The Memory Fragmentation Problem: Solved?*

Mark S. Johnstone and Paul R. Wilson
The University of Texas at Austin
Austin, Texas 78712-1188 USA
markj, wilson@cs.utexas.edu

October 18, 1997

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 trace). As we show 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 that we studied. These policies fall into four basic categories: *sequential fits*, *simple segregated storage*, *segregated fits*, and *buddy systems*.

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.

2.1.2 Next Fit

A common "optimization" of first fit is to use a *roving pointer* for allocation [Knu73]. This 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 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 to 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[1][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.

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

2.3 Buddy systems

Buddy systems [Kno65, PN77] are a variant of segregated 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.

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 sub-parts 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—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). 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.[2]

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 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

[2] 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].

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 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 an allocator's scalability to medium and large-sized heaps.[3]

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.[4] 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[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] 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.

[4] We did not use one very memory-intensive program 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, the implementors eventually chose to use the Boehm-Weiser conservative garbage collector.

[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.

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.

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

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 the average size of objects varies over time. The *avg lifetime* column gives the average 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).

Descriptions of the programs follow:

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

[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] 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].

[9] 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.

[10] 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).

- *Ghostscript* is a widely-used portable interpreter for PostScript (a page rendering language) written by Peter Deutsch and modified by Zorn 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.[11]

- *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.[12]

- *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 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[13] 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.[14]

- *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

[11] 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.

[12] 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.

[13] 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 descendants to the left of the node. This is 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.

[14] 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.)

- 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 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.[15] 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.)

[15] 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.

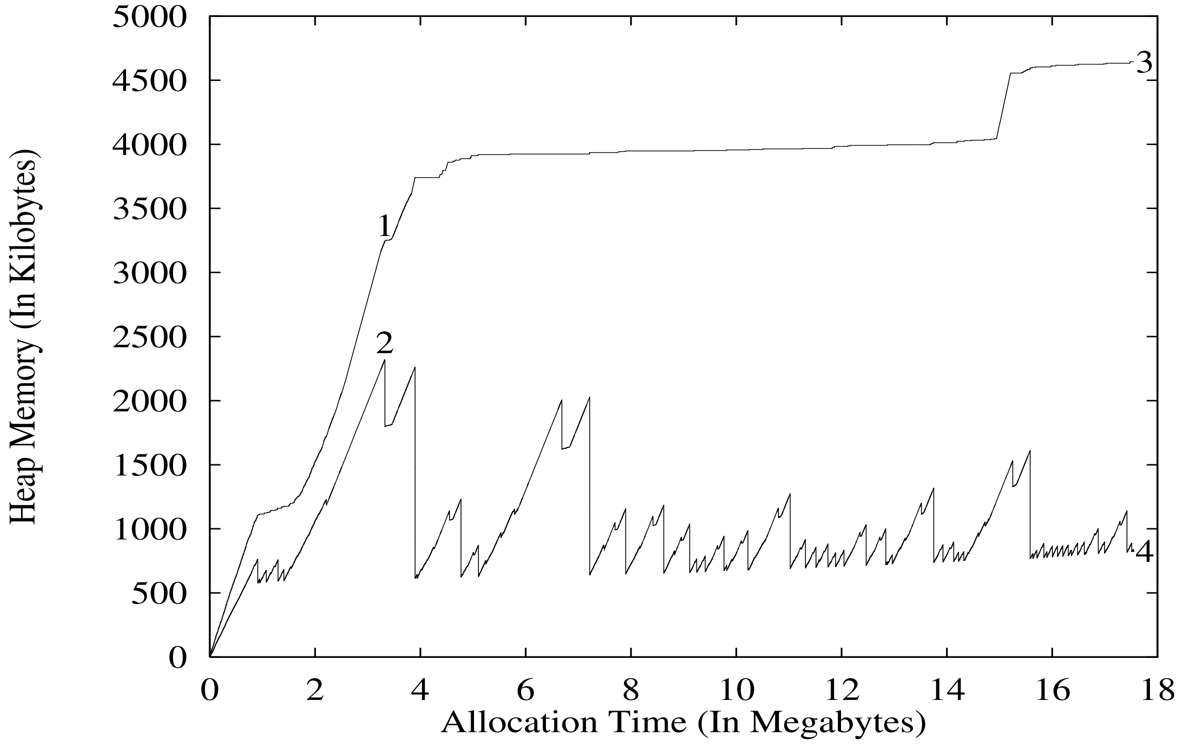

Figure 1: Measurements of Fragmentation for GCC using simple seg 2^N

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:

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.

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 because allocator placement choices can have a dramatic effect on locality [Joh97].

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 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 up to 4K bytes. This error is even larger for three of our allocators: double buddy, simple seg 2^N, and simple seg 2^N & $3 * 2^N$

The double buddy allocator requests memory from the operating system in two different sizes: 4K and 6K, yielding an average size of 5K. Thus, this allocator can yield an over-estimate of the amount of memory used by as much as 10K.

Neither of the simple segregated storage allocators (simple seg 2^N and simple seg 2^N & $3 * 2^N$) perform any coalescing. *Each size class* can contribute to an over-estimate of up to 4K bytes. Thus, for the simple seg 2^N allocator, the measure of the amount of memory used can be over-estimated by up to 4K times the number of size classes, which is roughly $4K * \ln(largest\_size - smallest\_size)$. For the simple seg 2^N & $3 * 2^N$ allocator, the measure of the amount of memory used can be over-estimated by up to 4K times the number of size classes, which is roughly $4K * 2 * \ln(largest\_size - smallest\_size)$.

6 Results

In Table 2 we present the fragmentation results for each of our allocation policies averaged across all eight test programs, for both the third and fourth methods of measuring fragmentation we presented in section 5.

From Table 2 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.

Allocator name	% Frag #3	% Frag #4
first fit AO 8K	0.90%	0.77%
best fit AO 8K	0.98%	0.83%
best fit FIFO	3.94%	2.23%
best fit AO	3.73%	2.27%
Lea 2.6.1	3.81%	2.27%
best fit LIFO	3.76%	2.30%
first fit AO	6.63%	2.30%
first fit FIFO	4.98%	3.14%
next fit AO	13.73%	8.04%
next fit FIFO	20.86%	18.37%
double buddy	36.19%	34.25%
first fit LIFO	50.71%	36.24%
next fit LIFO	52.87%	38.45%
binary buddy	59.69%	53.35%
simple seg 2^N & $3 * 2^N$	1468.22%	61.50%
simple seg 2^N	1818.40%	73.61%

Table 2: Percentage fragmentation (accounting for headers, footers, minimum object size, and hardware required alignment) for all allocators averaged across all programs.

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.

The two allocators that perform best both request their memory from the operating system in 8K chunks rather than 4K chunks as is done by the other allocators. This results in considerably lower fragmentation for one program: Perl. However, Perl only uses 69K of memory, and the difference in fragmentation for this program when using 4K vs. 8K operating system memory requests is within our measurement error. Thus, for our experiments, there is no statistical difference between the performance of the top six allocation policies using either measure of fragmentation.

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 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.

[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.

[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. In preparation.

[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 `www://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, July 1992.

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