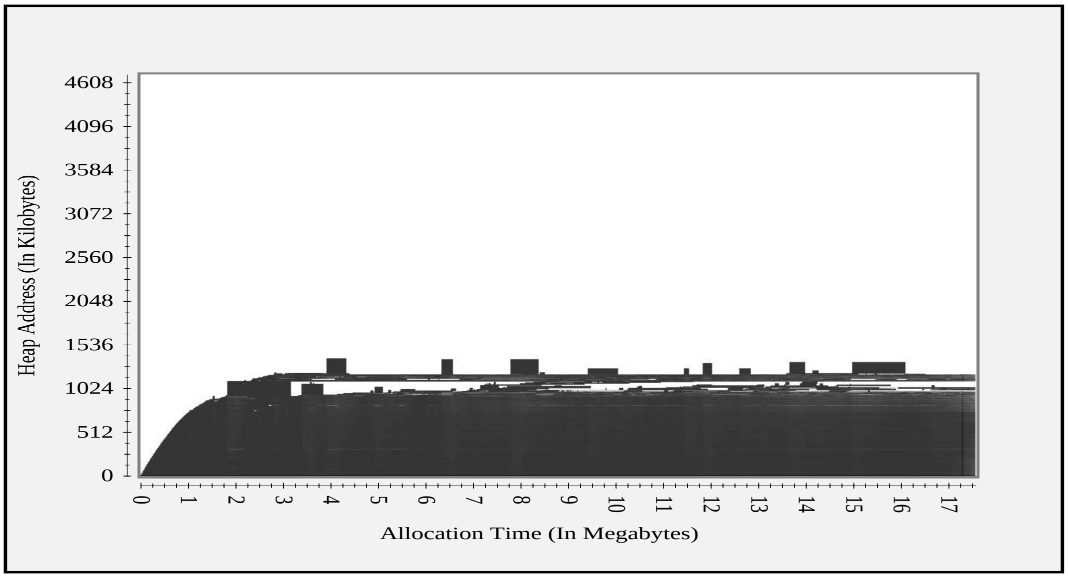

Figure D.12: Fragmentation plot for GCC *shuffled* using Lea's allocator (accounting for all overheads)

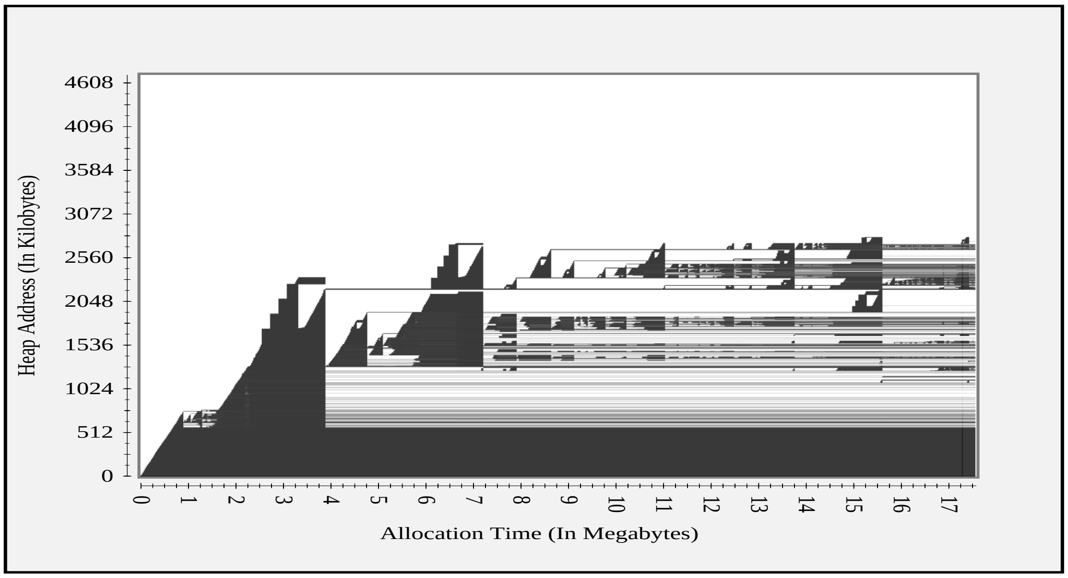

Figure D.13: Fragmentation plot for GCC using the next-fit LIFO no-footer allocator (accounting for all overheads)

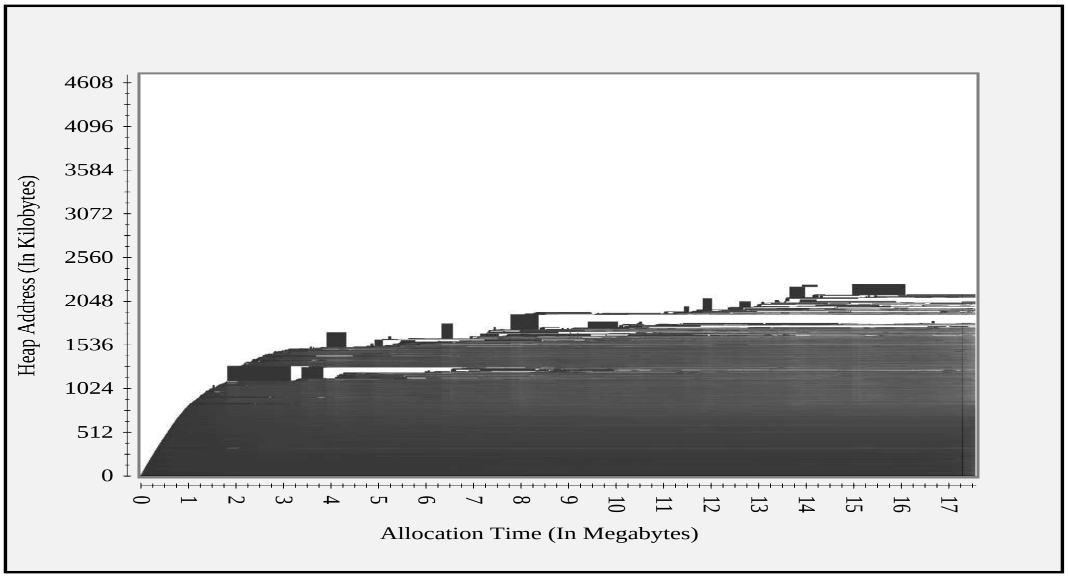

Figure D.14: Fragmentation plot for GCC *shuffled* using the next-fit LIFO no-footer allocator (accounting for all overheads)

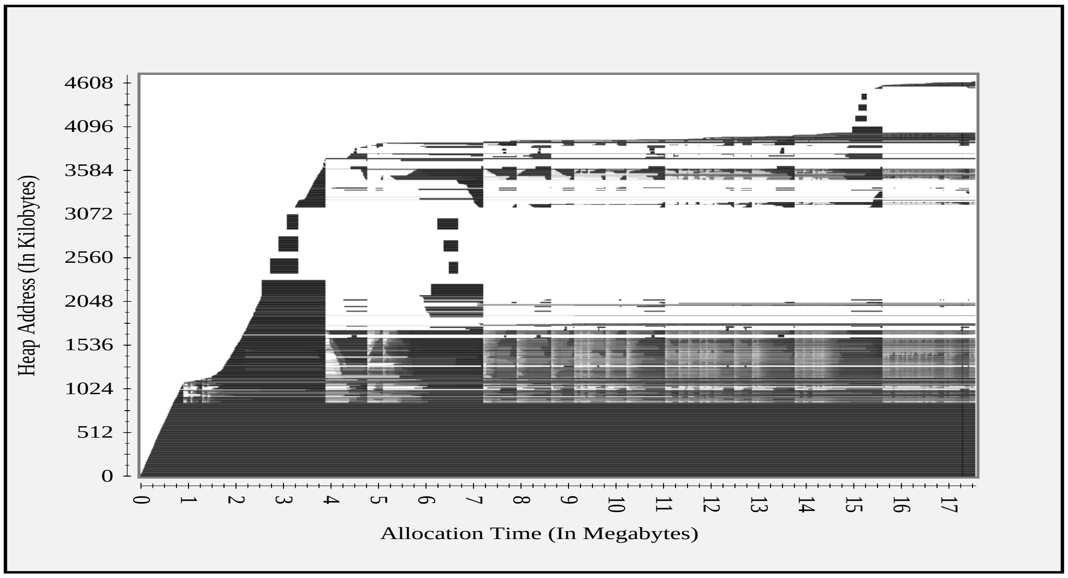

Figure D.15: Fragmentation plot for GCC using the simple segregated storage 2^N allocator (accounting for all overheads)

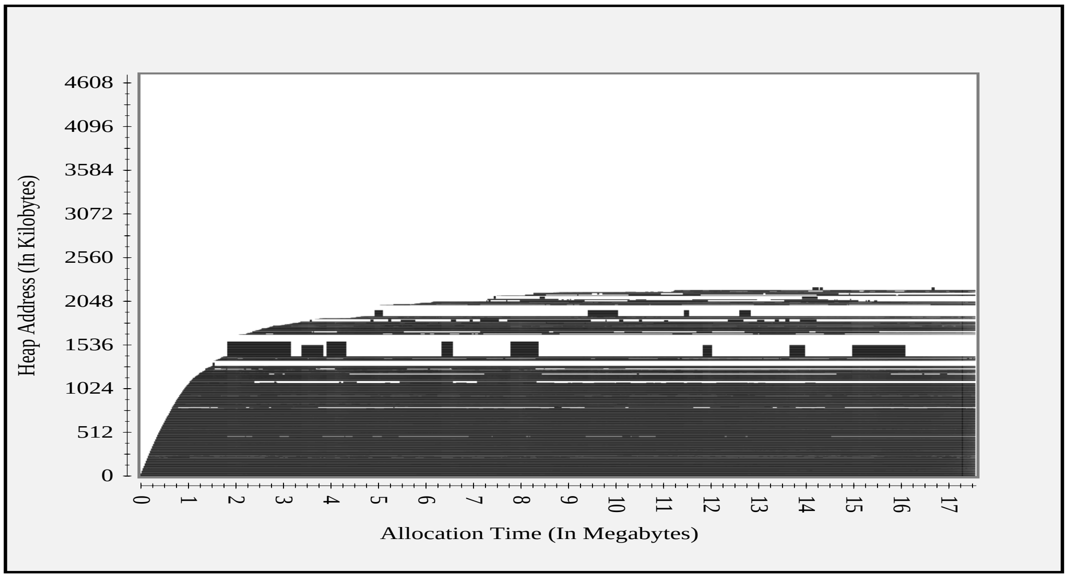

Figure D.16: Fragmentation plot for GCC *shuffled* using the simple segregated storage 2^N allocator (accounting for all overheads)

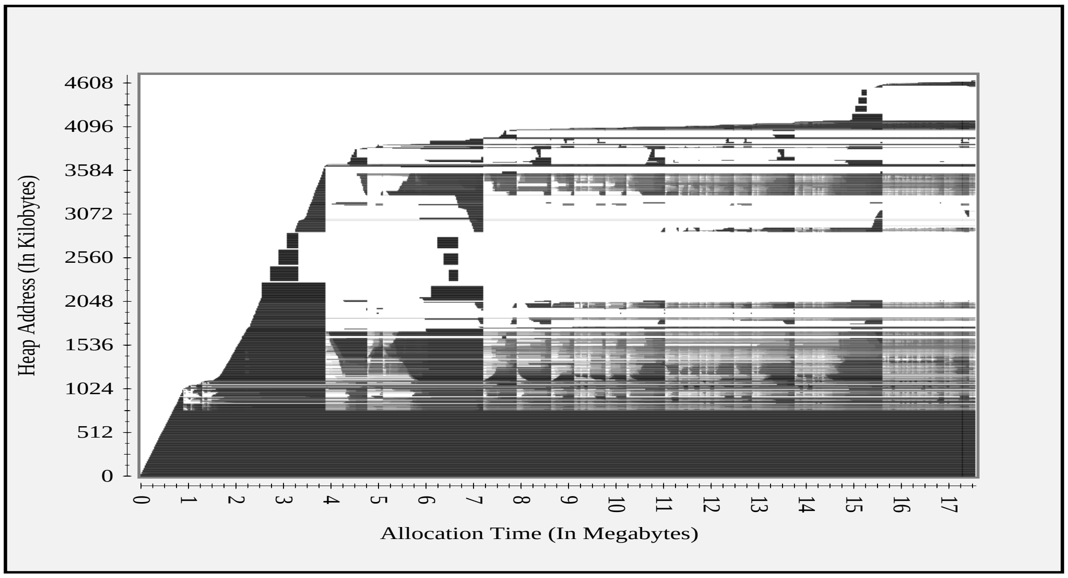

Figure D.17: Fragmentation plot for GCC using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

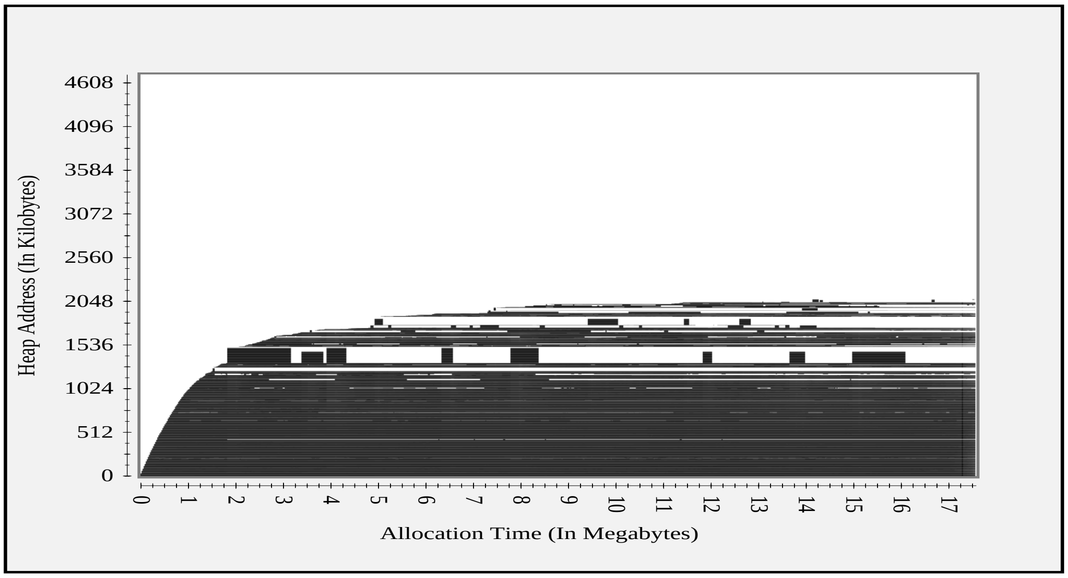

Figure D.18: Fragmentation plot for GCC *shuffled* using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

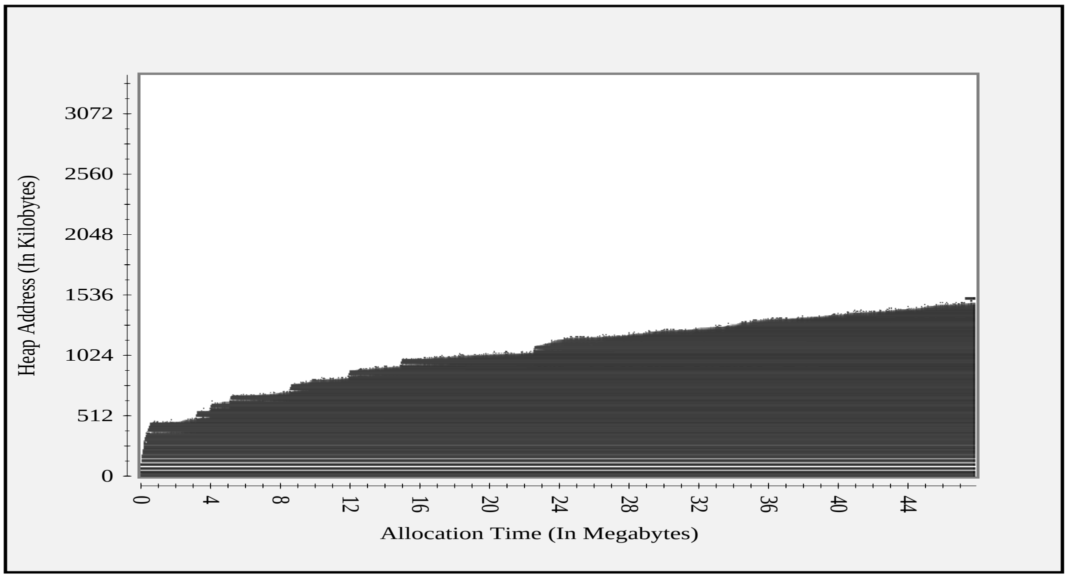

Figure D.19: Fragmentation plot for Ghostscript using the binary-buddy allocator (accounting for all overheads)

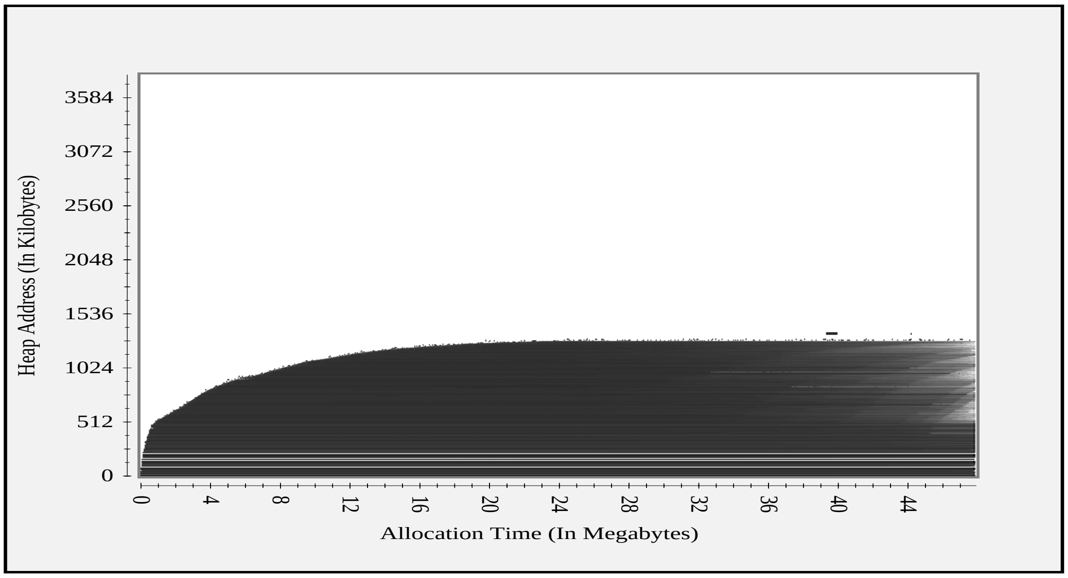

Figure D.20: Fragmentation plot for Ghostscript *shuffled* using the binary-buddy allocator (accounting for all overheads)

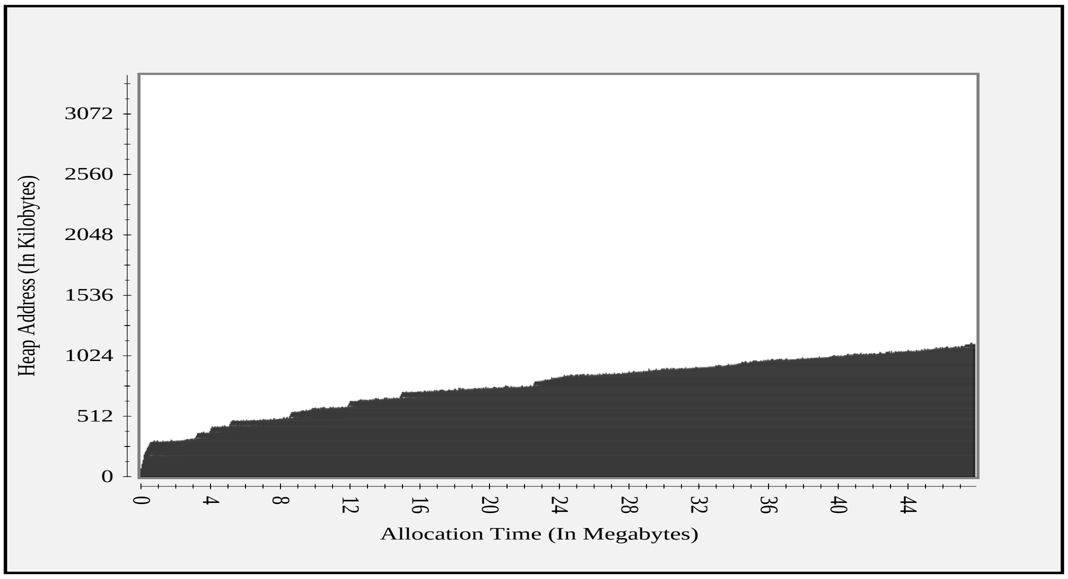

Figure D.21: Fragmentation plot for Ghostscript using the best-fit LIFO no-footer allocator (accounting for all overheads)

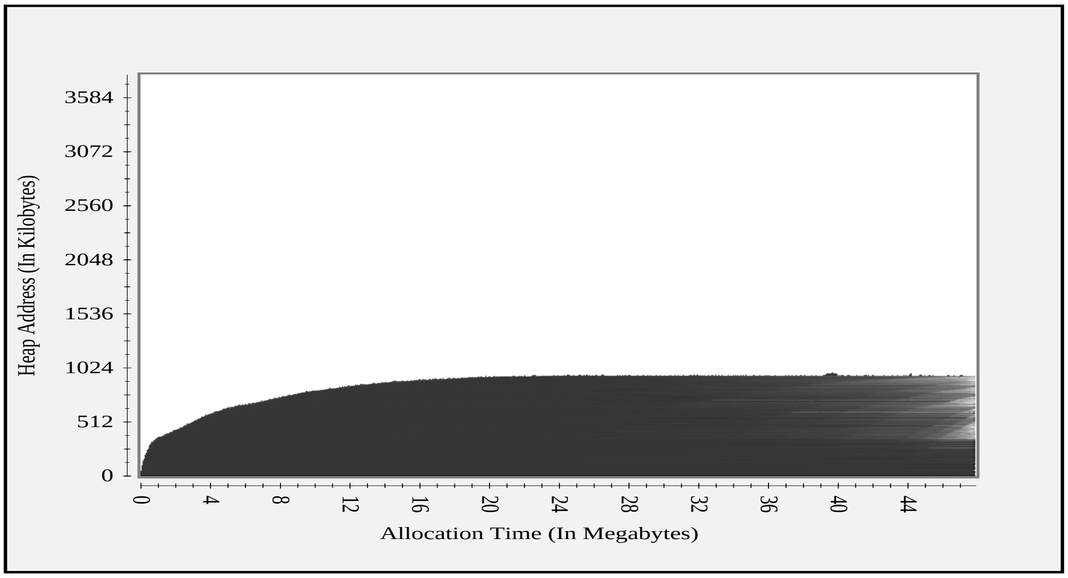

Figure D.22: Fragmentation plot for Ghostscript *shuffled* using the best-fit LIFO no-footer allocator (accounting for all overheads)

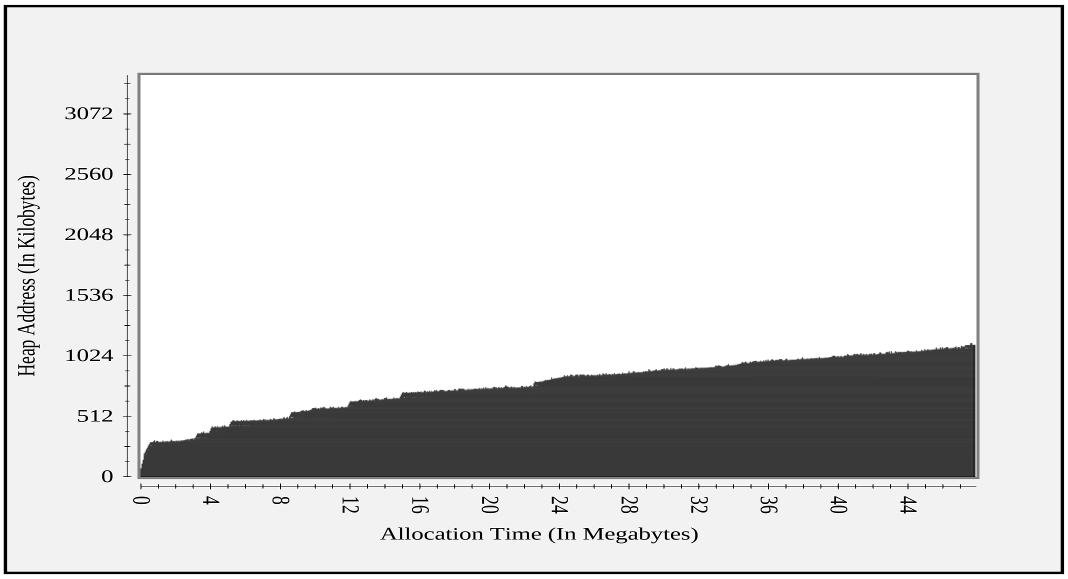

Figure D.23: Fragmentation plot for Ghostscript using the first-fit address-ordered no-footer allocator (accounting for all overheads)

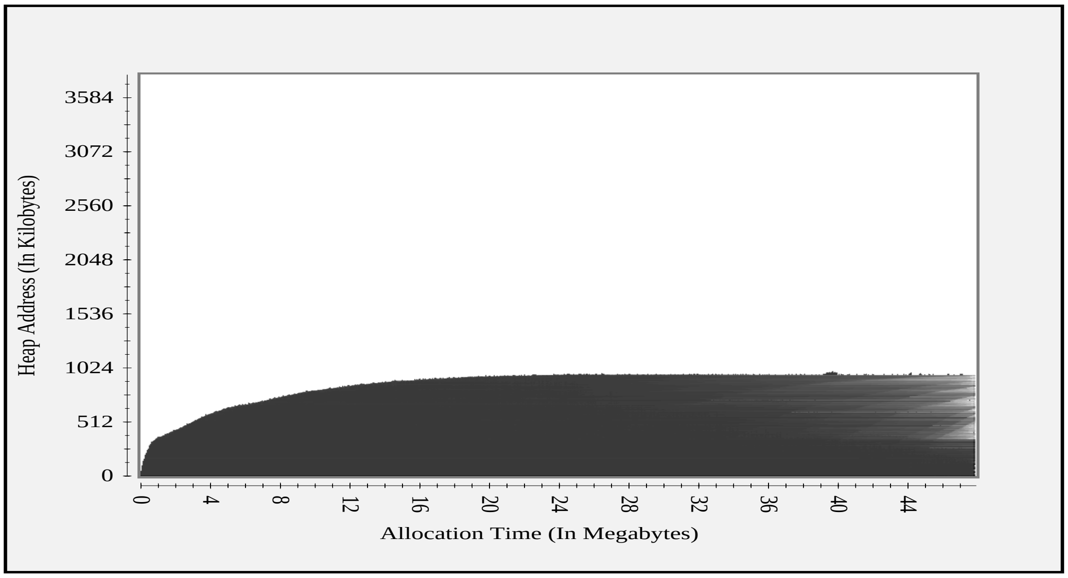

Figure D.24: Fragmentation plot for Ghostscript *shuffled* using the first-fit address-ordered no-footer allocator (accounting for all overheads)

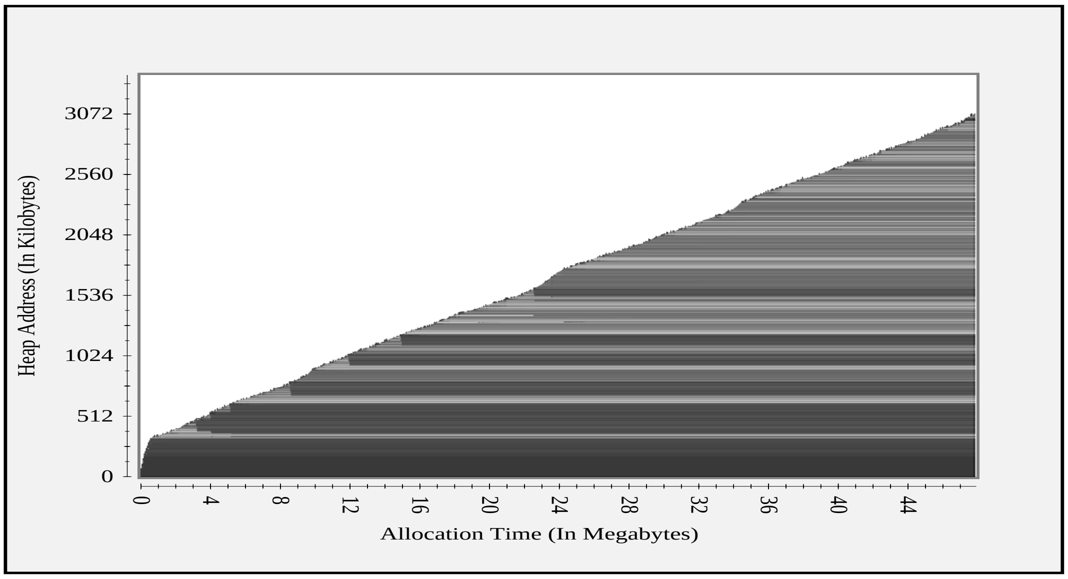

Figure D.25: Fragmentation plot for Ghostscript using the first-fit LIFO no-footer allocator (accounting for all overheads)

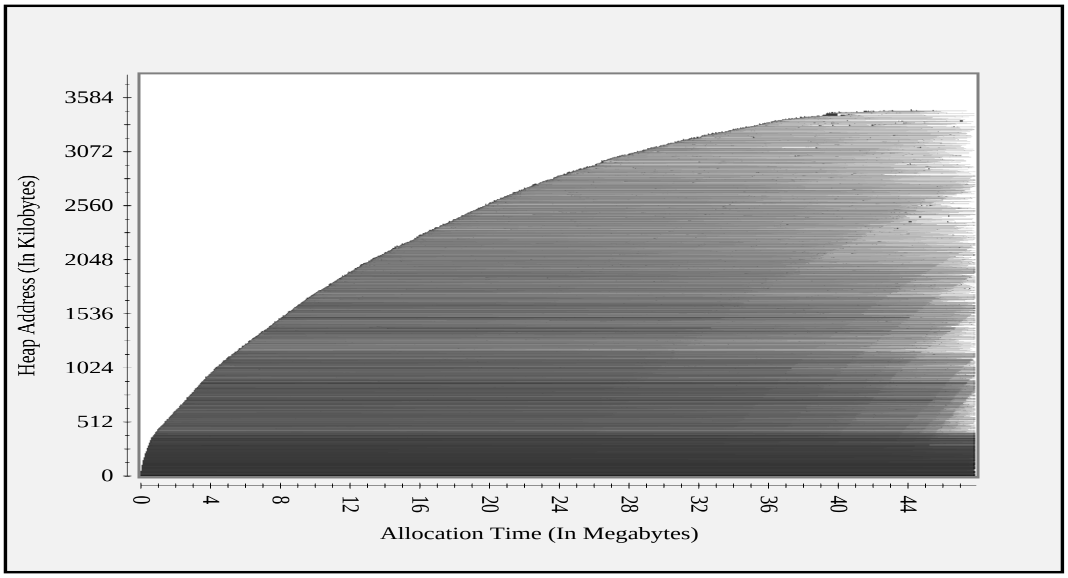

Figure D.26: Fragmentation plot for Ghostscript *shuffled* using the first-fit LIFO no-footer allocator (accounting for all overheads)

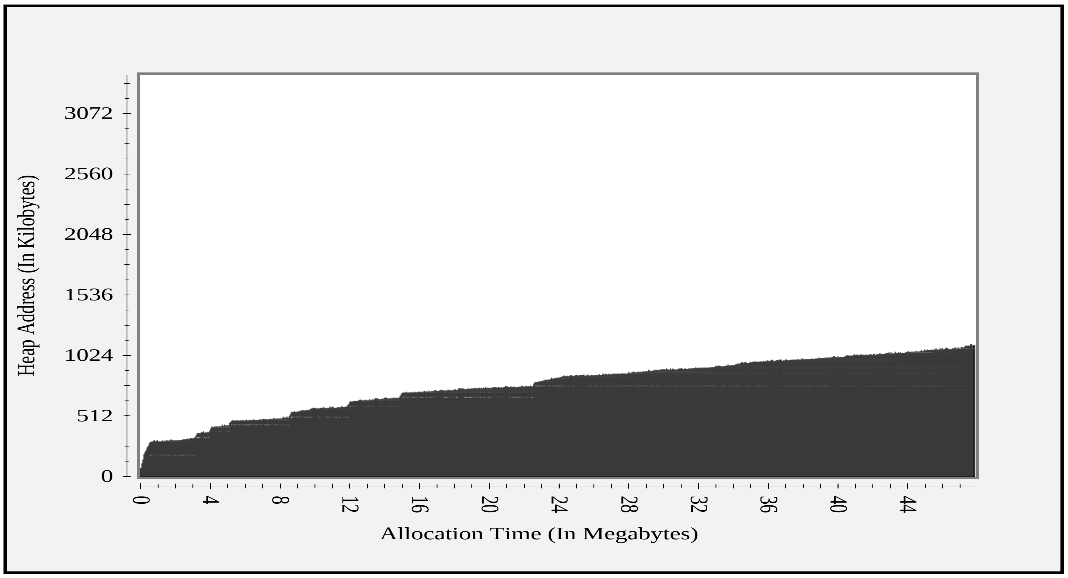

Figure D.27: Fragmentation plot for Ghostscript using the half-fit allocator (accounting for all overheads)

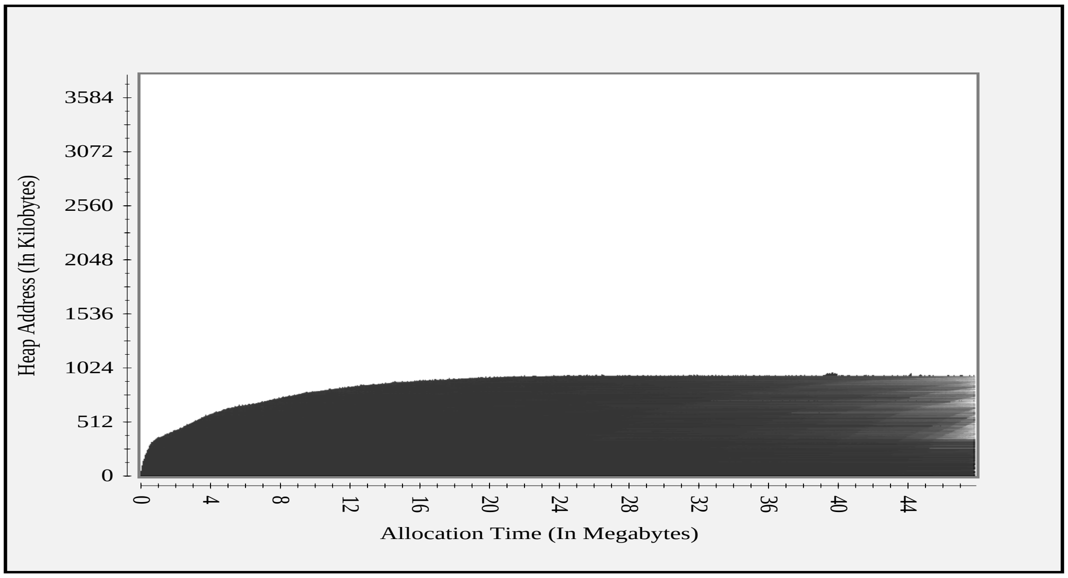

Figure D.28: Fragmentation plot for Ghostscript *shuffled* using the half-fit allocator (accounting for all overheads)

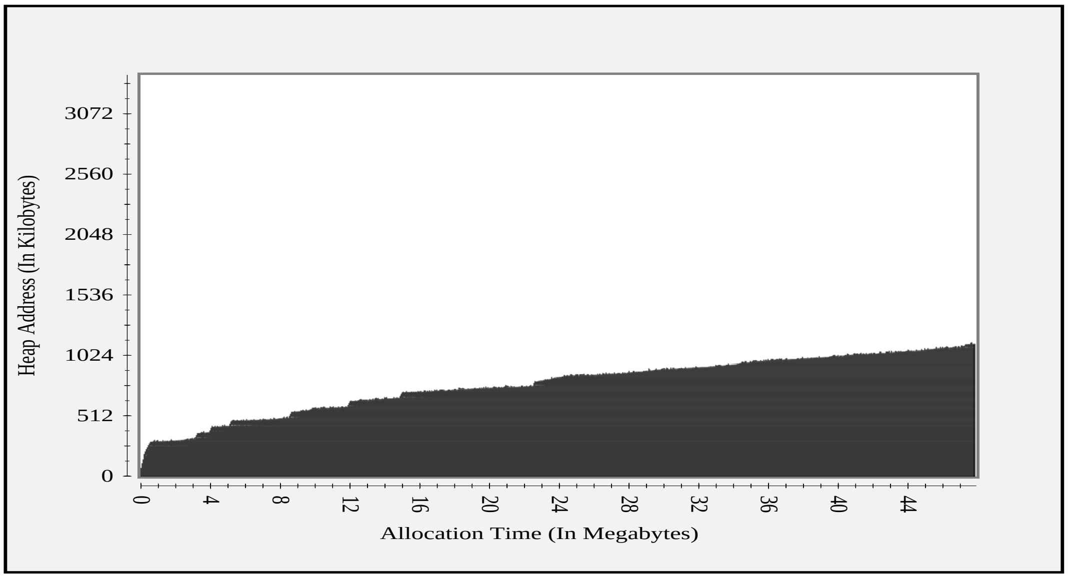

Figure D.29: Fragmentation plot for Ghostscript using Lea's 2.6.1 allocator (accounting for all overheads)

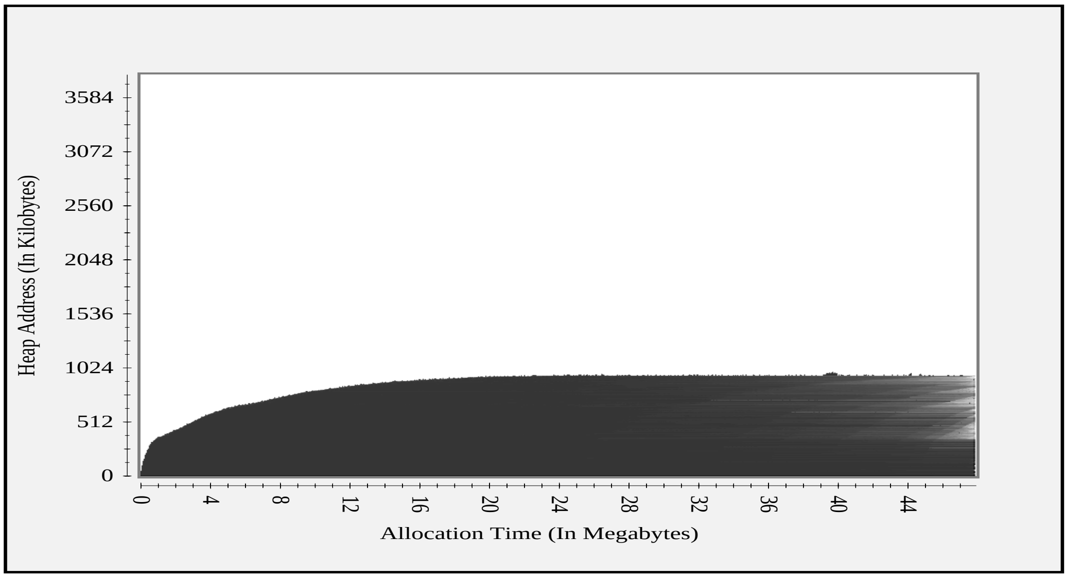

Figure D.30: Fragmentation plot for Ghostscript *shuffled* using Lea's 2.6.1 allocator (accounting for all overheads)

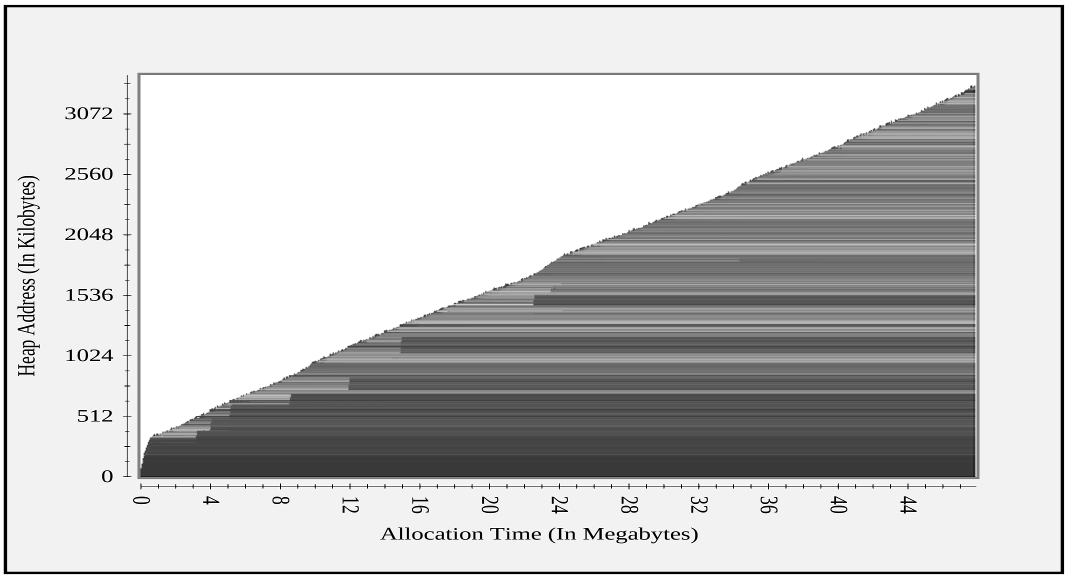

Figure D.31: Fragmentation plot for Ghostscript using the next-fit LIFO no-footer allocator (accounting for all overheads)

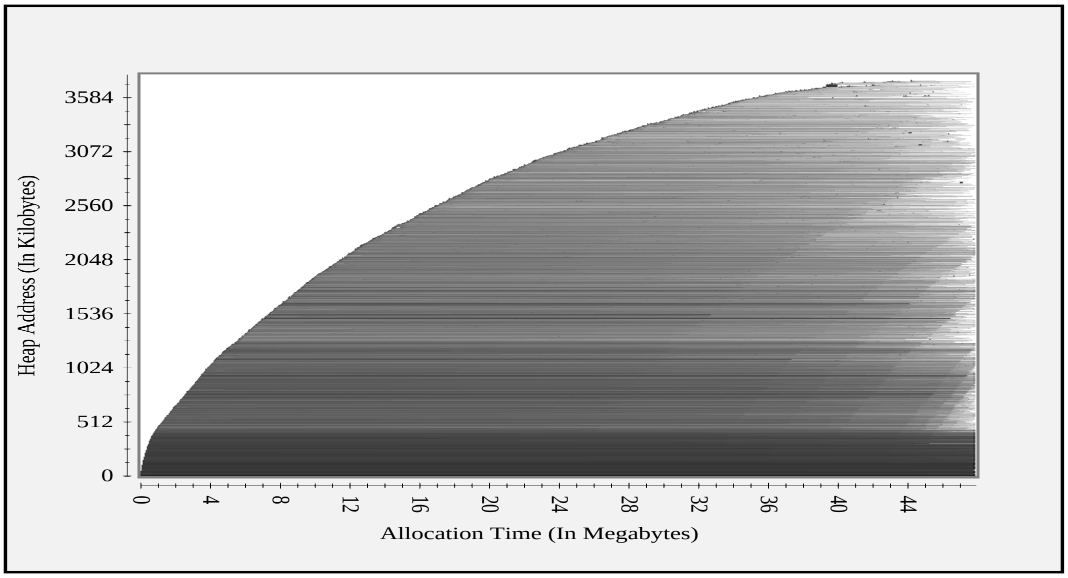

Figure D.32: Fragmentation plot for Ghostscript *shuffled* using the nest-fit LIFO no-footer allocator (accounting for all overheads)

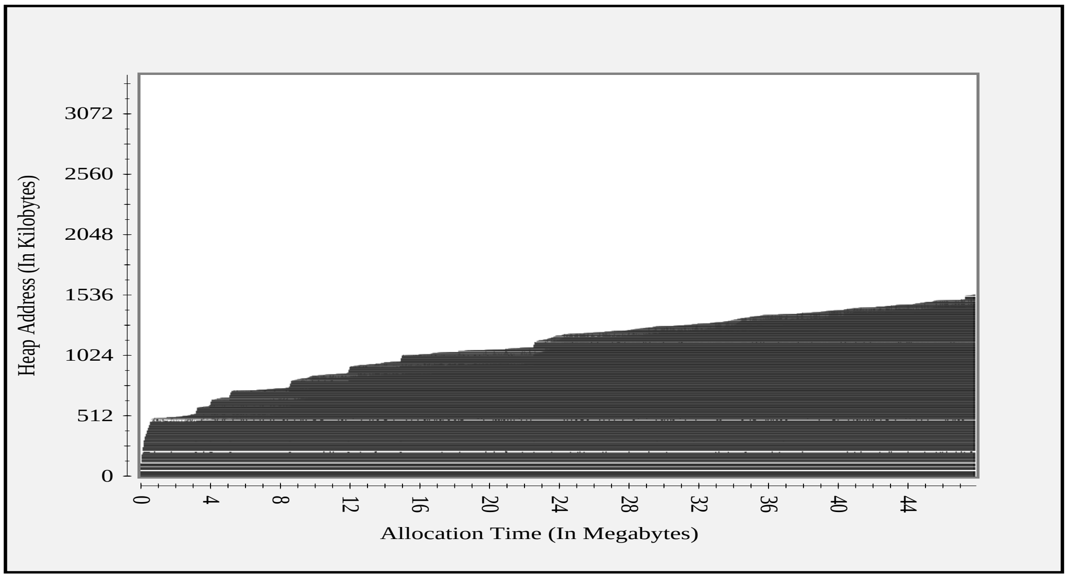

Figure D.33: Fragmentation plot for Ghostscript using the simple segregated storage 2^N allocator (accounting for all overheads)

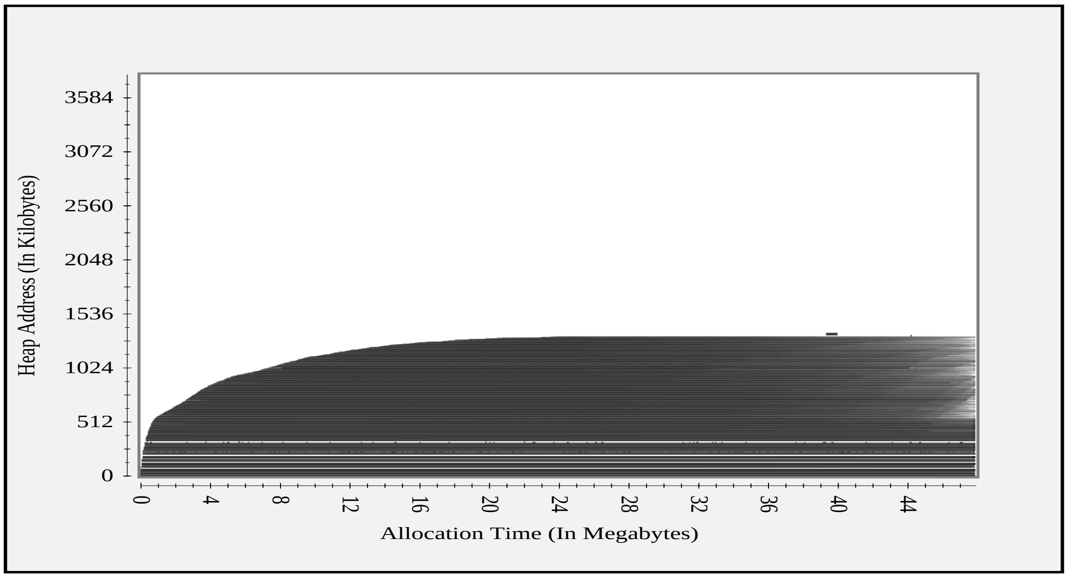

Figure D.34: Fragmentation plot for Ghostscript *shuffled* using the using the simple segregated storage 2^N allocator (accounting for all overheads)

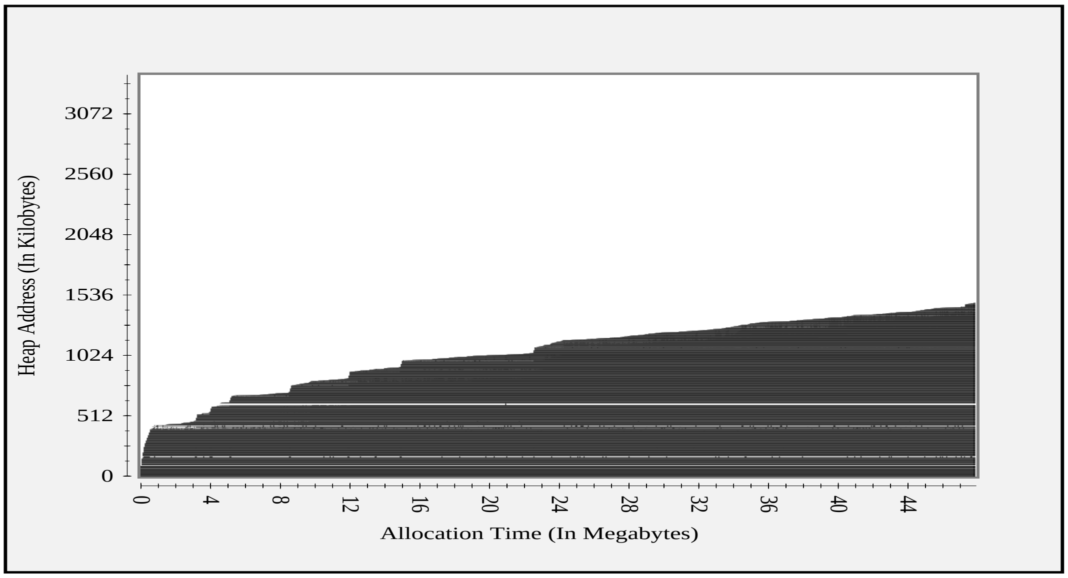

Figure D.35: Fragmentation plot for Ghostscript using the simple segregated storage 2^N & $3 * 2^N$ allocator (accounting for all overheads)

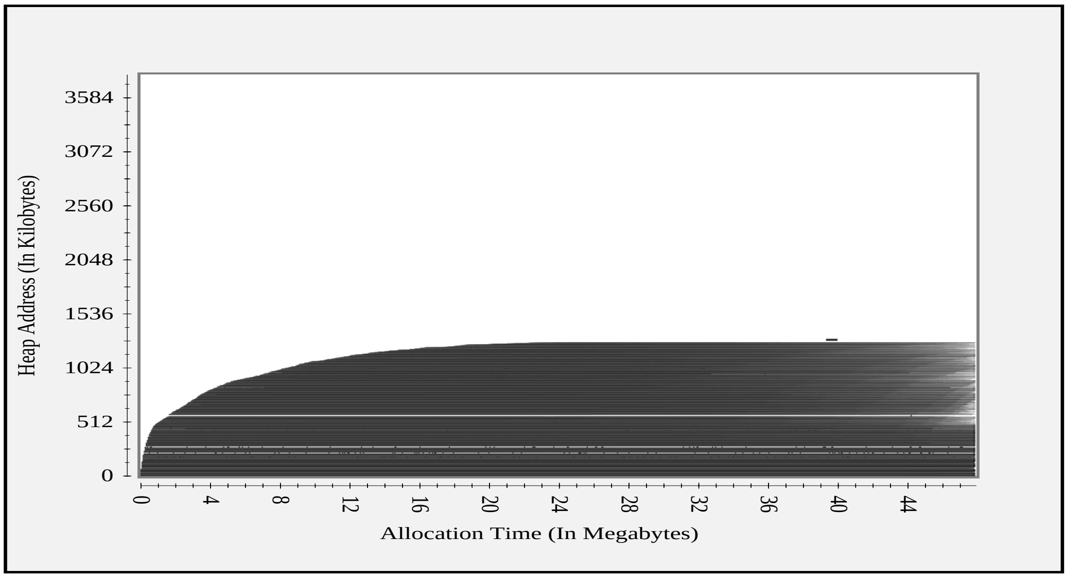

Figure D.36: Fragmentation plot for Ghostscript *shuffled* using the simple segregated storage 2^N $3 * 2^N$ allocator (accounting for all overheads)

Appendix E

Locality Plots for Selected Allocators

The graphs that follow are memory access plots of the form first shown in Section 3.9. These pictures show loads and stores to heap locations over time for each of the selected allocators. As was described in Section 3.9, darker pixels indicate a larger number of loads and stores to a particular region of memory at a particular point in time.

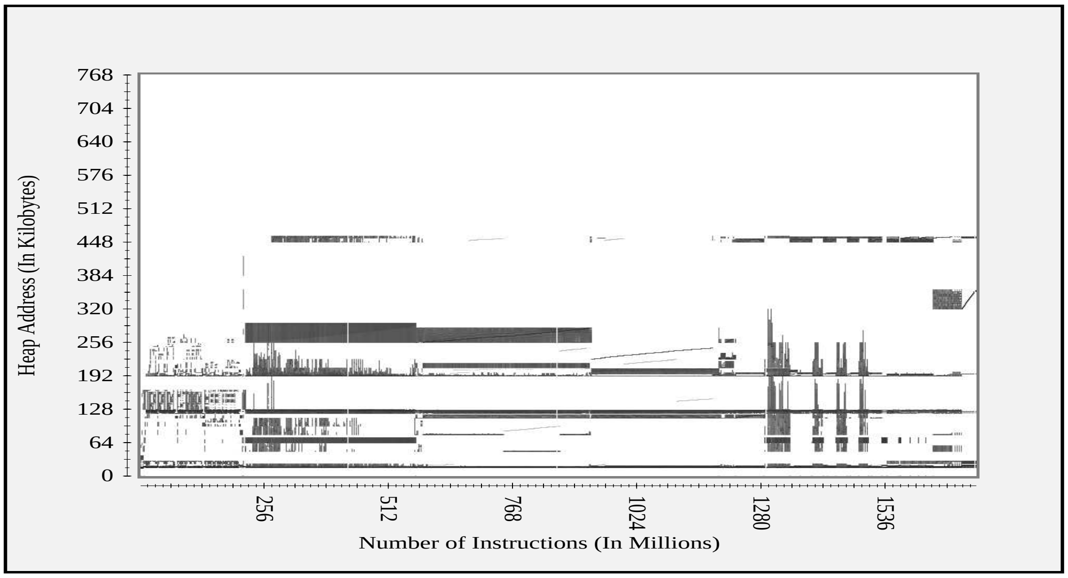

Figure E.1: Memory access plot for Espresso using the binary-buddy allocator

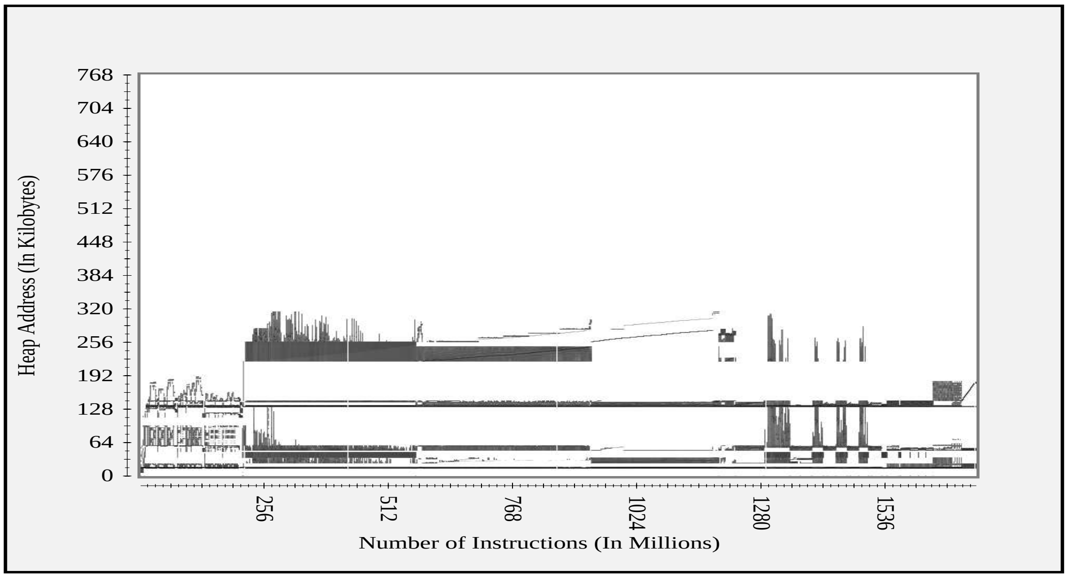

Figure E.2: Memory access plot for Espresso using the best-fit LIFO no-footer allocator

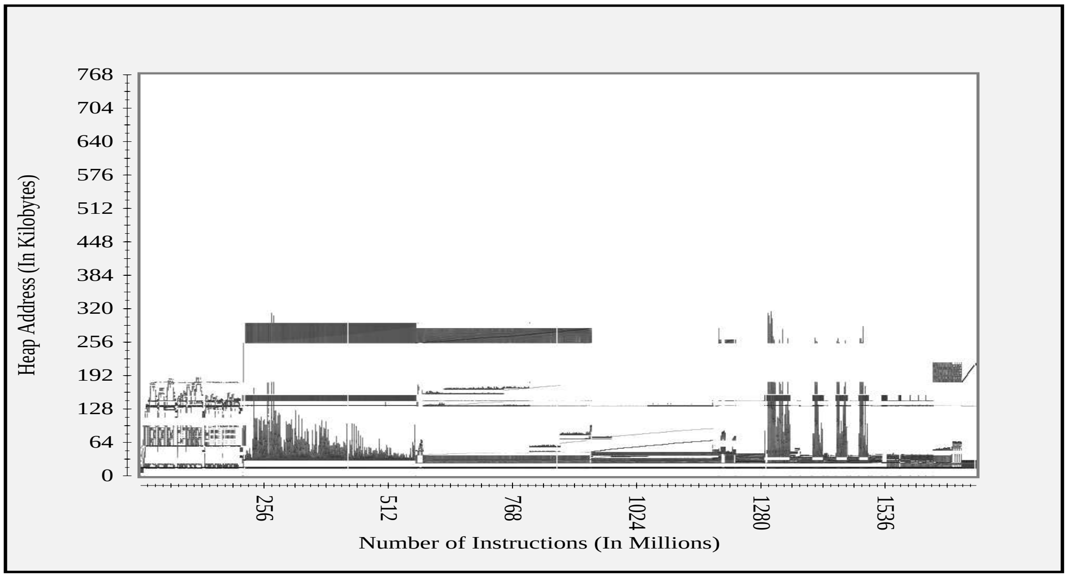

Figure E.3: Memory access plot for Espresso using the first-fit address-ordered no-footer allocator

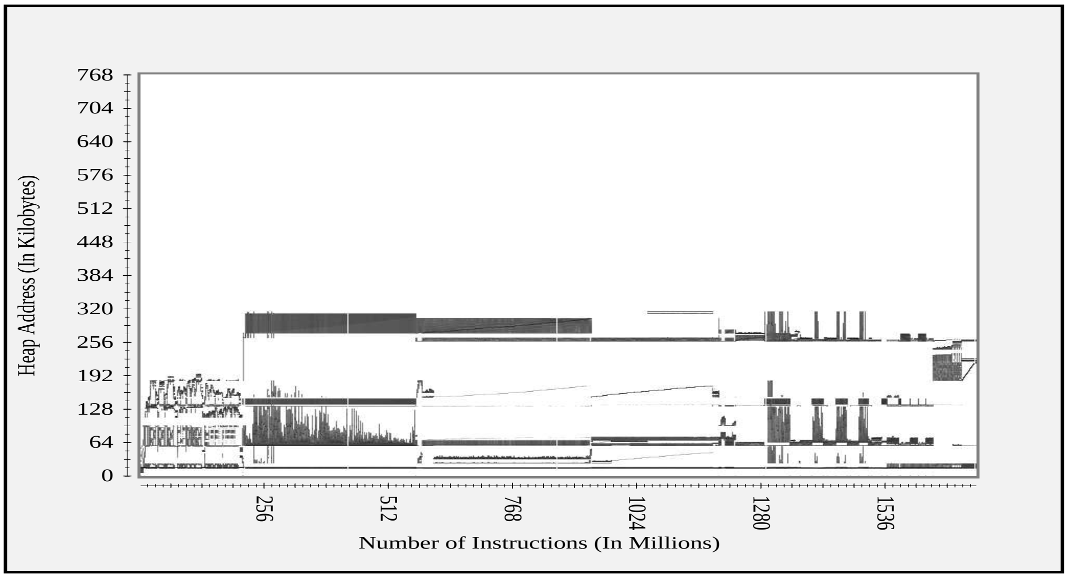

Figure E.4: Memory access plot for Espresso using the first-fit LIFO no-footer allocator

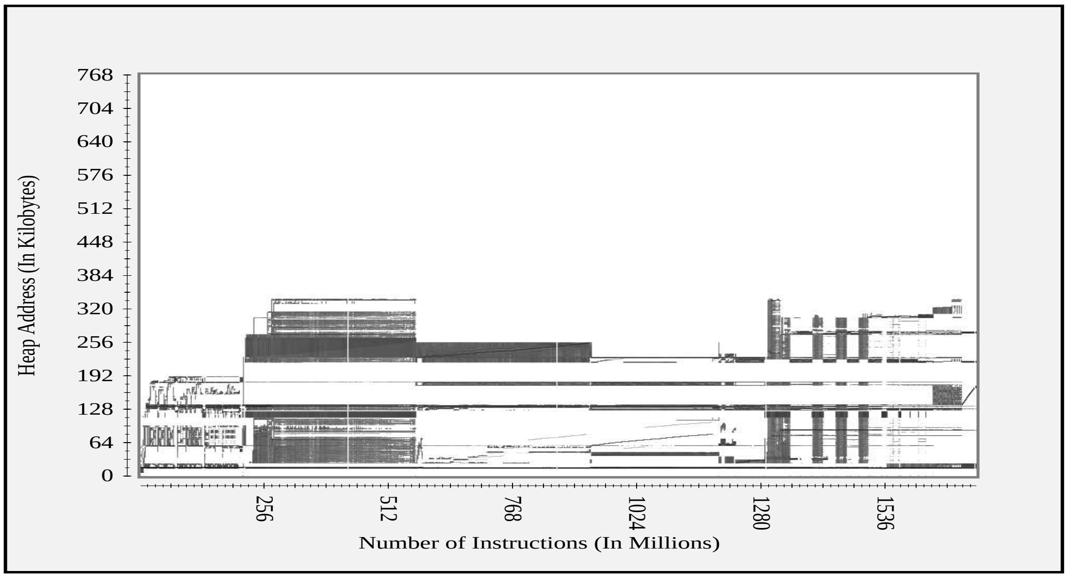

Figure E.5: Memory access plot for Espresso using the half-fit allocator

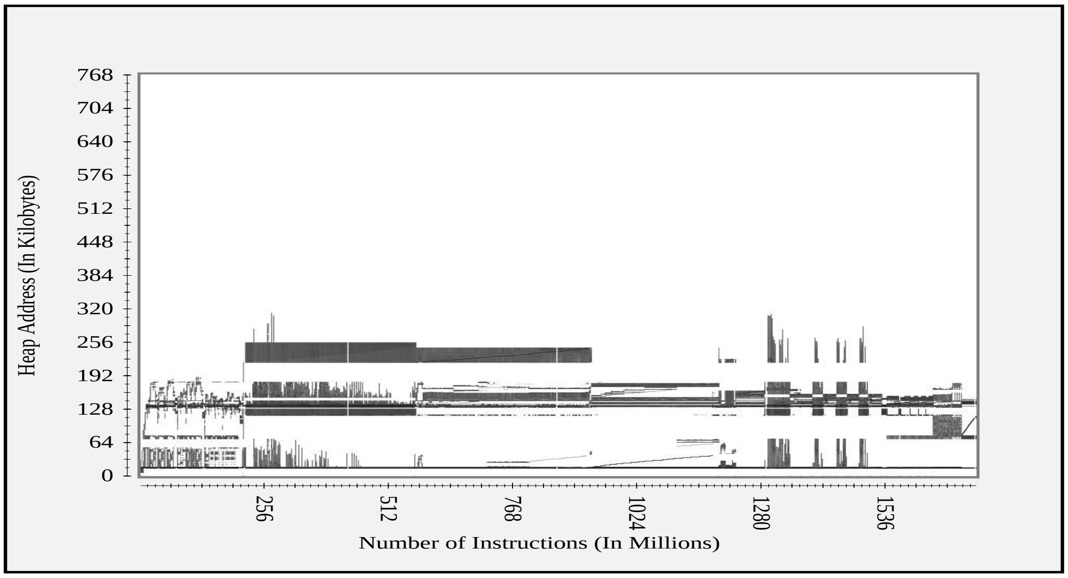

Figure E.6: Memory access plot for Espresso using Lea's 2.6.1 allocator

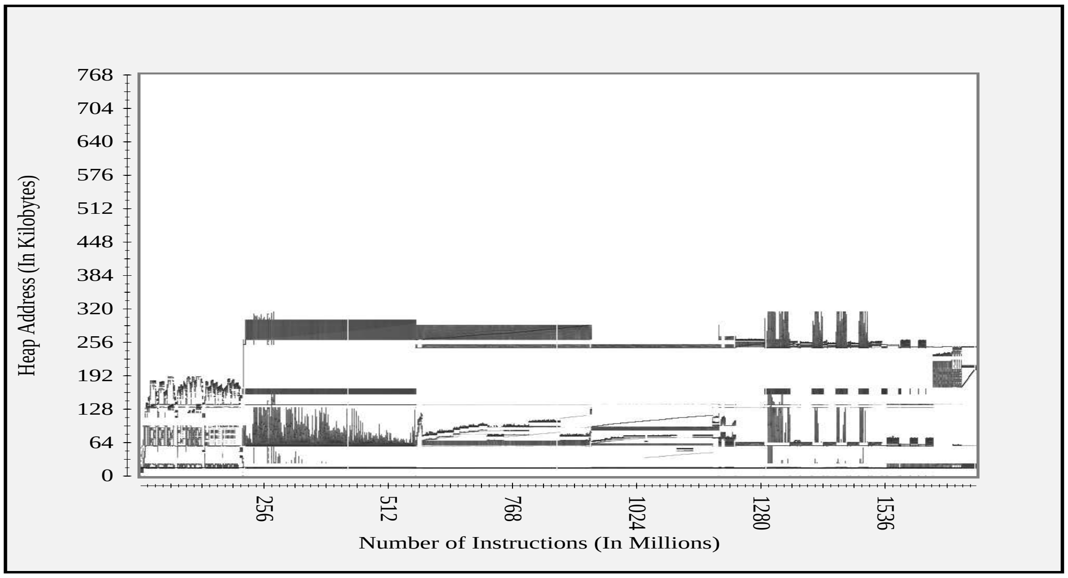

Figure E.7: Memory access plot for Espresso using the next-fit LIFO no-footer allocator

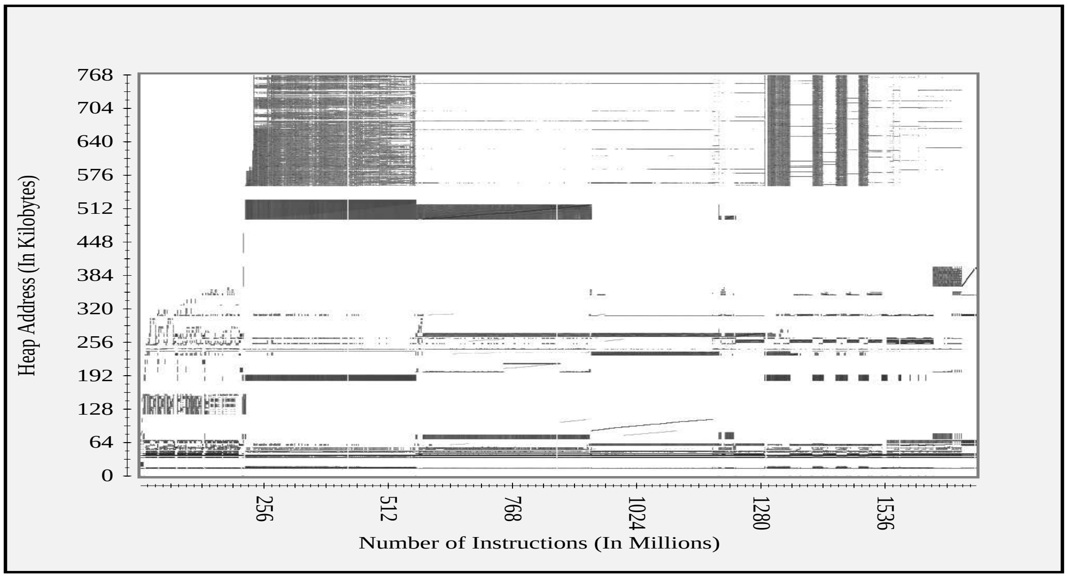

Figure E.8: Memory access plot for Espresso using the simple segregated storage 2^N allocator

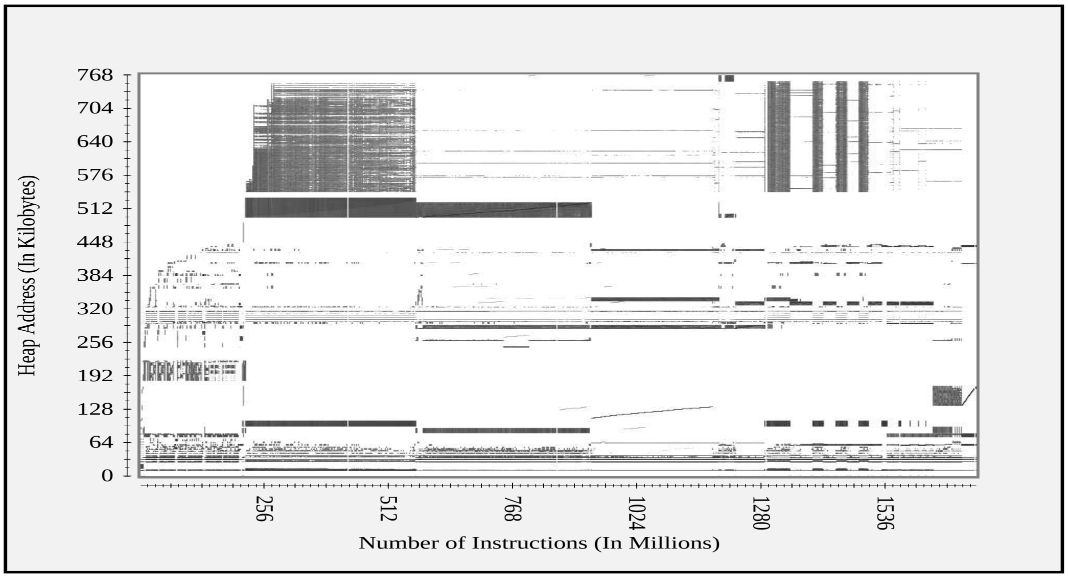

Figure E.9: Memory access plot for Espresso using the simple segregated storage 2^N & $3*2^N$ allocator

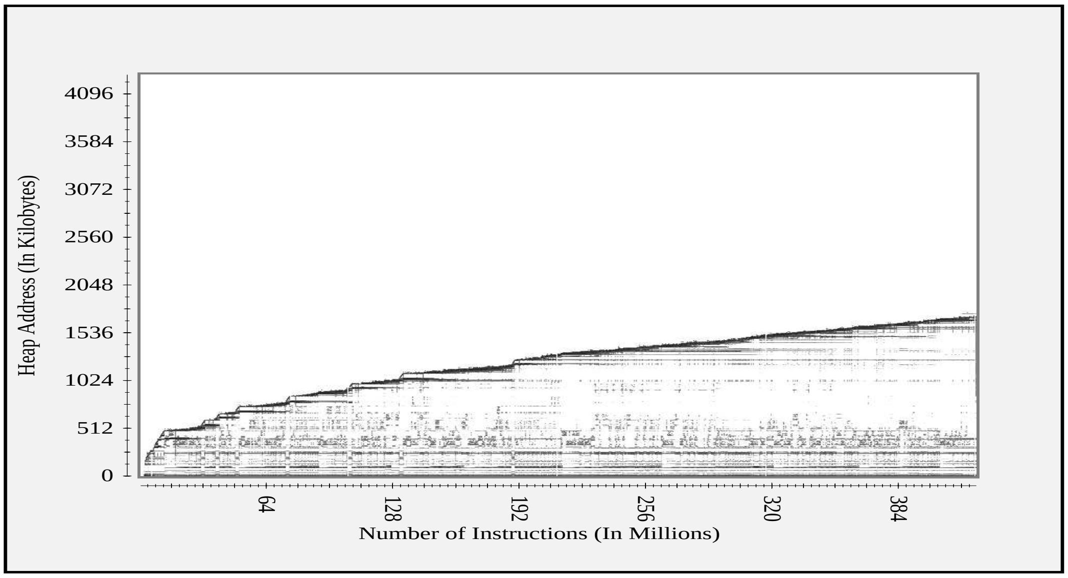

Figure E.10: Memory access plot for Ghostscript using the binary-buddy allocator

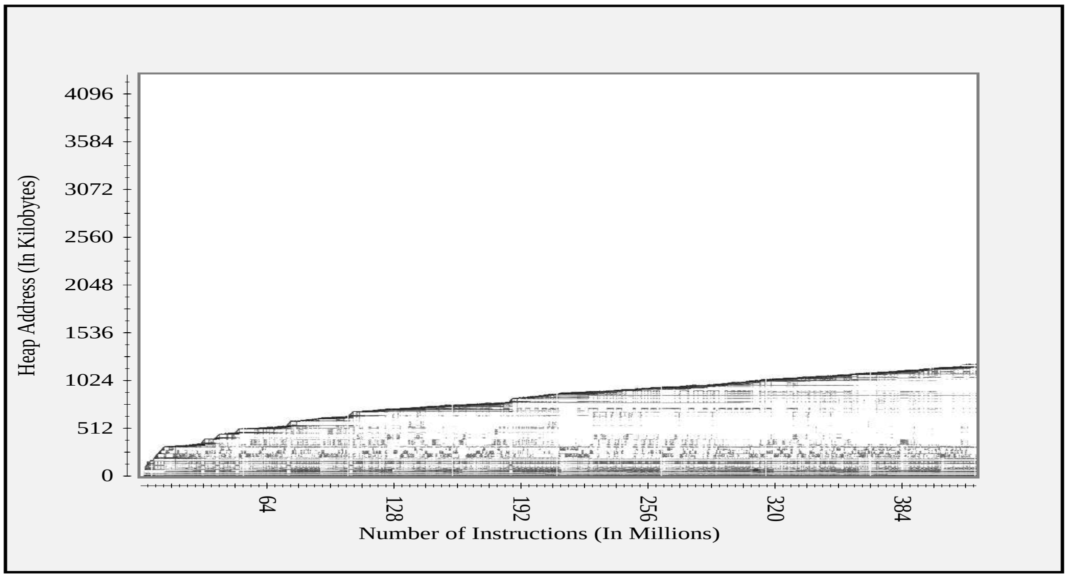

Figure E.11: Memory access plot for Ghostscript using the best-fit LIFO no-footer allocator

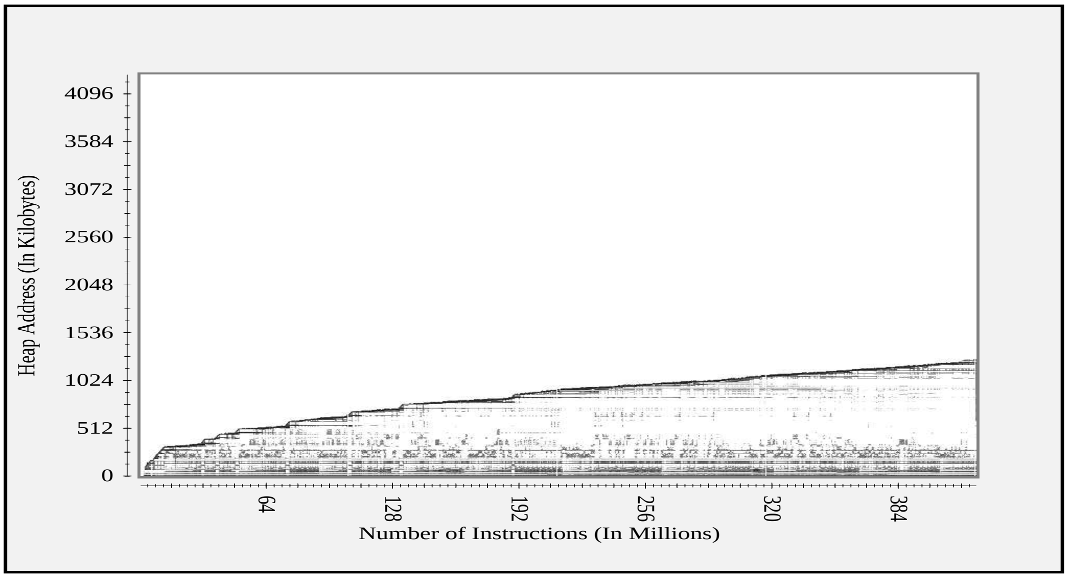

Figure E.12: Memory access plot for Ghostscript using the first-fit address-ordered no-footer allocator

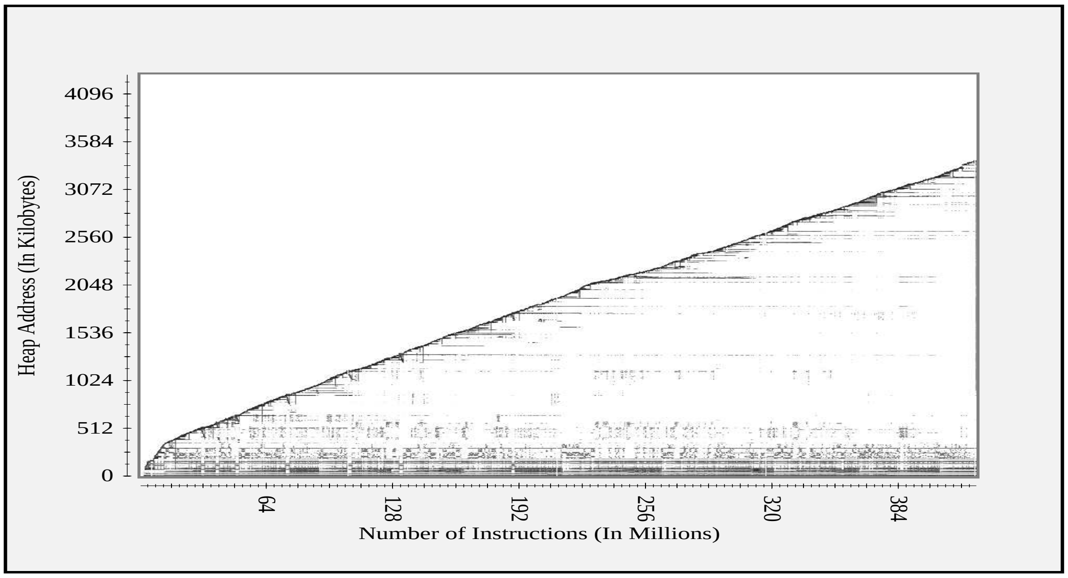

Figure E.13: Memory access plot for Ghostscript using the first-fit LIFO no-footer allocator

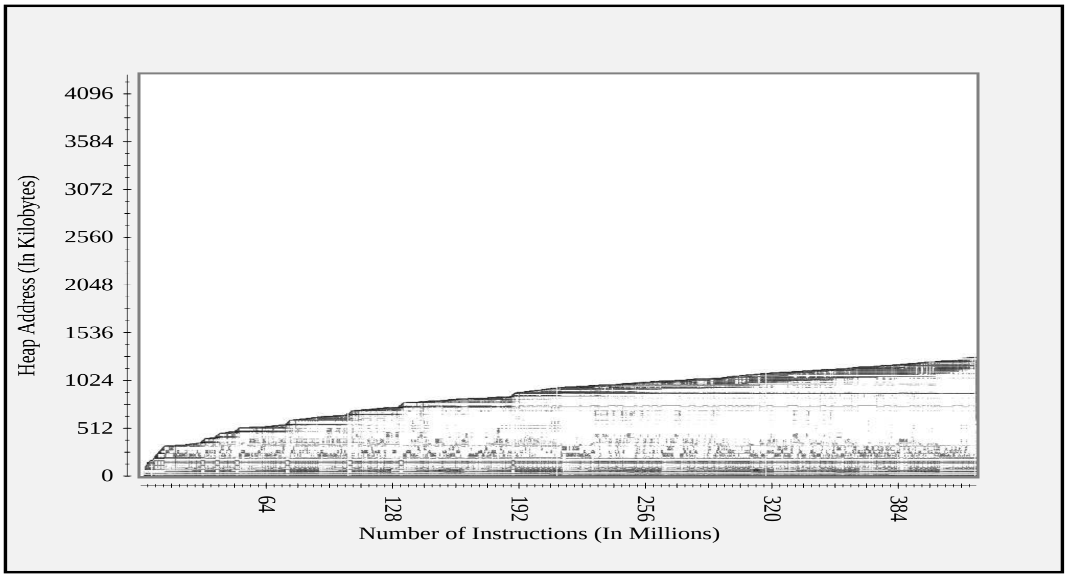

Figure E.14: Memory access plot for Ghostscript using the half-fit allocator

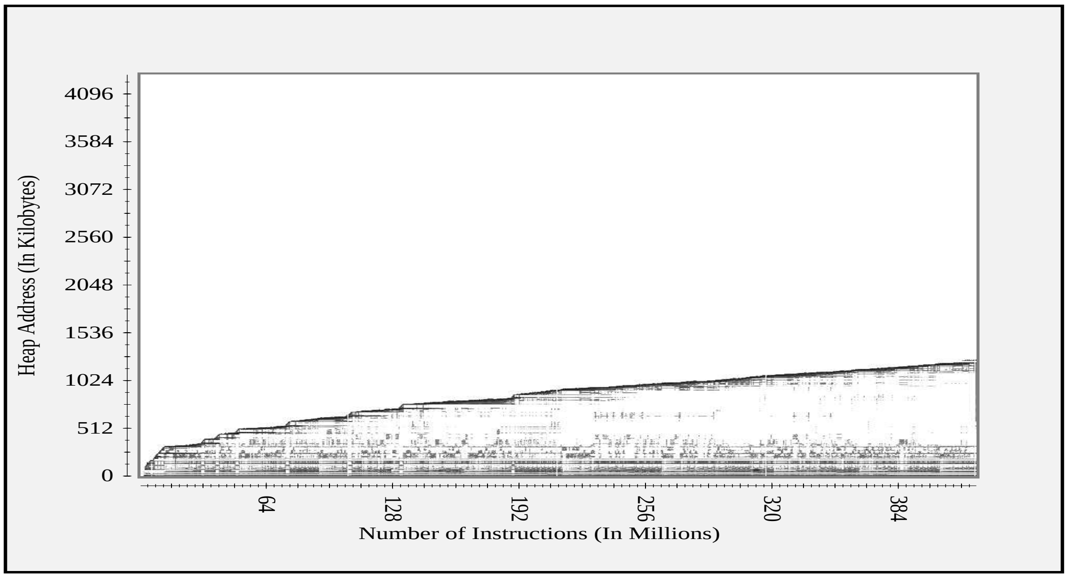

Figure E.15: Memory access plot for Ghostscript using Lea's 2.6.1 allocator

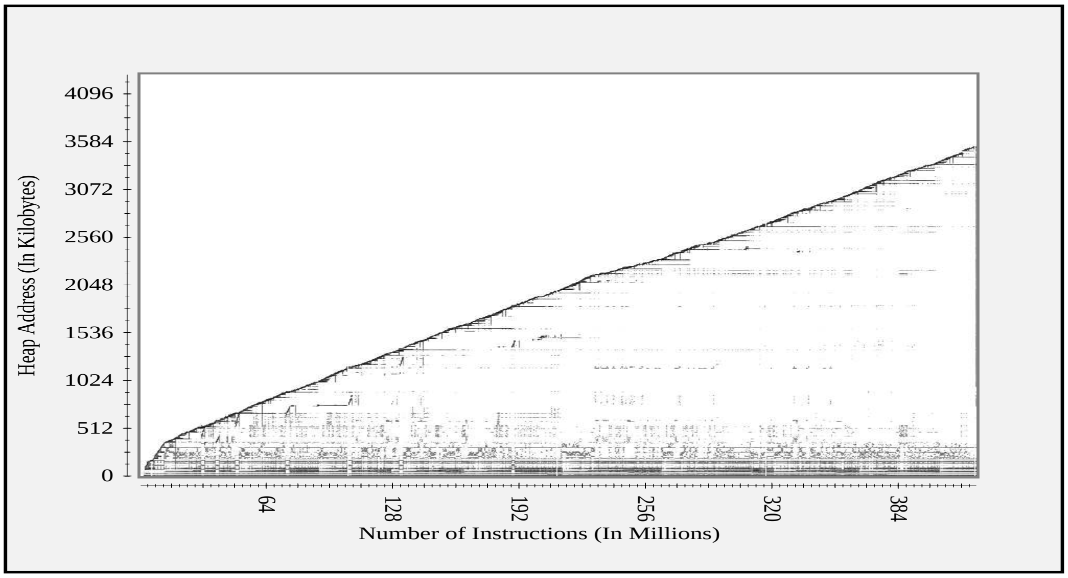

Figure E.16: Memory access plot for Ghostscript using the next-fit LIFO no-footer allocator

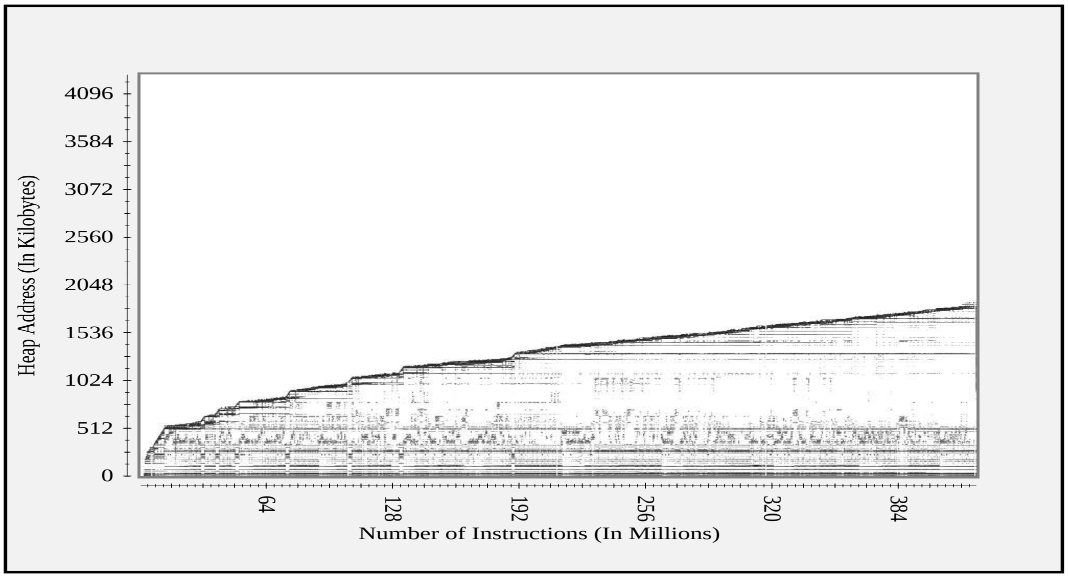

Figure E.17: Memory access plot for Ghostscript using the simple segregated storage 2^N allocator

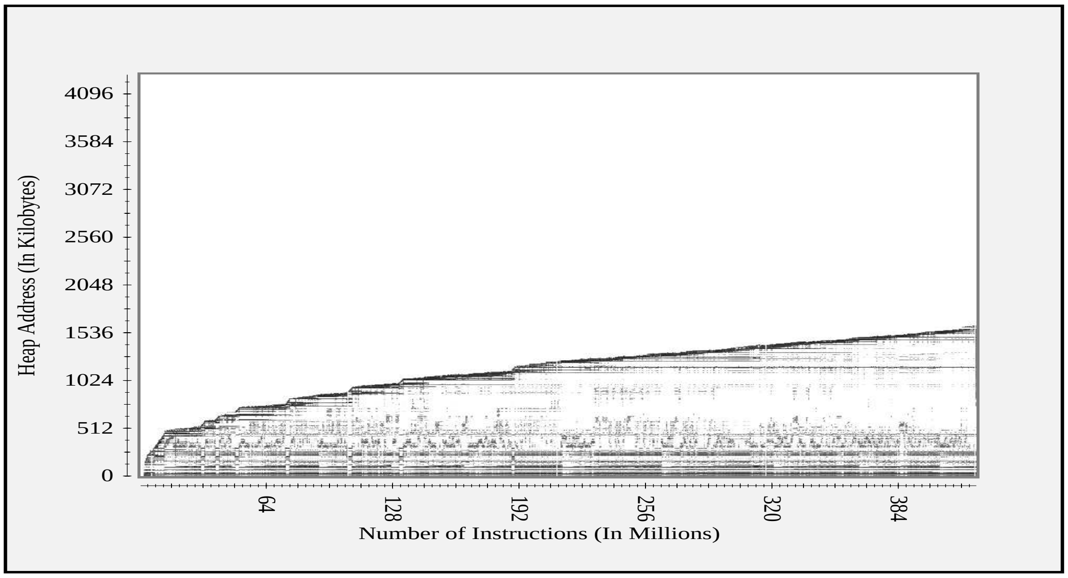

Figure E.18: Memory access plot for Ghostscript using the simple segregated storage 2^N & $3 * 2^N$ allocator

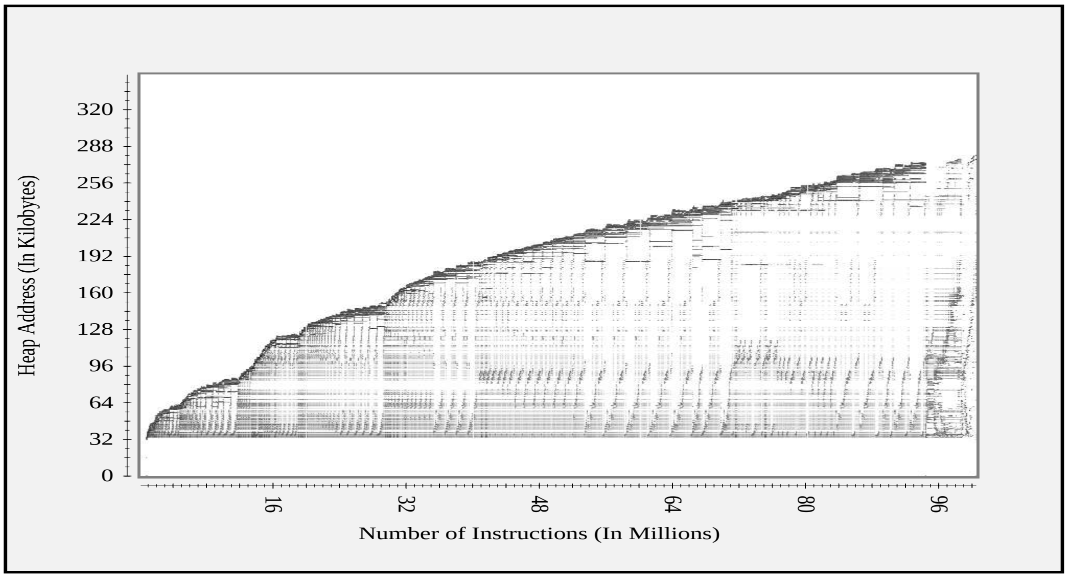

Figure E.19: Memory access plot for Grobner using the binary-buddy allocator

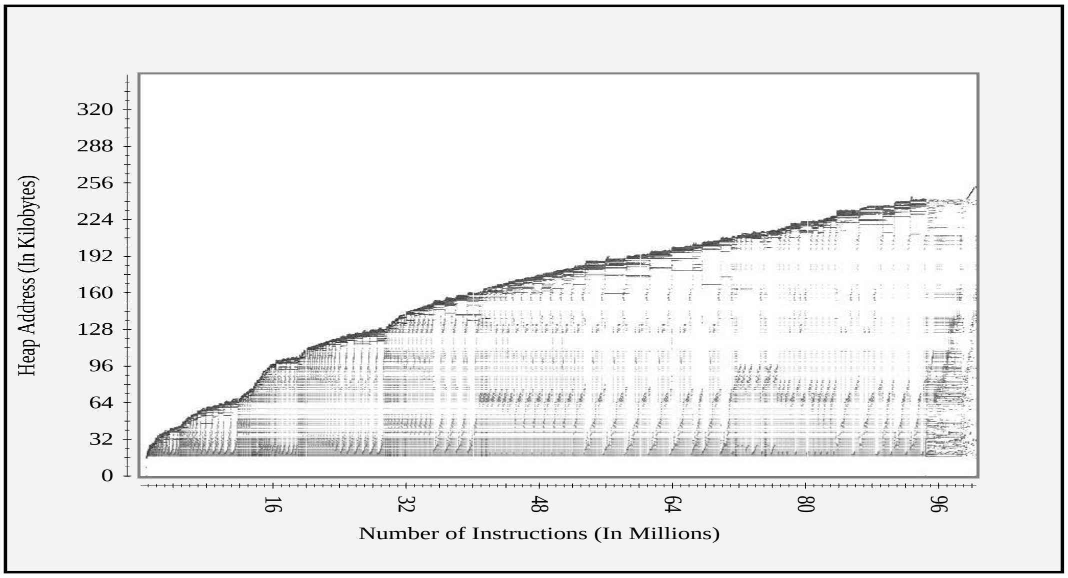

Figure E.20: Memory access plot for Grobner using the best-fit LIFO no-footer allocator

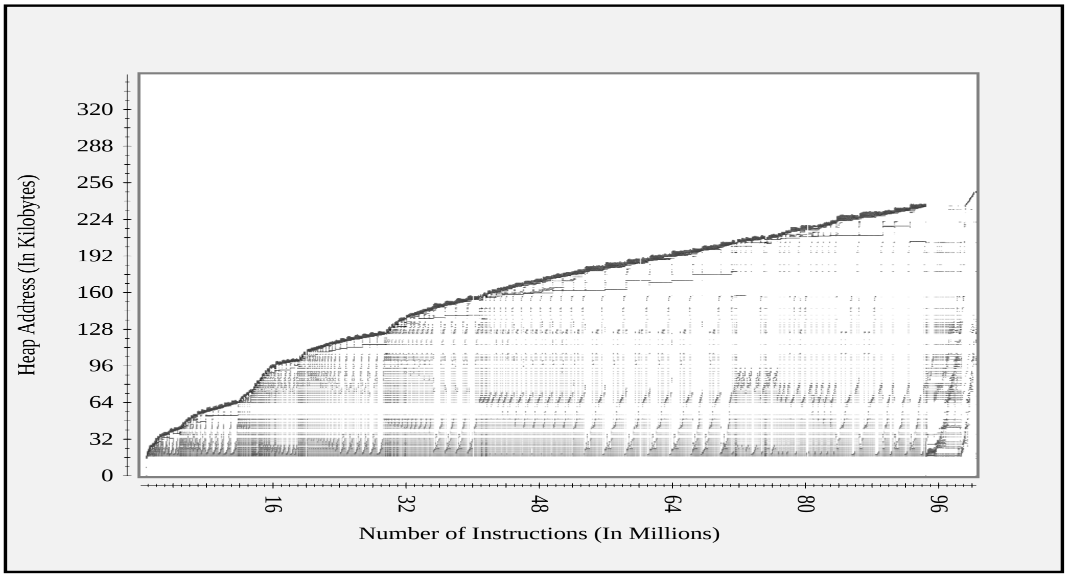

Figure E.21: Memory access plot for Grobner using the first-fit address-ordered no-footer allocator

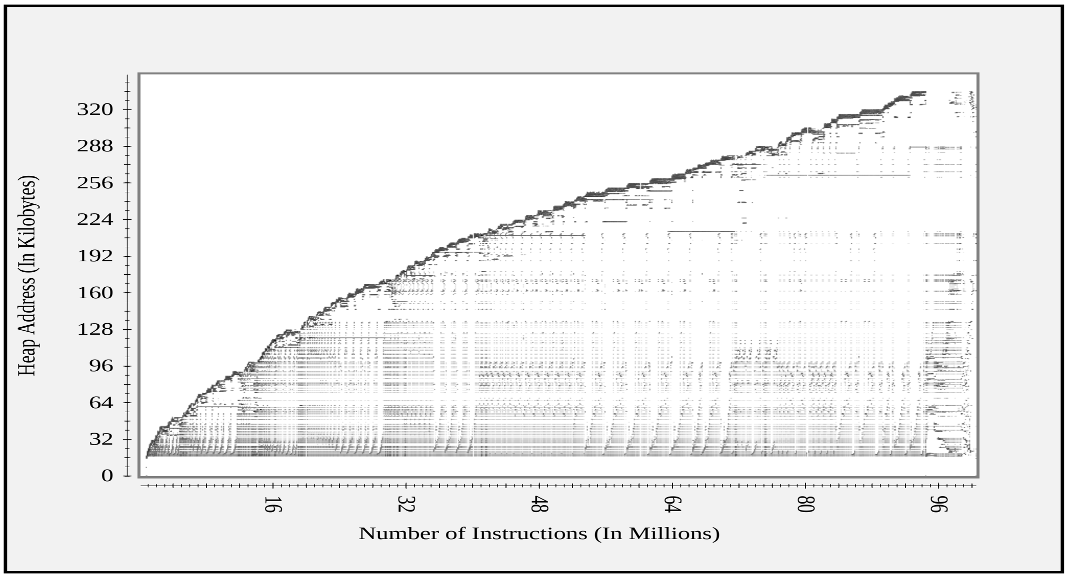

Figure E.22: Memory access plot for Grobner using the first-fit LIFO no-footer allocator

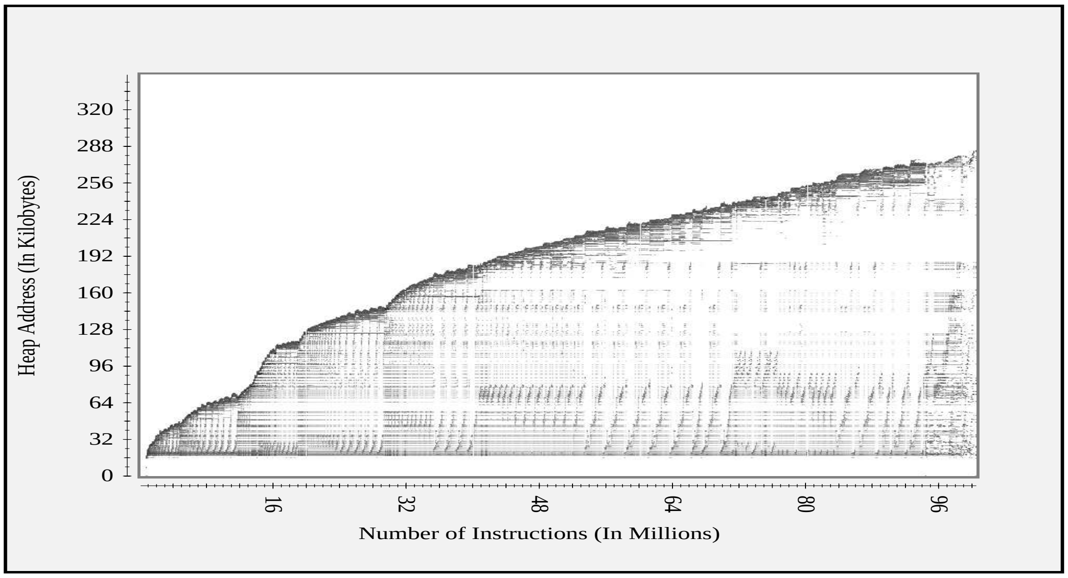

Figure E.23: Memory access plot for Grobner using the half-fit allocator

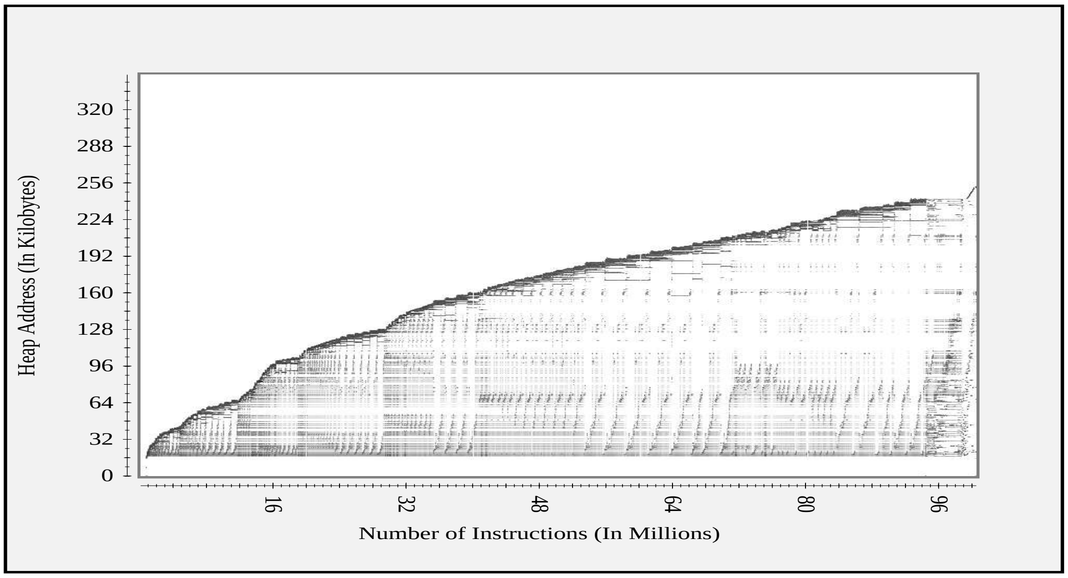

Figure E.24: Memory access plot for Grobner using Lea's 2.6.1 allocator

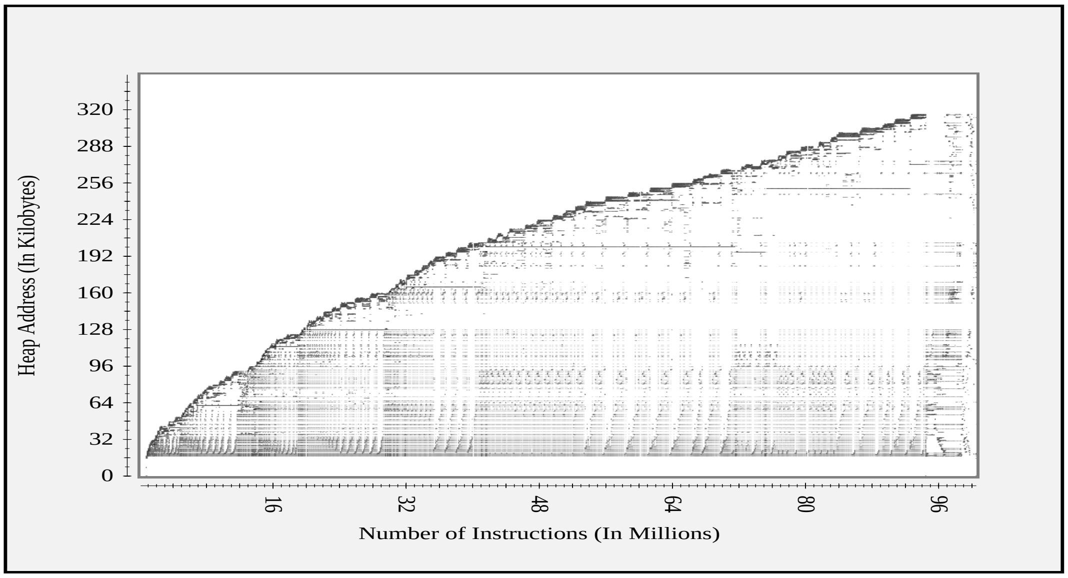

Figure E.25: Memory access plot for Grobner using the next-fit LIFO no-footer allocator

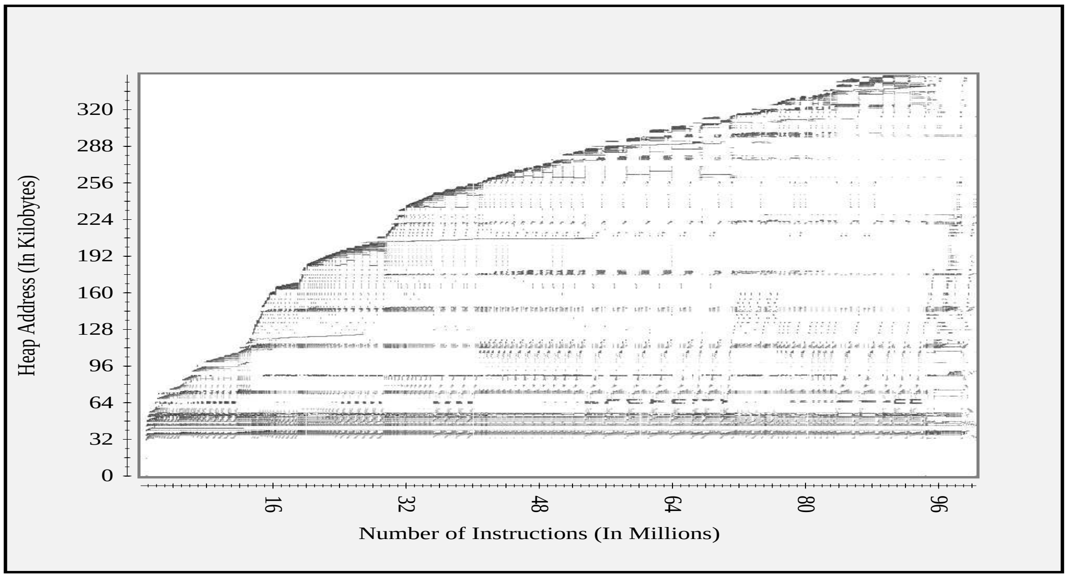

Figure E.26: Memory access plot for Grobner using the simple segregated storage 2^N allocator

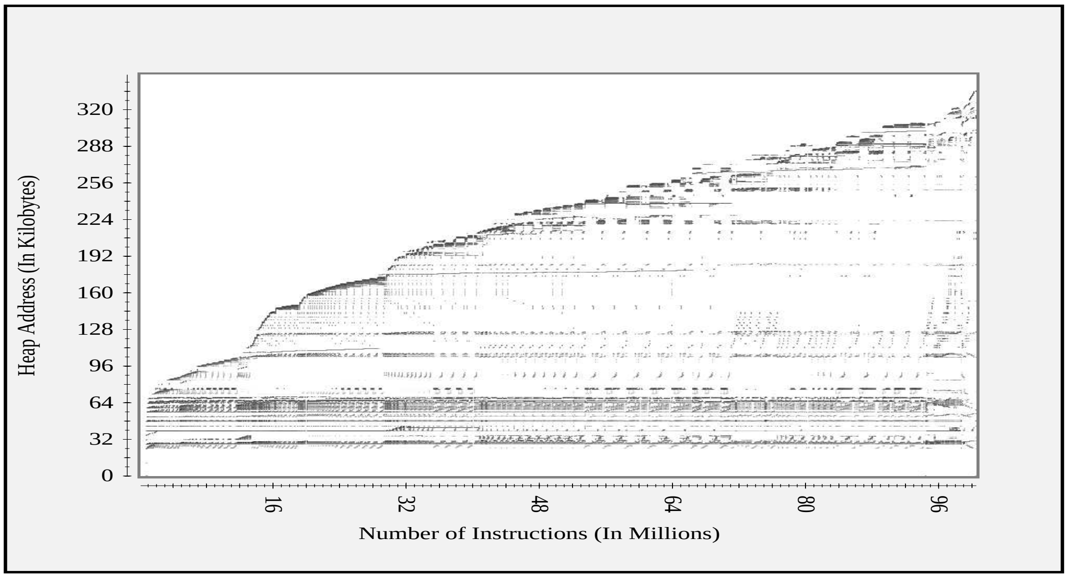

Figure E.27: Memory access plot for Grobner using the simple segregated storage 2^N & $3*2^N$ allocator

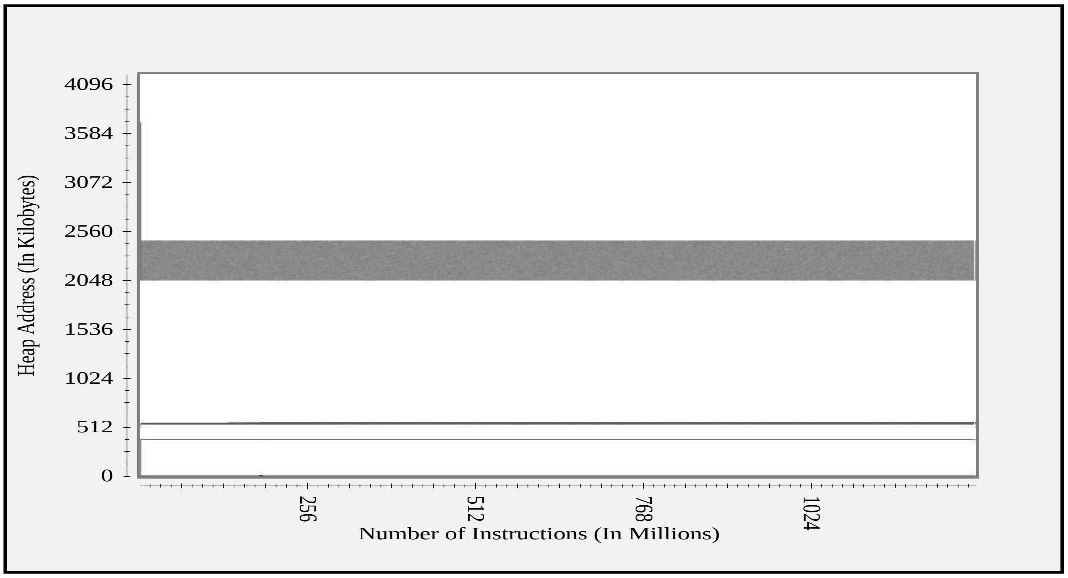

Figure E.28: Memory access plot for Hyper using the binary-buddy allocator

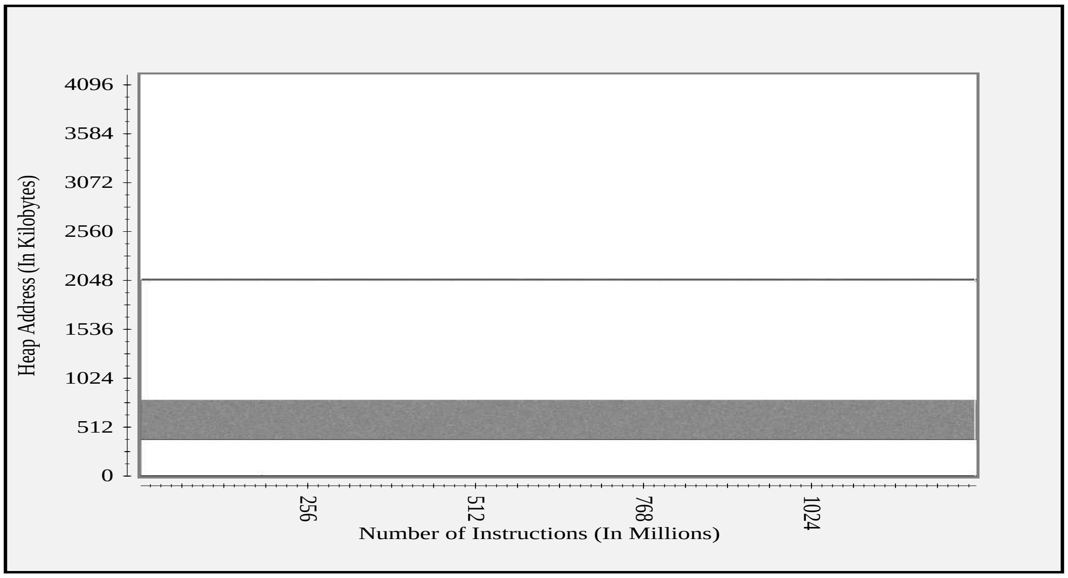

Figure E.29: Memory access plot for Hyper using the best-fit LIFO no-footer allocator

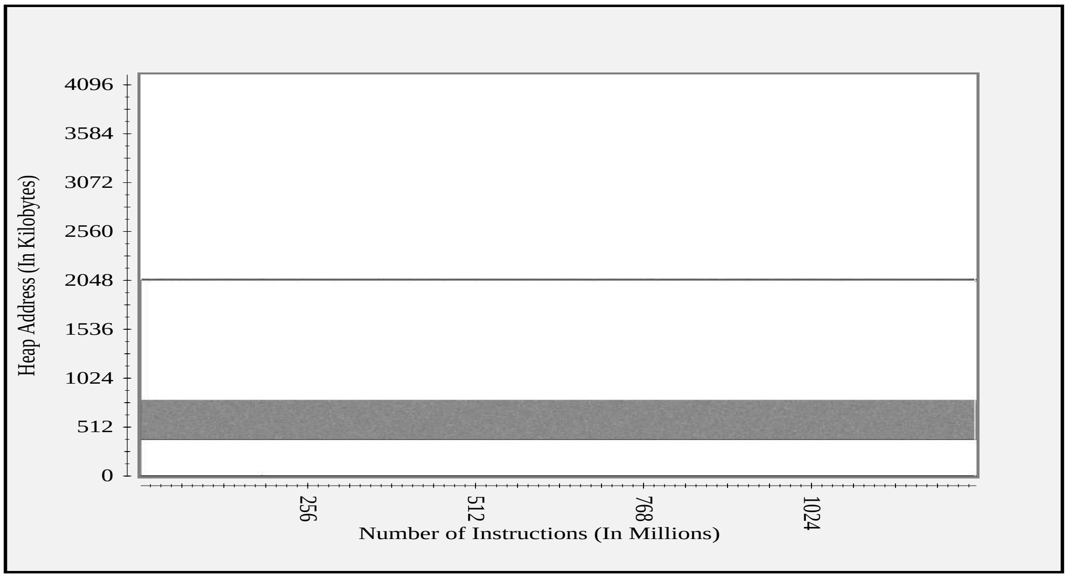

Figure E.30: Memory access plot for Hyper using the first-fit address-ordered no-footer allocator

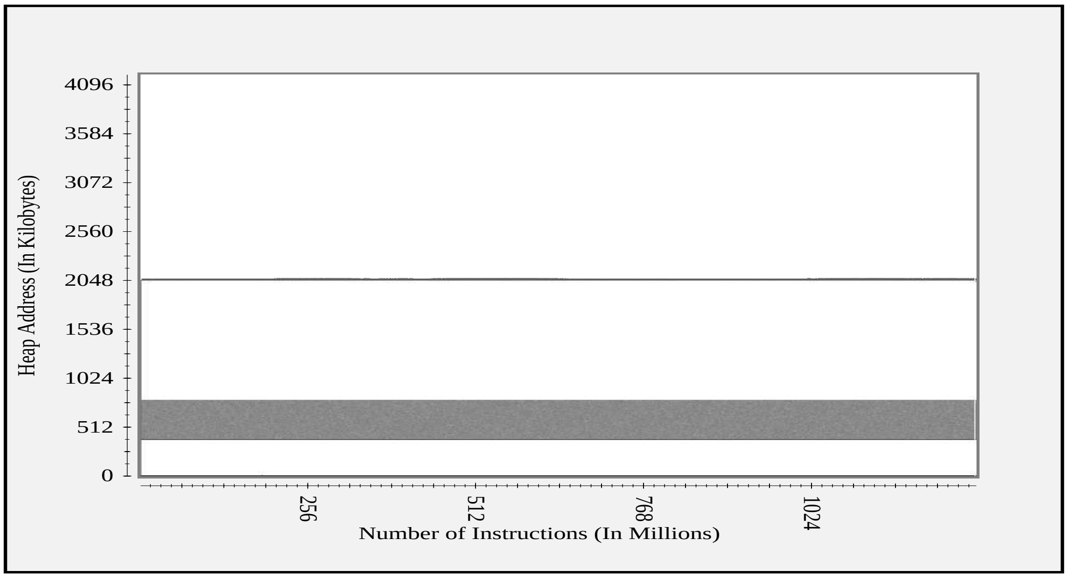

Figure E.31: Memory access plot for Hyper using the first-fit LIFO no-footer allocator

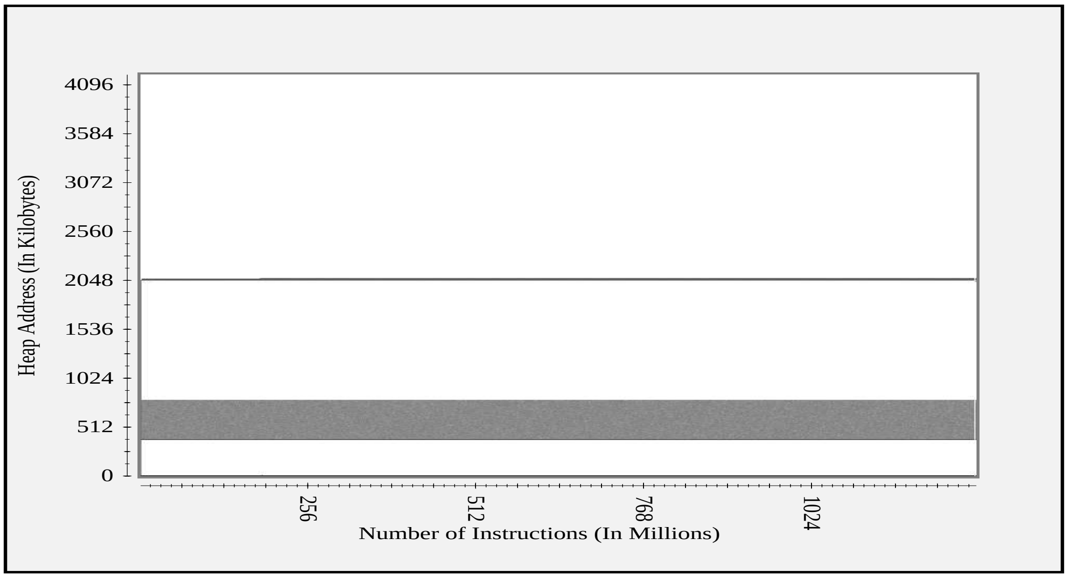

Figure E.32: Memory access plot for Hyper using the half-fit allocator

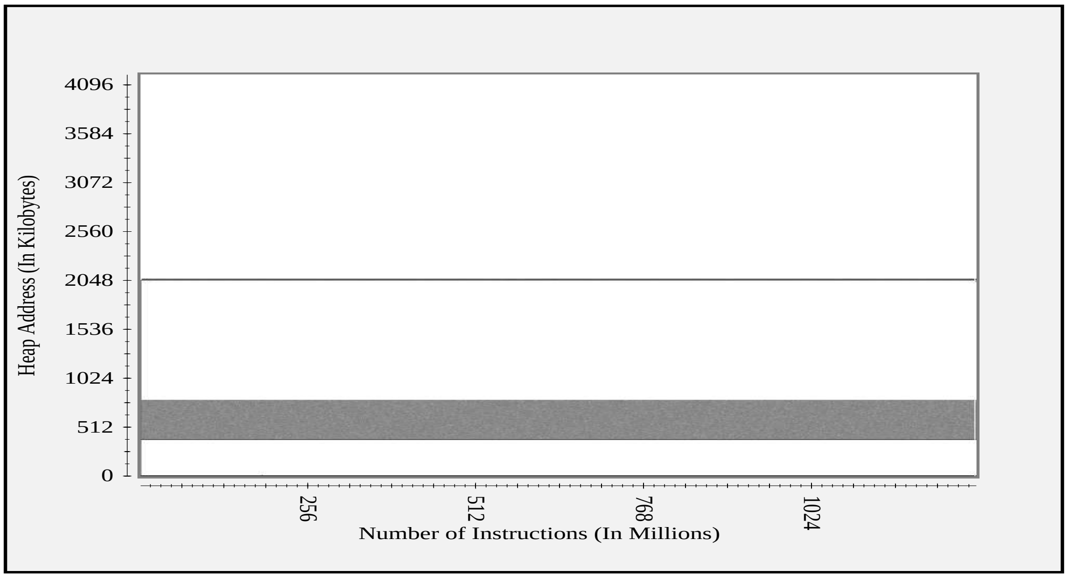

Figure E.33: Memory access plot for Hyper using Lea's 2.6.1 allocator

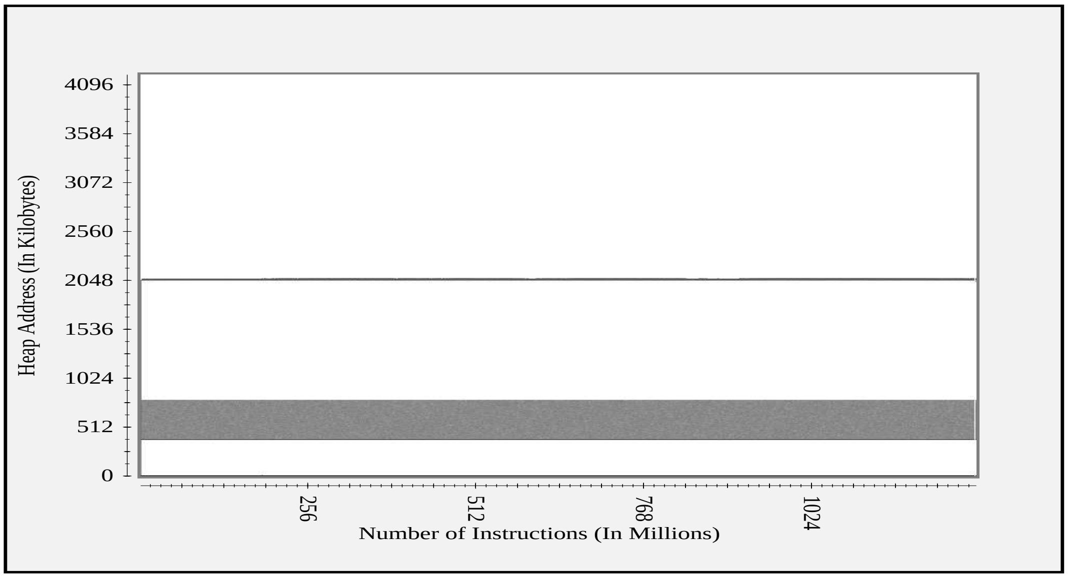

Figure E.34: Memory access plot for Hyper using the next-fit LIFO no-footer allocator

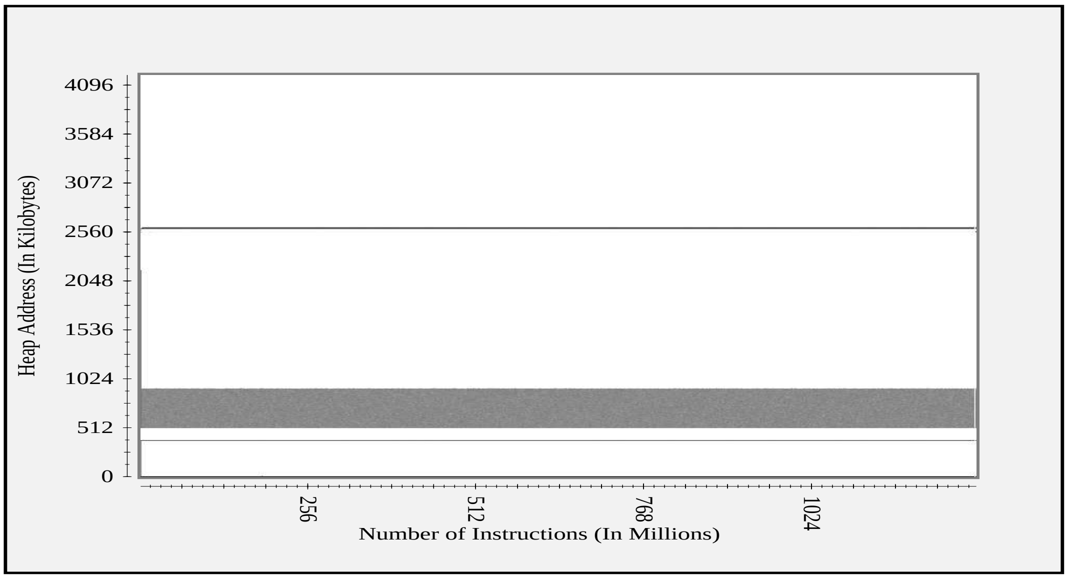

Figure E.35: Memory access plot for Hyper using the simple segregated storage 2^N allocator

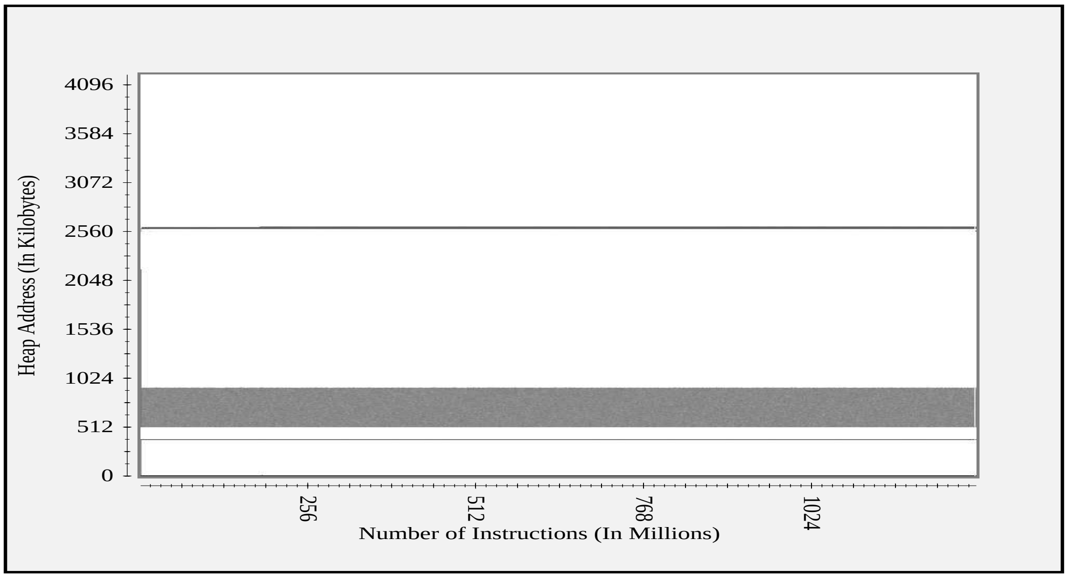

Figure E.36: Memory access plot for Hyper using the simple segregated storage 2^N & $3 * 2^N$ allocator

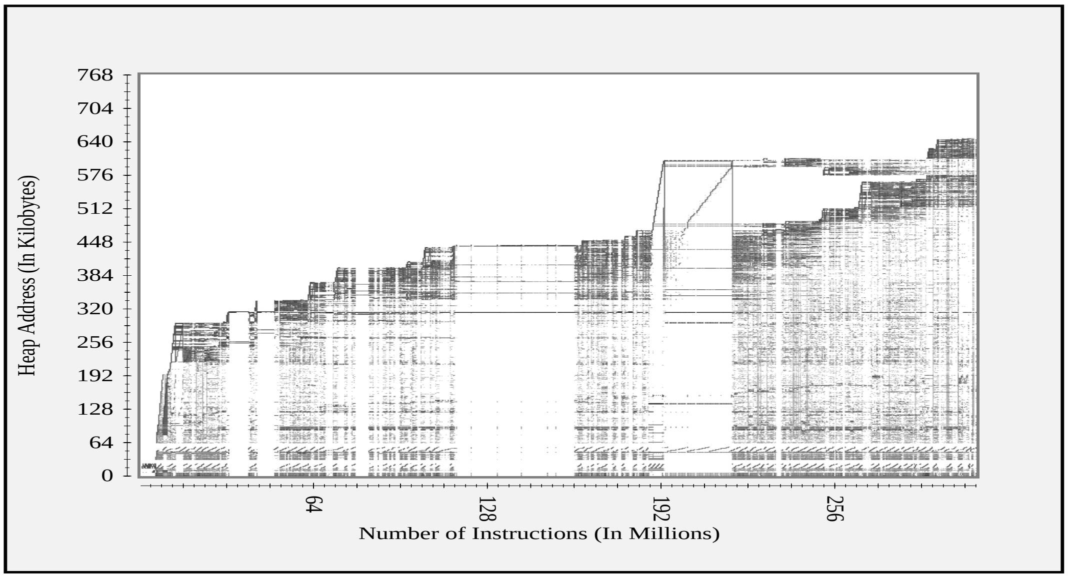

Figure E.37: Memory access plot for P2C using the binary-buddy allocator

Figure E.38: Memory access plot for P2C using the best-fit LIFO no-footer allocator

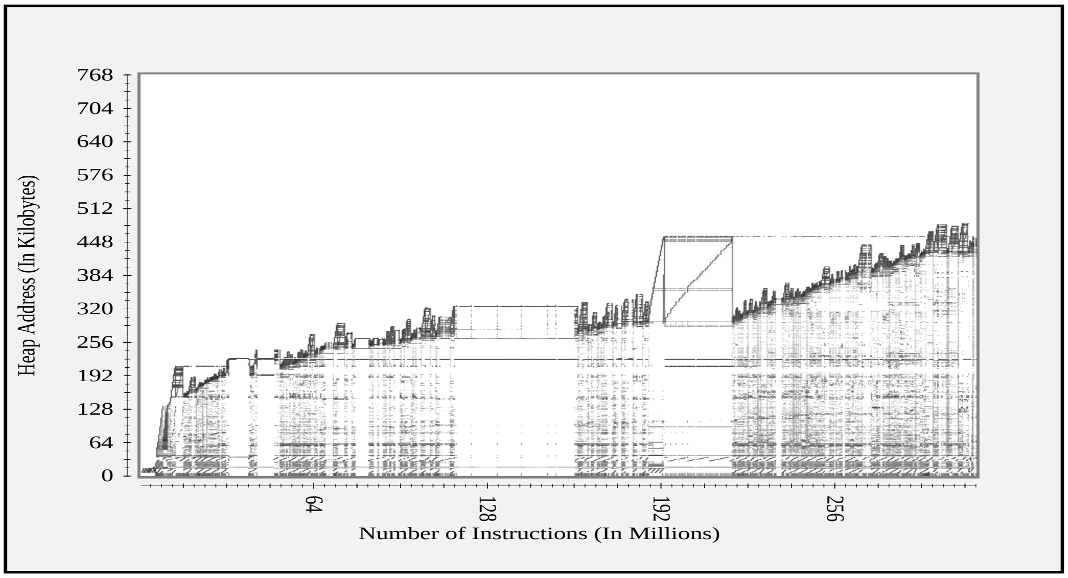

Figure E.39: Memory access plot for P2C using the first-fit address-ordered no-footer allocator

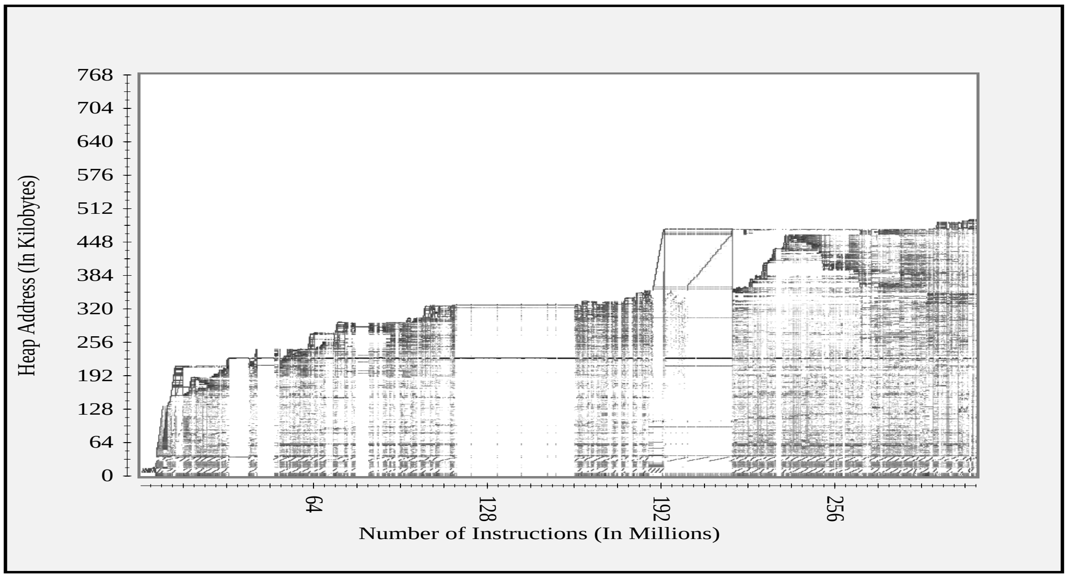

Figure E.40: Memory access plot for P2C using the first-fit LIFO no-footer allocator

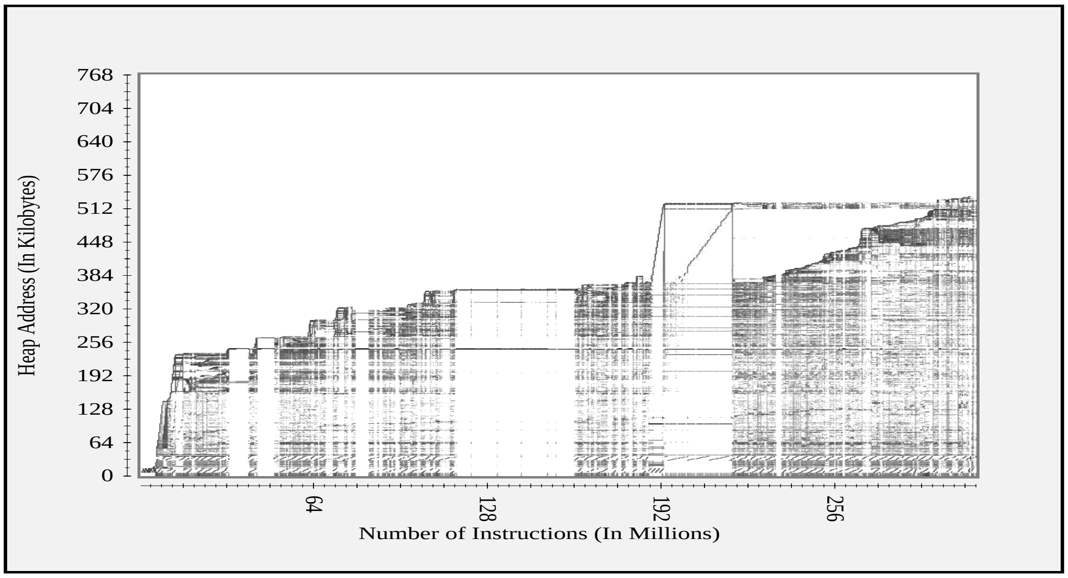

Figure E.41: Memory access plot for P2C using the half-fit allocator

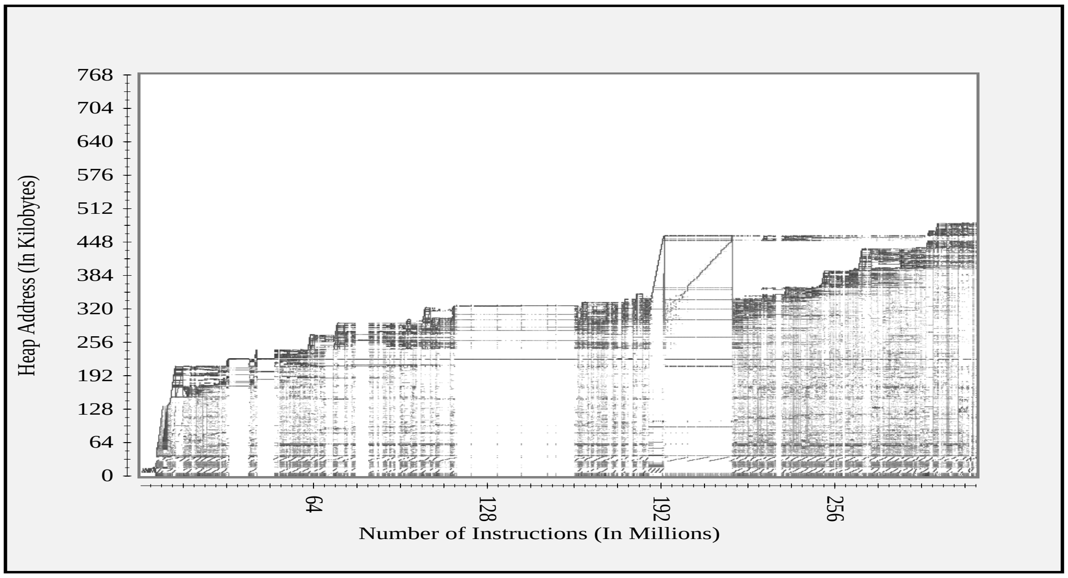

Figure E.42: Memory access plot for P2C using Lea's 2.6.1 allocator

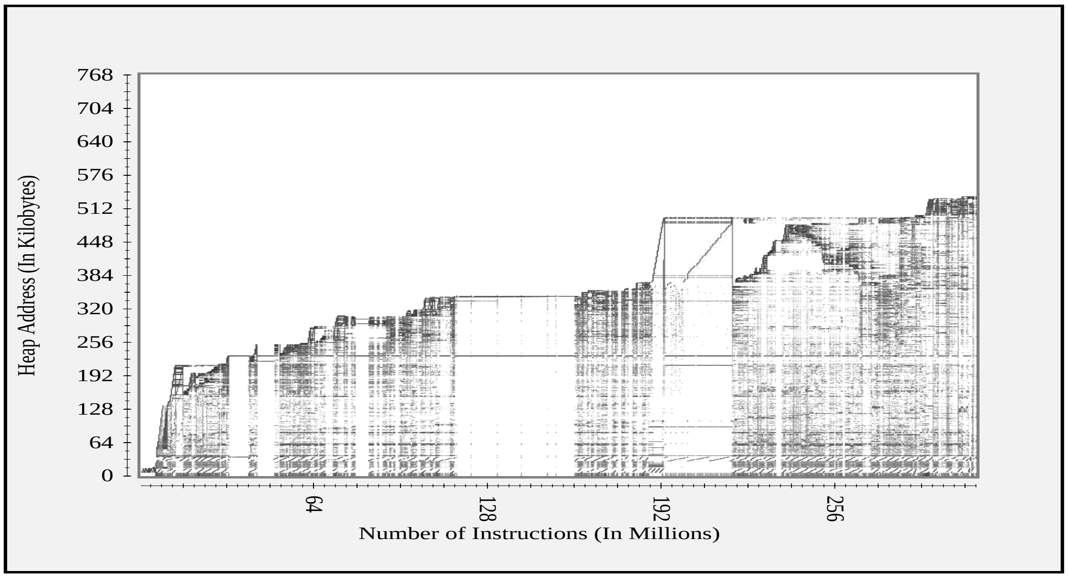

Figure E.43: Memory access plot for P2C using the next-fit LIFO no-footer allocator

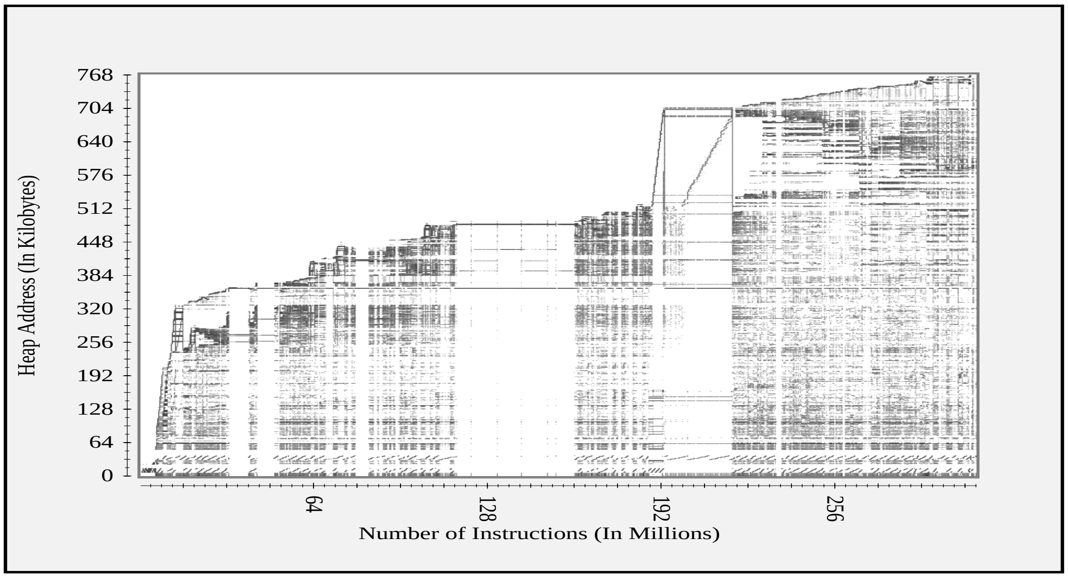

Figure E.44: Memory access plot for P2C using the simple segregated storage 2^N allocator

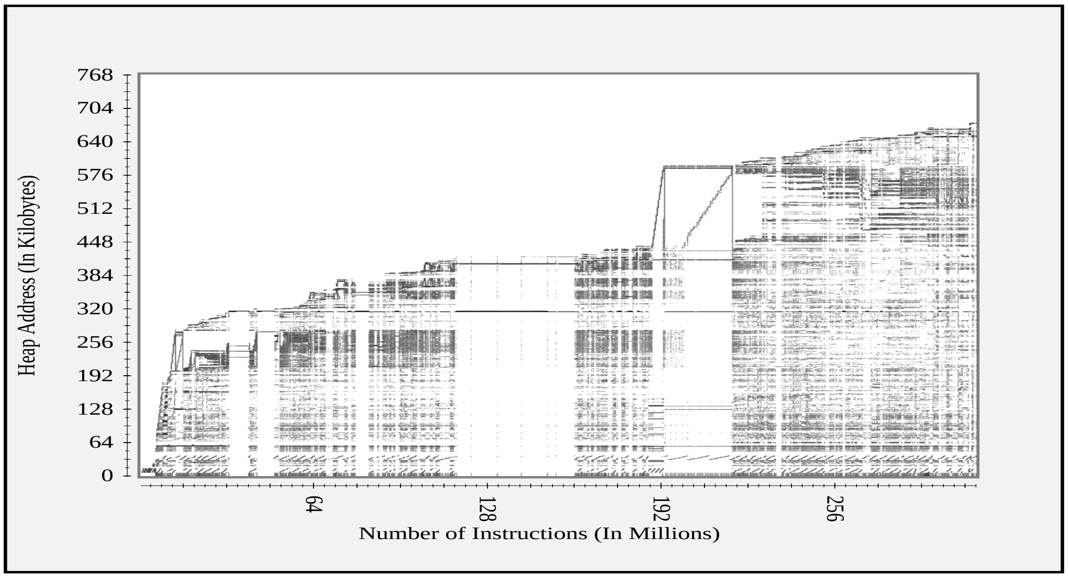

Figure E.45: Memory access plot for P2C using the simple segregated storage 2^N & $3 * 2^N$ allocator

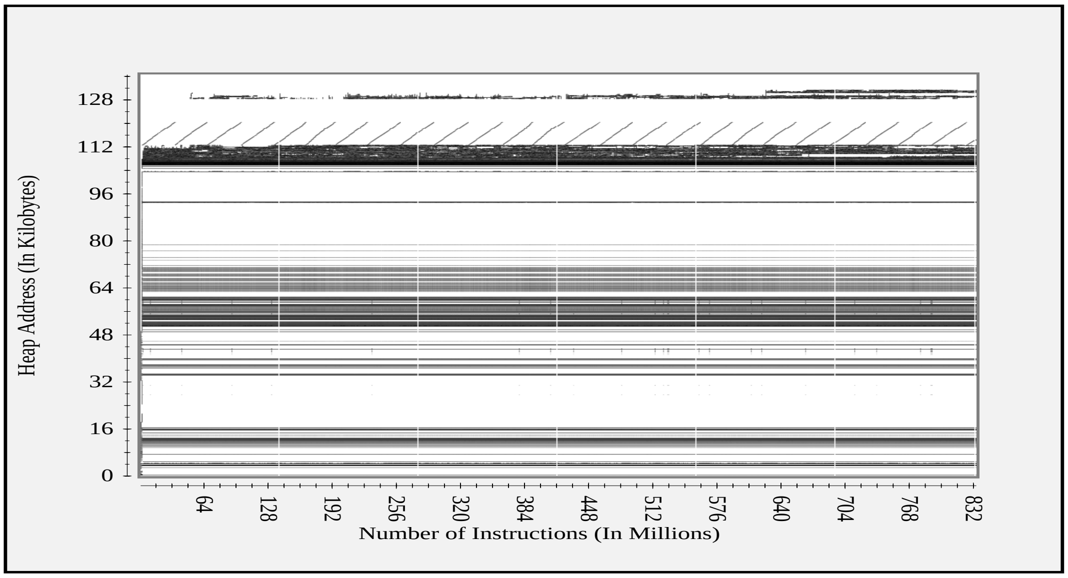

Figure E.46: Memory access plot for Perl using the binary-buddy allocator

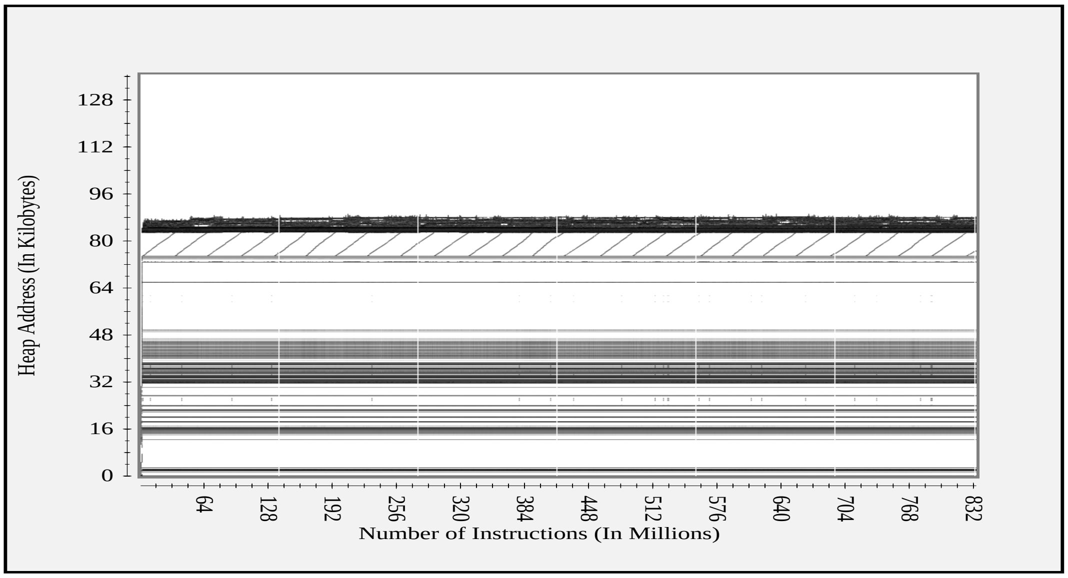

Figure E.47: Memory access plot for Perl using the best-fit LIFO no-footer allocator

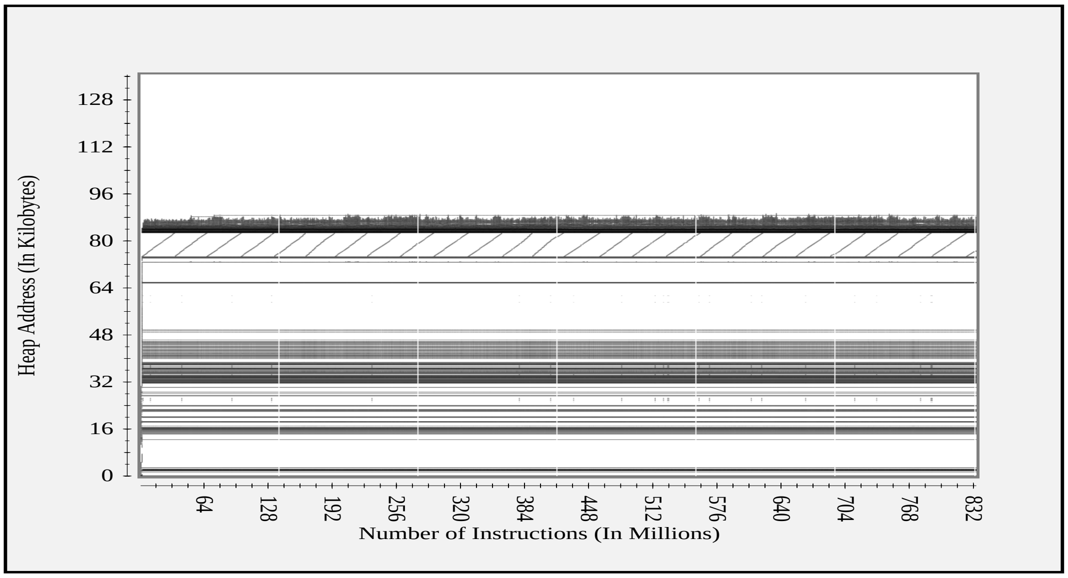

Figure E.48: Memory access plot for Perl using the first-fit address-ordered no-footer allocator

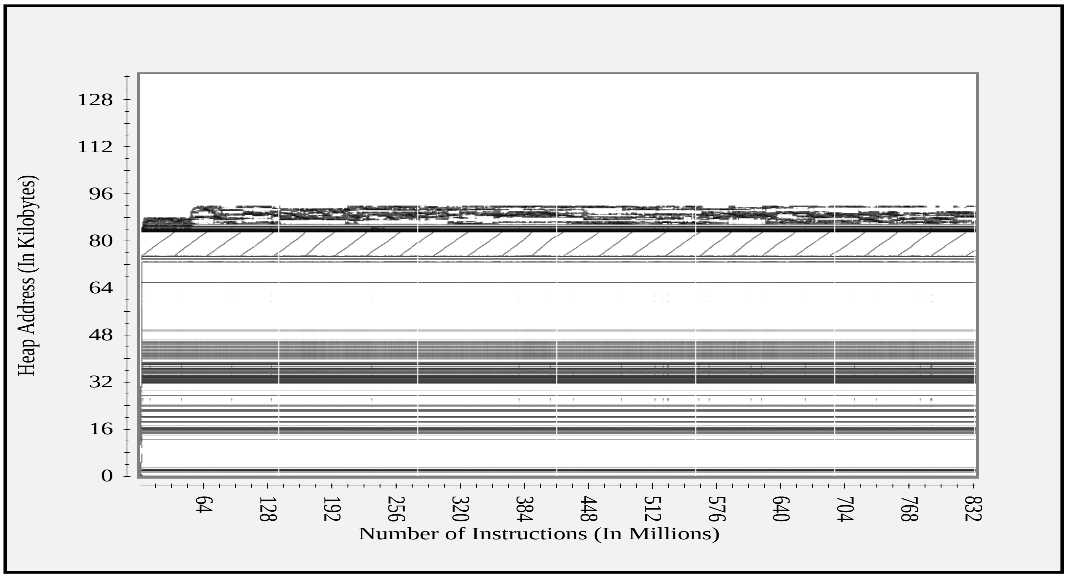

Figure E.49: Memory access plot for Perl using the first-fit LIFO no-footer allocator

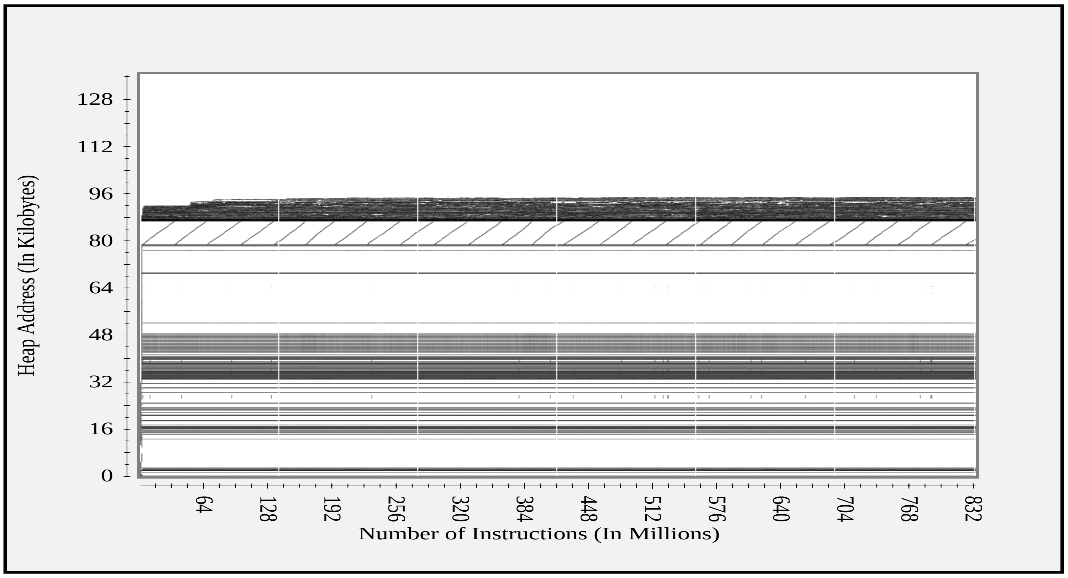

Figure E.50: Memory access plot for Perl using the half-fit allocator

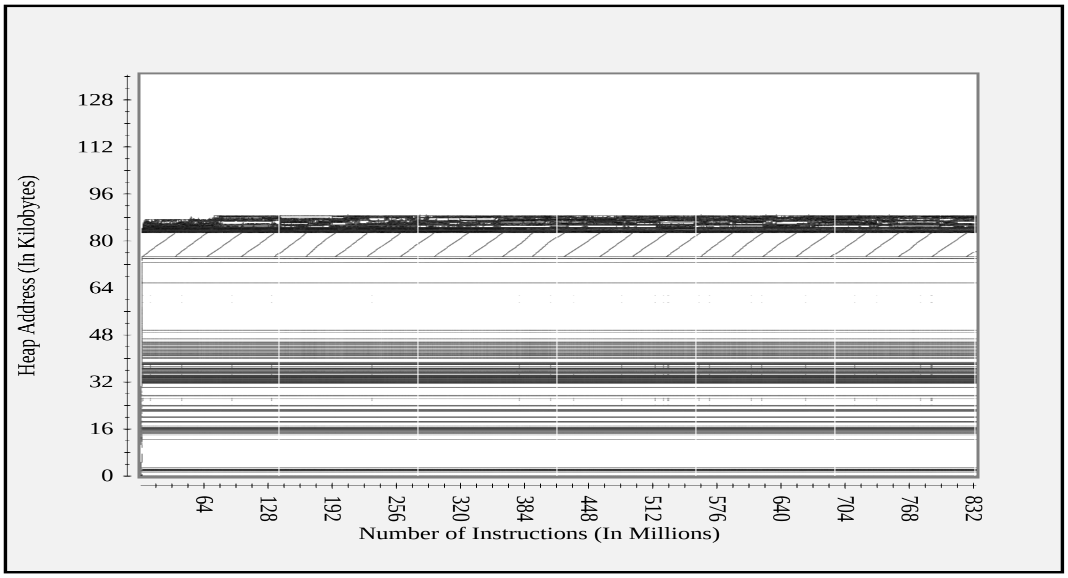

Figure E.51: Memory access plot for Perl using Lea's 2.6.1 allocator

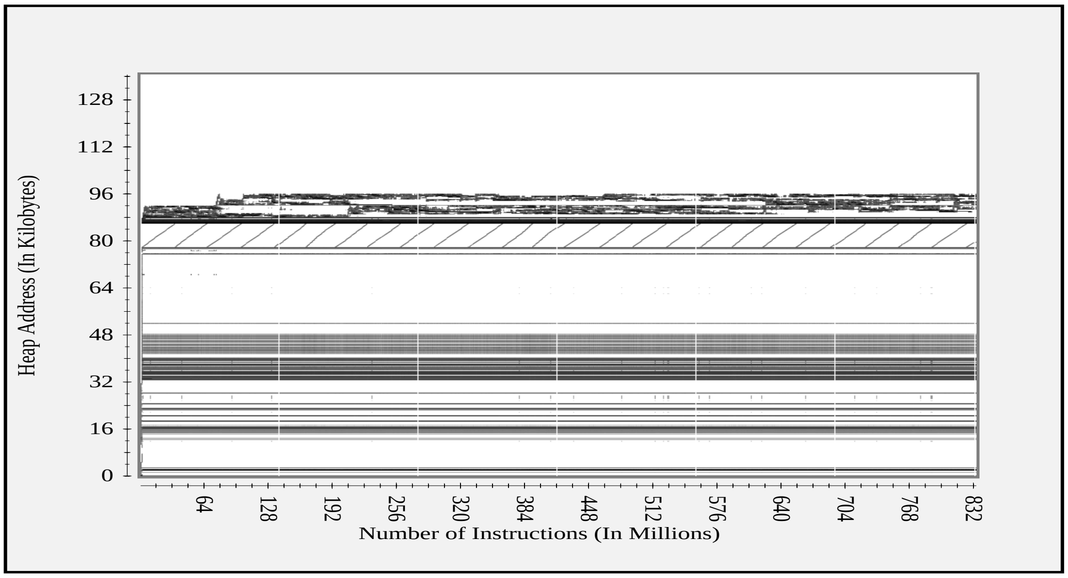

Figure E.52: Memory access plot for Perl using the next-fit LIFO no-footer allocator

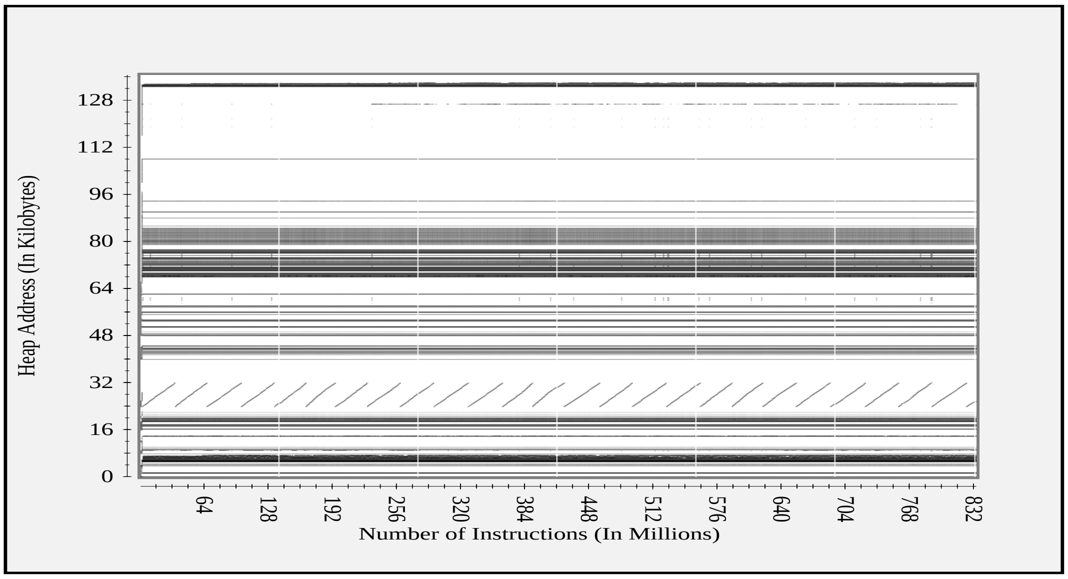

Figure E.53: Memory access plot for Perl using the simple segregated storage 2^N allocator

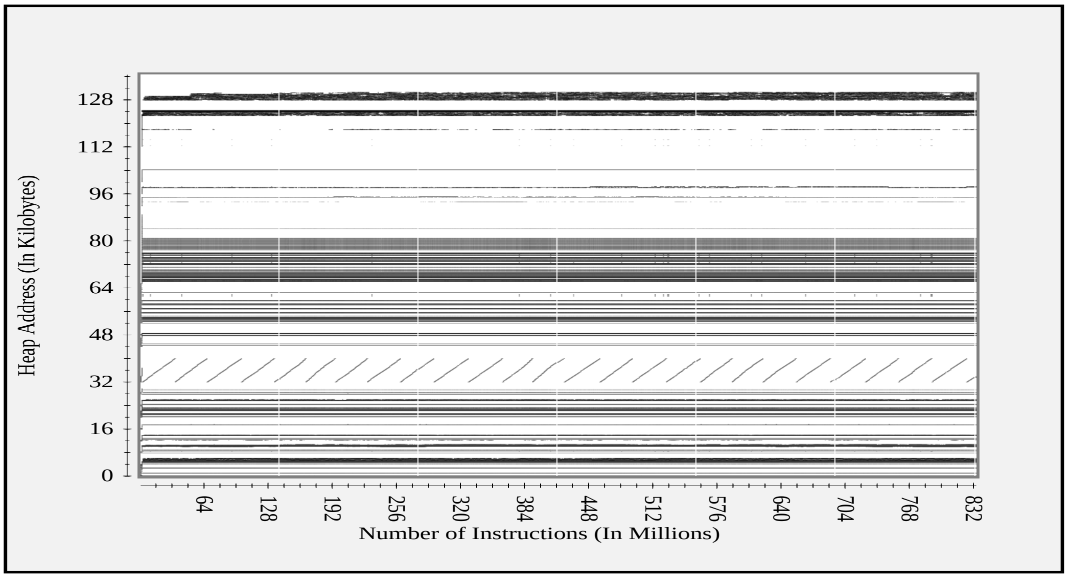

Figure E.54: Memory access plot for Perl using the simple segregated storage 2^N & $3 * 2^N$ allocator

Bibliography

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, Georgia, June 1988. ACM Press. Cited on page 137.

[AP87] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In E. Chiricozzi and A. D'Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–246, L'Aquila, Italy, September 1987. Elsevier North-Holland. Cited on page 146.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978. Cited on page 136.

[Bak91] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper. Also appears as *SIGPLAN Notices 27*(3):66–70, March 1992. Cited on pages 6, 8, 10, 147, and 150.

[BBB+88] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler. Cosmos: A compiled simulator for MOS circuits. In *25 Years of Electronic Design Automation*, pages 496–503, New York, New York, 1988. ACM Press. Cited on page 33.

[BC92] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag. Cited on pages 319 and 321.

[BDS91] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation* [PLD91], pages 157–164. Cited on pages 137 and 147.

[Bea97] Derek L. Beatty, 1997. personal communication. Cited on page 33.

[Bec82] Leland L. Beck. A dynamic storage allocation technique based on memory residence time. *Communications of the ACM*, 25(10):714–724, October 1982. Cited on page 15.

[Ber88] Steven H. Bergstein. Best-case caching in a symbolic multiprocessor. Bachelor's thesis, Massachusetts Institute of Technology EECS Department, Cambridge, Massachusetts, February 1988. Cited on page 4.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [LFP84], pages 108–113. Cited on pages 137 and 138.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970. Cited on page 6.

[CK93] Robert Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, Dept. of Computer Science and Engineering, University of Washington, Seattle, Washington, 1993. Cited on pages 39 and 106.

[Col61] G. O. Collins. Experience in automatic storage allocation. *Communications of the ACM*, 4(10):436–440, October 1961. Cited on pages 12 and 86.

[DDZ93] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, August 1993. Cited on pages 4 and 35.

[Den70] Peter J. Denning. Virtual memory. *Computing Surveys*, 3(2):153–189, September 1970. Cited on page 101.

[Det90a] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, May 1990. Cited on page 137.

[DeT90b] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, Palo Alto, California, August 1990. Cited on page 8.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978. Cited on pages 7, 8, 141, 144, and 147.

[Ede92] Daniel Ross Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference*, pages 1–19, Portland, Oregon, August 1992. USENIX Association. Technical Report UCSC-CRL-92-27, University of California at Santa Cruz, Baskin Center for Computer Engineering and Information Sciences, June 1992. Cited on page 161.

[EV91] Steven Engelstad and Jim Vandendorp. Automatic storage management for systems with real time constraints. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper. Cited on page 136.

[Fre84] John E. Freund. *Modern Elementary Statistics.* Prentice Hall, Englewood Cliffs, New Jersey, 1984. Cited on page 47.

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3), March 1986. Cited on page 42.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969. Cited on page 6.

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, Albuquerque, New Mexico, June 1993. ACM Press. Cited on pages 4, 5, 10, 96, and 102.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1), January 1990. Cited on page 37.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In Andreas Paepcke, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press. Cited on page 8.

[Hil87] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California at Berkeley, Berkeley, California, November 1987. Cited on page 106.

[JLS92] Phillip J. Koopman Jr., Peter Lee, and Daniel P. Siewiorek. Cache performance of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992. Cited on page 4.

[Joh92] Ralph E. Johnson. Reducing the latency of a real-time garbage collector. *ACM Letters on Programming Languages and Systems*, 1(1):46–58, March 1992. Cited on page 137.

[Kak97] Sheetal V. Kakkad. *Address Translation and Storage Management for Persistent Object Stores.* PhD thesis, The University of Texas at Austin, Austin, Texas, December 1997. Cited on page 161.

[Kin] Chris Kingsley. Implementation of malloc from bsd 4.2 unix. `http://www.plaza.hitachi-sk.co.jp/ftp/FreeBSD/FreeBSD-stable/src/lib/libc/stdlib/malloc.c`. Cited on page 18.

[Kno65] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, October 1965. Cited on pages 24 and 25.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973. First edition published in 1968. Cited on pages 15, 21, 22, 23, 45, and 93.

[Kuh70] Thomas S. Kuhn. *The Structure of Scientific Revolutions (Second Edition, Enlarged)*. University of Chicago Press, Chicago, Illinois, 1970. Cited on page 14.

[KV85] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *Proc. USENIX Summer 1985*, pages 489–506, Portland, Oregon, June 1985. USENIX Association. Cited on pages 27 and 94.

[Lea] Doug Lea. Implementations of malloc. See also the short paper on the implementation of this allocator. Available at `www://g.oswego.edu`. Cited on page 95.

[LFP84] *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984. ACM Press. Cited on pages 317 and 319.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983. Cited on pages 8, 9, and 162.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960. Cited on page 6.

[MH95] B. Magnusson and R. Henriksson. Garbage collection for control systems. In Henry Baker, editor, *International Workshop on Memory Management*, Lecture Notes in Computer Science, Scotland, September 1995. Springer-Verlag. Cited on page 138.

[Moo84] David Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [LFP84], pages 235–246. Cited on pages 9 and 162.

[Nil88] Kelvin Nilsen. Garbage collection of strings and linked data structures in real time. *Software Practice and Experience*, 18(7):613–640, July 1988. Cited on page 136.

[NOPH92] Scott Nettles, James O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [BC92], pages 357–364. Cited on pages 137 and 140.

[NS90] Kelvin Nilsen and William J. Schmidt. A high-level overview of hardware assisted real-time garbage collection. Technical Report TR 90-18a, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1990. Cited on page 137.

[Nut87] Peter R. Nuth. Communication patterns in a symbolic multiprocessor. Technical Report MIT/LCS/TR-395, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, June 1987. Cited on page 4.

[OOP91] *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. available at `ftp://ftp.cs.utexas.edu/pub/garbage/GC91`. Cited on pages 316, 317, and 322.

[PH86] Ivor P. Page and Jeff Hagins. Improving the performance of buddy systems. *IEEE Transactions on Computers*, C-35(5):441–447, May 1986. Cited on page 25.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996. 2nd Edition. Cited on pages 4, 42, 98, 99, and 100.

[PLD91] *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992. Cited on page 316.

[PN77] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977. Cited on page 24.

[PS89] C.-J. Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Dept. University of Wisconsin, Madison, Wisconsin, July 1989. Cited on page 4.

[PSC71] P. W. Purdom, S. M. Stigler, and Tat-Ong Cheam. Statistical investigation of three storage allocation algorithms. *BIT*, 11:187–195, 1971. Cited on page 29.

[RK68] Brian Randell and C. J. Kuehner. Dynamic storage allocation systems. *Communications of the ACM*, 12(7):297–306, May 1968. Cited on page 2.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Palo Alto, California, February 1988. Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory. Cited on page 8.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In Antonio Albano and Ron Morrison, editors, *Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer-Verlag. Cited on page 28.

[Smi82] Alan J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982. Cited on page 38.

[Ss92] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1992. Cited on pages 135 and 161.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3), 1985. Cited on page 102.

[Sta80] Thomas Standish. *Data Structure Techniques.* Addison-Wesley, Reading, Massachusetts, 1980. Cited on pages 18, 22, and 95.

[Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975. Cited on page 147.

[Ste83] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 30–32, Bretton Woods, New Hampshire, October 1983. ACM Press. Published as *Operating Systems Review 17*(5), October 1983. Cited on pages 18, 22, 27, and 95.

[Tad78] M. Tadman. Fast-fit: A new hierarchical dynamic storage allocation technique. Master's thesis, UC Irvine, Computer Science Dept., 1978. Cited on page 18.

[Tec97] Solid State Technology. W.s.t.s.: Chip sales back on track, good 4th quarter will usher in 1997 growth, 1997. available at `http://www.solid-state.com/sst/Newsitems/9612/news01.html`. Cited on page 11.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *Computing Surveys*, 29(2):128–170, June 1997. Cited on page 38.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM Press, April 1984. Published as *ACM SIGPLAN Notices 19*(5), May, 1987. Cited on pages 8, 9, and 162.

[Vo95] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 1995. To appear. Cited on page 4.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 29(4):229–239, April 1980. Cited on page 18.

[Wan89] Thomas Wang. MM garbage collector for C++. Master's thesis, California Polytechnic State University, San Luis Obispo, California, October 1989. Cited on pages 6, 10, and 147.

[Wil] Paul R. Wilson. Garbage collection. Expanded version of [Wil92]. In revision for ACM Computing Surveys. Draft available at `http://www.cs.utexas.edu/users/oops`. Cited on page 136.

[Wil88] Paul R. Wilson. Opportunistic garbage collection. *SIGPLAN Notices*, 23(12):98–102, December 1988. Cited on page 164.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [BC92], pages 1–42. Cited on pages 9, 146, 162, and 321.

[Wis78] David S. Wise. The double buddy-system. Technical Report 79, Computer Science Department, Indiana University, Bloomington, Indiana, December 1978. Cited on page 25.

[Wit91] P. T. Withington. How real is "real time" garbage collection? In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper. Cited on page 136.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, December 1993. Expanded version of workshop position paper submitted for publication. Cited on page 162.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS. Cited on pages 19, 86, and 95.

[WLM90] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching consideration for generational garbage collection: A case for large and set-associative caches. Technical Report UIC-EECS-90-5, University of Illinois at Chicago EECS Dept., Chicago, Illinois, December 1990. Cited on page 4.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on LISP and Functional Programming*, pages 32–42, San Francisco, California, June 1992. ACM Press. Cited on page 4.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings*, pages 23–35, New Orleans, Louisiana, 1989. ACM Press. Cited on page 9.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 1990. Cited on pages 144 and 146.

[ZG92] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. Technical Report CU-CS-604-92, University of Colorado at Boulder, Dept. of Computer Science, July 1992. Cited on pages 4 and 35.

[ZG94] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):107–131, 1994. Cited on pages 19 and 28.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conference Record of the 1990 ACM Symposium on LISP and Functional Programming*, pages 87–98, Nice, France, June 1990. ACM Press. Cited on page 8.

[Zor91] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, May 1991. Cited on page 4.

[Zor93] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, July 1993. Cited on pages 37 and 38.

Vita

Mark Stuart Johnstone was born in San Diego, California on April 13, 1965, the son of Hubert Crosby Johnstone and Margery Schurr Johnstone. After completing his work at Clairemont High School, San Diego, California, in 1983, he attended the University of California at Davis. He received the degree of Bachelor of Science with Honors from the University of California at Davis in June 1987. During the following three years, he was employed as a Scientific Programmer for the Lockheed Missiles and Space Company in the Space Systems Division, Sunnyvale, California. In August 1990 he entered the Graduate School of the University of Texas. He received the degree of Master of Science in Computer Science in May 1992. He has been employed by the Motorola Corporation since June 1996.

Permanent Address: 3721 Mt. Sandy Dr.
San Diego, California 92117

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

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