The Memory Fragmentation Problem: Solved?[*]

Mark S. Johnstone
The Motorola Somerset Design Center
Austin, Texas, USA
johnstone@acm.org

Paul R. Wilson
The University of Texas at Austin
Austin, Texas, USA
wilson@cs.utexas.edu

Abstract

We show that for 8 real and varied C and C++ programs, several conventional dynamic storage allocators provide near-zero fragmentation, once we account for overheads due to implementation details such as headers, alignment, etc. This substantially strengthens our previous results showing that the memory fragmentation problem has generally been misunderstood, and that good allocator policies can provide good memory usage for most programs. The new results indicate that for most programs, excellent allocator policies are readily available, and efficiency of implementation is the major challenge. While we believe that our experimental results are state-of-the-art and our methodology is superior to most previous work, more work should be done to identify and study unusual problematic program behaviors not represented in our sample.

1 Introduction

Memory allocation has been an active area of research. A large number of algorithms have been proposed which were at least partly motivated by the belief that fragmentation can be a severe problem for some programs. Other algorithms have been proposed with the emphasis on performance rather than fragmentation. In this paper, we show that some well-known memory allocation algorithms, which can be implemented very efficiently, have essentially zero fragmentation for a wide variety of programs.

The extremely low fragmentation of these algorithms has gone unnoticed largely because 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 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). As we showed in [WJNB95, Joh97], 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 memory allocation *policies* using memory allocation traces from real programs. Using a large set of tools we built, we measured how well these allocation algorithms performed on a set of eight real traces. Our results confirm, and in some cases exceed, the speculations we made in our survey on memory allocation research [WJNB95]. In particular, we stated that:

> In simulations of two of the best allocators (address-ordered first fit and best fit), eliminating all header overhead reduced their memory waste to about 14%. We suspect that using one-word alignment and a smaller minimum object size could reduce this by several percent more. This suggests the "real" fragmentation produced by these policies—as opposed to waste caused by the implementation mechanisms we used—may be less than 10%.

In this paper, we show that almost none of the wasted memory is due to true fragmentation.

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. Unfortunately, many good policies are discounted because the obvious implementation is inefficient. All of the measurements presented in this paper are for the memory allocation *policy* under consideration, independent of any particular *implementation* of that policy.

[*] This research was sponsored by the National Science Foundation under grant CCR-9410026

2 Description of Allocators

In this section, we will describe the memory allocation policies we studied.[1] These policies fall into three basic categories:

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

2.1 Sequential fit algorithms

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 also be implemented using a tree of lists of same sized objects [Sta80], and address-ordered first fit can be implemented using a Cartesian tree [Ste83]. For concreteness and simplicity, we describe the sequential-fit algorithms' well-known implementations, but we stress that the same policies can be implemented more efficiently.

2.1.1 First Fit

A first-fit allocator 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 first fit is that the larger blocks near the beginning of the list tend to be split first, and the remaining fragments result in a lot of small free blocks near the beginning of the list.

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

2.1.2 Next Fit

A common "optimization" of first fit is to use a *roving pointer* for allocation [Knu73]. This pointer records the point 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 do not result in an accumulation of small unusable blocks in one part of the list. As we will show in Section 6, this "optimization" generally increases fragmentation.

2.1.3 Best Fit

A best-fit sequential-fit allocator searches the free list to find the smallest free block large enough to satisfy a request. The basic strategy here is to minimize the amount of wasted space by ensuring that fragments are as small as possible. 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 6, 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 2.2.2) can be a very good approximation of best fit and are easy to implement efficiently.

In this paper, we present results for first fit, next fit, and best fit, each with LIFO, FIFO, and address ordered (AO) free lists. Memory is requested from the operating system in 4K blocks, and all free blocks are immediately coalesced with their neighbors when possible. Two versions: first fit AO 8K and best fit AO 8K, request memory from the operating system in 8K blocks.

2.2 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* scheme.

One common variation is to use *size classes* to group similar object sizes together onto a single free list. Free blocks from this list are used to satisfy any request for an object whose size falls within this 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.

2.2.1 Simple Segregated Storage

In this variant, larger free blocks are not split to satisfy requests for smaller sizes, and smaller free blocks are not coalesced 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 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*.

In this paper, we present results for two variations on simple segregated storage. The first, which we call simple seg 2^N, uses size classes which are powers of two (e.g., 16, 32, 64, etc., bytes), and requests memory from the operating system in 4K blocks. The second, which we call simple seg 2^N & $3 * 2^N$, uses twice as many size classes as simple seg 2^N in an attempt to reduce internal fragmentation at the possible cost of increased external fragmentation. Simple seg 2^N & $3 * 2^N$ uses size classes which are powers of two and three times powers of two (e.g., 16, 24, 32, 48, 64, etc., bytes), and requests memory from the operating system in 4K and 6K blocks depending on which size-class is empty. Neither allocator does any coalescing.

2.2.2 Segregated Fit Algorithms

Another variation on the segregated free list 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, with each list holding free blocks of any size between this 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. 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.

In this paper, we present results for Doug Lea's memory allocator version 2.6.1[2][Lea]. This allocator is a segregated fit algorithm with 128 size classes. Size classes for sizes less than 512 bytes each hold exactly one size, spaced 8 bytes apart. Searches for available chunks are processed in best-fit order. All freed chunks are immediately coalesced.

2.3 Buddy systems

Buddy systems [Kno65, PN77] are a variant of segregated free lists, supporting a limited but efficient kind of splitting and coalescing. In 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.

[2] At the time of this writing Doug Lea's allocator is at version 2.6.4.

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 of its subparts allocated but not others.) 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.

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

2.3.1 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 computations can be implemented efficiently with bitwise logical operations.

A major problem with binary buddies is that internal fragmentation is usually relatively high: about 25%. This fragmentation is caused by the requirement that any object size 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). The memory allocator used in our study was originally implemented for the COSMOS circuit simulator [BBB$^+$88, Bea97].

2.3.2 Double Buddy

Double buddy [Wis78, PH86] systems use a different technique to allow a closer spacing of size classes. They use two different buddy systems, with staggered sizes. For example, one buddy system may use powers-of-two sizes (i.e., 2, 4, 8, 16, ...) while the other uses a powers-of-two spacing starting at a different size, such as 3, resulting in sizes 3, 6, 12, 24, etc. 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.[3]

3 The Test Programs

For our test programs, we used eight varied C and C++ programs that run under UNIX (SunOS 5.5). These programs

[3] 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].

allocate between 1.3 and 104 megabytes of memory during a run, and have a maximum of between 69 KB and 2.3 MB live data at some point during execution. On average they allocate 27 MB total data and have a maximum of 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 [ZG92, DDZ93]. We use these three to attempt to provide some points of comparison, while also using five new and different memory-intensive programs.

3.1 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. In addition, 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 these 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 an allocator's scalability to medium and large-sized heaps.[4]

We tried to obtain a 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 the reasons which follow. We attempted to avoid over-representation of particular program types, i.e., too many programs that do the same thing. In particular, we avoided having several scripting language interpreters—such programs are generally portable, widely available and widely used, but 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., fail to discard objects at the proper point, and lead 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 similar to 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. They choose to have poorer resource usage because attempts at plugging the leaks often result in worse bugs, such as 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[5]. (Two were the PTC and YACR programs from Zorn, *et al.*'s experiments [ZG92, DDZ93].) 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.[6]

3.2 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 1 gives some basic statistics for each of our eight test programs. The *Kbytes alloc'd* column gives the total allocation over a whole run, in kilobytes. 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[7]. Note that the maximum live objects and maximum live bytes might not occur at the same point in a trace, if, for example, the average size of objects varies over time. The *avg lifetime* column gives the average[8] object lifetime in bytes. This 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).

Table 2 is a histogram of the object sizes allocated by our eight test programs. To create this histogram, object sizes were rounded up to the next larger power of two. Thus, an entry in Table 2 in the column "4K" is the number of objects allocated by that program that were greater than 2K bytes, but less than or equal to 4K bytes in size.

Descriptions of the programs follow:

[4] Two programs used by Zorn and Grunwald [ZG92] and by Detlefs, Dosser, and Zorn [DDZ93] 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 for boundary effects to obscure scalability issues.

[5] Other programs were excluded because they either 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.

[6] LRUsim actually must retain almost all allocated objects for the duration of a run, because it cannot tell—even in principle—which ones will be needed again.

[7] This is the maximum 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.

[8] Following [FW86, DAP96], all averages presented in this paper are simple arithmetic means.

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 1: Basic Statistics for the Eight Test Programs

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
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 2: Maximum number of live objects per size class

- *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*[9] allocations to obtain this trace, and built a postprocessor to translate the use of obstack calls into equivalent `malloc()` and `free()` calls.[10] The input data for the compilation was the the largest source file of the compiler itself (`combine.c`).[11]

- *Ghostscript* is a widely-used portable interpreter for PostScript, written by Peter Deutsch and modified by Benjamin Zorn to remove hand-optimized memory allocation [Zor93]. The input used 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.[12]

- *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. Abstractly, this is roughly similar to a Fourier analysis, decomposing a function into a combination of other, simpler functions. Unlike 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.

- *Hyper* is a hypercube network communication simulator written by Don Lindsay. It builds a representation of a hypercube network, then simulates random messaging, while accumulating statistics about messaging

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

[10] 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.

[11] 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 and 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).

[12] 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.

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 reach its destination, 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 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.[13]

- *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) they used in their studies. This one allocates and deallocates more memory, at least for the input we used.

- *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 words in a dictionary into filled paragraphs. Hand-optimized memory allocation was removed by Zorn [Zor93].

4 Experimental Design

A goal of this research was 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 to perform the basic malloc functions and, as a side-effect, create a trace of the memory allocation activity of the program. This trace is composed of a series of records, each containing:

- the type of operation performed (malloc, realloc, free);

- the memory location of this record (for malloc, this was the memory location returned; for realloc and free, this was the memory location passed by the application); and

- the number of bytes requested (for free, this was 0).

Each of our test programs was linked with this malloc trace gathering library, and a trace for each program was generated.

The second step was to build a trace processor to read a trace and produce the following 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 another trace processor to read a trace and call malloc, realloc, and free of the implementation of the allocation policy under study. We modified each memory allocator implementation 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, as described in Section 5 for a particular program using that implementation of a memory allocation policy.

4.1 Removing implementation overheads

In order to study fragmentation costs due to policy, it was necessary to account for and remove implementation overheads. First, we removed header and footer overheads by requesting fewer bytes than recorded in the trace being simulated. Thus, if a request was for 24 bytes, and the particular implementation of malloc being studied used 4 bytes for header information, the simulation only requested 20 bytes of memory.[14] We also needed to remove minimum object size and hardware required alignment overheads. For all allocator implementations that we studied, the minimum object size was 16 bytes, and the hardware required that memory be aligned on 8 byte boundaries. We removed both of these overheads by multiplying all size requests by 16 and dividing the final memory use number by 16. The resulting memory use number is the number of bytes needed by that policy for the trace being simulated.

5 Our Measure of Fragmentation

In this paper, 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.)

There are a number of legitimate ways to measure fragmentation. Figure 1 illustrates four of these. Figure 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 live memory requested by GCC (in kilobytes). 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:

[13] 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 nondecreasing, 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.)

[14] We were able to do this because we were only simulating allocation and deallocation using a trace of the actual activity. The memory returned by malloc was unused.

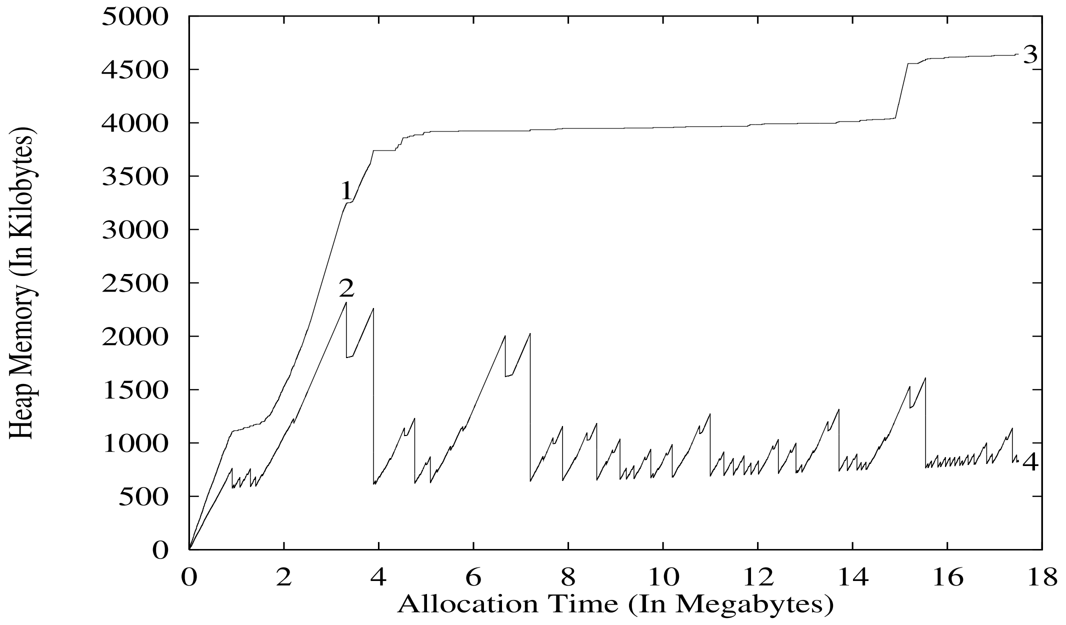

Figure 1: Measurements of Fragmentation for GCC using simple seg 2^N

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 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 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 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. In our experimental results (Section 6) we will see several examples of this effect.

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 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 measured fragmentation using both methods 3 and 4 (Section 6). 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, fragmentation will not be a problem at all. Allocation policy is still important for these programs, however, because allocator placement choices can have a dramatic effect on locality. In [Joh97], we presented results for a number of locality experiments using the programs and allocators presented in this paper. We found that there is little correlation between fragmentation and locality. However, we also found that the best placement policies in terms of fragmentation are also the best placement policies in terms of locality.

5.1 Experimental Error

In this research, we worked very hard to remove as much measurement error as possible. However, some error still remains, which we will describe next.

The remaining experimental error comes from the way our allocators request memory from the operating system (using the `sbrk` UNIX system call). Most of the implementations of our allocators request their memory from the operating system in 4K byte blocks. Because all calls to `sbrk` in these experiments returned memory contiguous with previously allocated memory, any measurement of the heap size of a program using a particular allocator can be an over-estimate by at most 4K bytes. Recall, however, that to remove alignment overheads, we multiplied all object sizes by 16, and divided the resulting heap size by 16. This also has the effect of decreasing our experimental error due to `sbrk` by a factor of 16. Thus, our measurements for most allocators can be an overestimate by at most 256 bytes. This error is slightly higher for three allocators.

The double buddy allocator requests memory from the operating system in two different sizes: 4K and 6K, yielding an average size of 5K. Thus, for this allocator, our experiments can overestimate the memory used by as much as 640 bytes.

Neither of the simple segregated storage allocators (simple seg 2^N and simple seg 2^N & $3*2^N$) perform any coalescing. Thus, *each size class* can contribute to an over-estimate of up to 256 bytes. For example, if a particular program allocates objects in 7 size classes, then the amount of memory used can be over-estimated by up to 1792 bytes.

6 Results

In this section, we present the results of our experiments, as shown in Tables 3 and 4. We begin by analyzing the allocation policies in our study. We then present a *strategy* for allocating memory, which is followed by the two best allocation policies. Finally, we present an analysis of two important properties of our test programs and how they reinforce this strategy.

6.1 Analysis of the Experiments

Tables 3 and 4 show the percentage actual fragmentation for each of the selected allocators, for each trace, using fragmentation methods #3 and #4. 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, first fit AO, and Lea's 2.6.1 allocators all perform quite well, using fragmentation measure #4, on all of the test programs. Perl is the only program for which they have any real fragmentation (10%), and because that program has only 70K bytes maximum live data, this fragmentation represents just 6.9K of wasted memory. The fact that first fit AO performs *very* well on Espresso, using just 0.26% more memory than the theoretical minimum, but shows over 25% fragmentation using measure #3 is evidence of the misleading behavior of fragmentation measure #3.

From Tables 3 and 4 we can see that the two best allocation policies, first-fit addressed-ordered free list with 8K `sbrk`, and best-fit addressed-ordered free list with 8K `sbrk`, both average less than 1% actual fragmentation, using both measures of fragmentation. Conversely, the worst of our allocators (those that tried to trade increased internal fragmentation for reduced external fragmentation, and did not coalesce all possible blocks) had over 50% actual fragmentation, giving further evidence that this is not a good policy decision. The extremely high fragmentation numbers for the simple segregated storage allocators using fragmentation measurement #3 are largely due to their lack of coalescing and their performance with one program: LRUsim. Excluding LRUsim, simple seg 2^N averaged 174% fragmentation, and simple seg 2^N & $3*2^N$ averaged 164% fragmentation using measurement #3. The extremely high fragmentation for the simple segregated storage allocators on the LRUsim trace is further evidence that fragmentation measure #3 can produce misleading results.

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 implements an approximation of best fit fairly efficiently, it shows that a reasonable approximation of a best-fit policy is both desirable and achievable.

6.2 A *Strategy* That Works

In Sections 1 and 2, we discussed the importance of separating *policy* from *mechanism*. There is yet a third consideration that is important to separate: *strategy*. In Section 6.1, 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.[15] 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

[15] 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.

Allocator	Espresso	GCC	Ghost	Grobner	Hyper	Perl	P2C	LRUsim	Average	Std Dev
best fit AO 8K	0.30%	0.34%	3.25%	1.67%	0.03%	1.07%	1.17%	0.03%	0.98%	1.09%
best fit AO	1.64%	0.49%	3.65%	4.14%	0.20%	17.3%	2.09%	0.29%	3.73%	5.68%
best fit LIFO	1.65%	0.50%	3.65%	4.35%	0.20%	17.3%	2.11%	0.29%	3.76%	5.68%
best fit FIFO	1.63%	0.49%	5.40%	4.11%	0.20%	17.3%	2.05%	0.29%	3.93%	5.72%
first fit AO 8K	0.30%	0.36%	3.02%	1.11%	0.03%	1.06%	1.33%	0.03%	0.91%	0.99%
first fit AO	25.3%	0.51%	3.40%	3.78%	0.20%	17.3%	2.22%	0.29%	6.63%	9.41%
first fit LIFO	41.4%	92.9%	180%	68.6%	0.23%	17.3%	5.28%	0.29%	50.7%	62.3%
first fit FIFO	1.66%	0.53%	7.59%	9.74%	0.21%	17.3%	2.46%	0.29%	4.97%	6.13%
next fit AO	27.2%	0.84%	19.4%	36.2%	0.24%	17.3%	8.34%	0.29%	13.7%	13.6%
next fit LIFO	41.4%	88.6%	201%	57.2%	0.23%	17.3%	16.9%	0.29%	52.9%	67.0%
next fit FIFO	1.69%	0.88%	54.6%	81.3%	0.24%	17.3%	10.6%	0.29%	20.9%	30.5%
Lea 2.6.1	1.66%	0.48%	3.65%	4.52%	0.20%	17.3%	2.31%	0.29%	3.80%	5.68%
binary buddy	82.3%	34.2%	41.5%	37.0%	102%	48.3%	55.1%	77.6%	59.8%	24.7%
double buddy	77.7%	34.9%	22.0%	12.7%	51.2%	24.9%	32.7%	33.6%	36.2%	20.2%
simp seg 2^N	480%	451%	41.0%	57.5%	26.0%	76.4%	87.1%	13328%	1818%	4654%
simp seg $3 * 2^N$	475%	452%	34.4%	41.6%	26.0%	56.1%	64.9%	10595%	1468%	3693%
Average	78.8%	72.4%	39.2%	26.6%	13.0%	23.8%	18.5%	1502%		
Std Dev	158%	151%	61.5%	27.5%	27.9%	19.8%	27.0%	4113%		

Table 3: Percentage actual fragmentation (using method #3) for each allocator for all traces

Allocator	Espresso	GCC	Ghost	Grobner	Hyper	Perl	P2C	LRUsim	Average	Std Dev
best fit AO 8K	0.26%	0.33%	3.00%	1.34%	0.01%	0.58%	1.02%	0.08%	0.83%	0.99%
best fit AO	0.26%	0.33%	3.40%	2.03%	0.16%	9.98%	1.78%	0.26%	2.28%	3.32%
best fit LIFO	0.26%	0.50%	3.40%	2.03%	0.16%	9.98%	1.78%	0.26%	2.30%	3.31%
best fit FIFO	0.26%	0.33%	3.04%	2.03%	0.16%	9.98%	1.78%	0.26%	2.23%	3.31%
first fit AO 8K	0.26%	0.35%	2.77%	1.00%	0.01%	0.58%	1.15%	0.08%	0.78%	0.90%
first fit AO	0.26%	0.50%	3.40%	2.03%	0.16%	9.98%	1.78%	0.26%	2.30%	3.31%
first fit LIFO	9.37%	23.7%	179%	62.7%	0.16%	9.98%	4.83%	0.26%	36.3%	61.2%
first fit FIFO	0.26%	0.50%	7.36%	4.79%	0.16%	9.98%	1.78%	0.26%	3.14%	3.81%
next fit AO	1.77%	0.67%	16.7%	26.9%	0.16%	9.98%	7.89%	0.26%	8.04%	9.64%
next fit LIFO	9.37%	21.0%	200%	51.7%	0.16%	9.98%	15.0%	0.26%	38.4%	67.3%
next fit FIFO	0.26%	0.85%	54.6%	71.0%	0.16%	9.98%	9.92%	0.26%	18.4%	28.1%
Lea 2.6.1	0.26%	0.33%	3.40%	2.03%	0.16%	9.98%	1.78%	0.26%	2.28%	3.32%
binary buddy	45.8%	34.1%	38.4%	36.9%	99.9%	39.0%	55.0%	77.7%	53.4%	23.6%
double buddy	68.6%	31.3%	20.8%	12.3%	50.0%	24.9%	32.4%	33.7%	34.3%	17.7%
simp seg 2^N	162%	99.5%	39.0%	57.4%	26.0%	51.2%	74.5%	79.0%	73.6%	42.7%
simp seg $3 * 2^N$	159%	99.9%	32.4%	41.0%	26.0%	33.9%	56.2%	43.2%	61.5%	45.7%
Average	28.6%	19.6%	38.2%	23.6%	12.7%	15.6%	16.8%	14.8%		
Std Dev	55.0%	33.5%	61.4%	25.8%	27.4%	14.1%	24.1%	28.0%		

Table 4: Percentage actual fragmentation (using method #4) for each allocator for all traces

Program	90%	99%	99.9%
GCC	1K	2,409K	17,807
Espresso	1K	8K	57K
Ghostscript	1K	40,091K	48,593K
Grobner	2K	3,311K	3,939K
Hyper	2K	12K	18K
P2C	11K	3,823K	4,494K
Perl	1K	11K	184K
LRUsim	1K	1K	1K
Average	2.5K	6,208K	9,387K

Table 5: Time before given % of free *objects* have both temporal neighbors free

Program	90%	99%	99.9%
GCC	223K	2,355K	17,805K
Espresso	1K	62K	9,552K
Ghostscript	14K	44,876K	48,752K
Grobner	2K	2,464K	3,836K
Hyper	1K	11K	16K
P2C	16K	4,142K	4,614K
Perl	1K	13K	7,153K
LRUsim	1K	1K	8K
Average	32K	6,740K	11,467K

Table 6: Time before given % of free *bytes* have both temporal neighbors free

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.

6.3 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 5 shows the number of bytes allocated 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 6 shows the same information as Table 5, except weighted by the *size* of the objects becoming free. Thus, the table shows how long (in allocation time) before 90%, 99%, and 99.9% of the *bytes* allocated can be coalesced with neighboring memory. Here, we see that if we wait for just 32K of allocation, 90% of all memory allocated can be coalesced with its neighboring memory.

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

Program	90%	99%	99.9%	100%
GCC	5	12	254	641
Espresso	9	95	308	758
Ghostscript	7	85	344	589
Grobner	12	55	100	139
Hyper	1	2	2	6
LRUsim	1	1	5	21
P2C	4	26	58	92
Perl	10	27	60	99
Average	6	38	141	293

Table 7: Number of object sizes representing given percent of all object sizes

6.4 Programs Tend to Allocate Only a Few Sizes

For most programs, the vast majority of objects allocated are of only a few sizes. Table 7 shows the number of object sizes represented by 90%, 99%, 99.9%, and 100% of all objects allocated. 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 7, 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.

7 Conclusions

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 address-ordered first fit can be implemented using a Cartesian tree [Ste83]. Most importantly, an excellent allocator implementation that runs on many platforms was written by Doug 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 is a problem of recognizing that good allocation *policies* already exist, and have inexpensive implementations. For most programs, the problem of simple overheads is more significant than the problem of fragmentation itself.

References

[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.

[Bea97] Derek L. Beatty, 1997. Personal communication.

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

[DAP96] John L. Hennessy David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996. 2nd Edition.

[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.

[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.

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

[Joh97] Mark S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, Austin, Texas, December 1997.

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

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1973. First edition published in 1968.

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

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

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

[Sta80] Thomas Standish. *Data Structure Techniques*. Addison-Wesley, Reading, Massachusetts, 1980.

[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.

[Wis78] David S. Wise. The double buddy-system. Technical Report 79, Computer Science Department, Indiana University, Bloomington, Indiana, December 1978.

[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.

[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, Boulder, Colorado, July 1992.

[Zor93] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.