Dynamic Storage Allocation: A Survey and Critical Review ⋆ ⋆⋆

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles⋆⋆⋆

Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78751, USA
(wilson|markj|neely@cs.utexas.edu)

Abstract. Dynamic memory allocation has been a fundamental part of most computer systems since roughly 1960, and memory allocation is widely considered to be either a solved problem or an insoluble one. In this survey, we describe a variety of memory allocator designs and point out issues relevant to their design and evaluation. We then chronologically survey most of the literature on allocators between 1961 and 1995. (Scores of papers are discussed, in varying detail, and over 150 references are given.)

We argue that allocator designs have been unduly restricted by an emphasis on mechanism, rather than policy, while the latter is more important; higher-level *strategic* issues are still more important, but have not been given much attention.

Most theoretical analyses and empirical allocator evaluations to date have relied on very strong assumptions of randomness and independence, but real program behavior exhibits important regularities that must be exploited if allocators are to perform well in practice.

1 Introduction

In this survey, we will discuss the design and evaluation of conventional dynamic memory allocators. By "conventional," we mean allocators used for general purpose "heap" storage, where the a program can request a block of memory to store a program object, and free that block at any time. A heap, in this sense, is a pool of memory available for the allocation and deallocation of arbitrary-sized blocks of memory in arbitrary order.[4] An allocated block is typically used to store a program "object," which is some kind of structured data item such as a Pascal record, a C struct, or a C++ object, but not necessarily an object in the sense of object-oriented programming.[5]

Throughout this paper, we will assume that while a block is in use by a program, its contents (a data object) cannot be relocated to compact memory (as is done, for example, in copying garbage collectors [Wil95]). This is the usual situation in most implementations of conventional programming systems (such as C, Pascal, Ada, etc.), where the memory manager cannot find and update pointers to program objects when they are moved.[6] The allocator does not

⋆ **A slightly different version of this paper appears in** *Proc. 1995 Int'l. Workshop on Memory Management*, **Kinross, Scotland, UK, September 27–29, 1995, Springer Verlag LNCS**. This version differs in several very minor respects, mainly in formatting, correction of several typographical and editing errors, clarification of a few sentences, and addition of a few footnotes and citations.

⋆⋆ This work was supported by the National Science Foundation under grant CCR-9410026, and by a gift from Novell, Inc.

⋆⋆⋆ Convex Computer Corporation, Dallas, Texas, USA. (dboles@zeppelin.convex.com)

[4] This sense of "heap" is not to be confused with a quite different sense of "heap," meaning a partially ordered tree structure.

[5] While this is the *typical* situation, it is not the only one. The "objects" stored by the allocator need not correspond directly to language-level objects. An example of this is a growable array, represented by a fixed size part that holds a pointer to a variable-sized part. The routine that grows an object might allocate a new, larger variable-sized part, copy the contents of the old variable-sized part into it, and deallocate the old part. We assume that the allocator knows nothing of this, and would view each of these parts as separate and independent objects, even if normal programmers would see a "single" object.

[6] It is also true of many garbage-collected systems. In

examine the data stored in a block, or modify or act on it in any way. The data areas within blocks that are used to hold objects are contiguous and nonoverlapping ranges of (real or virtual) memory. We generally assume that only entire blocks are allocated or freed, and that the allocator is entirely unaware of the type of or values of data stored in a block—it only knows the size requested.

Scope of this survey. In most of this survey, we will concentrate on issues of overall memory usage, rather than time costs. We believe that detailed measures of time costs are usually a red herring, because they obscure issues of strategy and policy; we believe that most good strategies can yield good policies that are amenable to efficient implementation. (We believe that it's easier to make a very fast allocator than a very memory-efficient one, using fairly straightforward techniques (Section 3.12). Beyond a certain point, however, the effectiveness of speed optimizations will depend on many of the same subtle issues that determine memory usage.)

We will also discuss locality of reference only briefly. Locality of reference is increasingly important, as the difference between CPU speed and main memory (or disk) speeds has grown dramatically, with no sign of stopping. Locality is very poorly understood, however; aside from making a few important general comments, we leave most issues of locality to future research.

Except where locality issues are explicitly noted, we assume that the cost of a unit of memory is fixed and uniform. We do not address possible interactions with unusual memory hierarchy schemes such as compressed caching, which may complicate locality issues and interact in other important ways with allocator design [WLM91, Wil91, Dou93].

We will not discuss specialized allocators for particular applications where the data representations and allocator designs are intertwined.[7]

Allocators for these kinds of systems share many properties with the "conventional" allocators we discuss, but introduce many complicating design choices. In particular, they often allow logically contiguous items to be stored non-contiguously, e.g., in pieces of one or a few fixed sizes, and may allow sharing of parts or (other) forms of data compression. We assume that if any fragmenting or compression of higher-level "objects" happens, it is done above the level of abstraction of the allocator interface, and the allocator is entirely unaware of the relationships between the "objects" (e.g., fragments of higher-level objects) that it manages.

Similarly, parallel allocators are not discussed, due to the complexity of the subject.

Structure of the paper. This survey is intended to serve two purposes: as a general reference for techniques in memory allocators, and as a review of the literature in the field, including methodological considerations. Much of the literature review has been separated into a chronological review, in Section 4. This section may be skipped or skimmed if methodology and history are not of interest to the reader, especially on a first reading. However, some potentially significant points are covered only there, or only made sufficiently clear and concrete there, so the serious student of dynamic storage allocation should find it worthwhile. (It may even be of interest to those interested in the history and philosophy of computer science, as documentation of the development of a scientific paradigm.[8])

The remainder of the current section gives our motivations and goals for the paper, and then frames the central problem of memory allocation—*fragmentation*—and the general techniques for dealing with it.

Section 2 discusses deeper issues in fragmentation, and methodological issues (some of which may be skipped) in studying it.

Section 3 presents a fairly traditional taxonomy of

some, insufficient information is available from the compiler and/or programmer to allow safe relocation; this is especially likely in systems where code written in different languages is combined in an application [BW88]. In others, real-time and/or concurrent systems, it is difficult for the garbage collector to relocate data without incurring undue overhead and/or disruptiveness [Wil95].

[7] Examples inlude specialized allocators for chained-block message-buffers (e.g., [Wol65]), "cdr-coded" list-processing systems [BC79], specialized storage for overlapping strings with shared structure, and allocators used to manage disk storage in file systems.

[8] We use "paradigm" in roughly the sense of Kuhn [Kuh70], as a "pattern or model" for research. The paradigms we discuss are not as broad in scope as the ones usually discussed by Kuhn, but on our reading, his ideas are intended to apply at a variety of scales. We are not necessarily in agreement with all of Kuhn's ideas, or with some of the extreme and anti-scientific purposes they have been put to by some others.

known memory allocators, including several not usually covered. It also explains why such mechanism-based taxonomies are very limited, and may obscure more important policy issues. Some of those policy issues are sketched.

Section 4 reviews the literature on memory allocation. A major point of this section is that the main stream of allocator research over the last several decades has focused on oversimplified (and unrealistic) models of program behavior, and that little is actually known about how to design allocators, or what performance to expect.

Section 5 concludes by summarizing the major points of the paper, and suggesting avenues for future research.

Table of Contents

1.1 Motivation

This paper is motivated by our perception that there is considerable confusion about the nature of memory allocators, and about the problem of memory allocation in general. Worse, this confusion is often unrecognized, and allocators are widely thought to be fairly well understood. In fact, we know little more about allocators than was known twenty years ago, which is not as much as might be expected. The literature on the subject is rather inconsistent and scattered, and considerable work appears to be done using approaches that are quite limited. We will try to sketch a unifying conceptual framework for understanding what is and is not known, and suggest promising approaches for new research.

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

We think it is likely that the widespread use of poor allocators incurs a loss of main and cache memory (and CPU cycles) upwards of of a billion U.S. dollars worldwide—a significant fraction of the world's memory and processor output may be squandered, at huge cost.[9]

Perhaps even worse is the effect on programming style due to the widespread use of allocators that are simply bad ones—either because better allocators are known but not *widely* known or understood, or because allocation research has failed to address the proper issues. Many programmers avoid heap allocation in many situations, because of perceived space or time costs.[10]

It seems significant to us that many articles in non-refereed publications—and a number in refereed publications outside the major journals of operating systems and programming languages—are motivated by extreme concerns about the speed or memory costs of general heap allocation. (One such paper [GM85] is discussed in Section 4.1.) Often, *ad hoc* solutions are used for applications that should not be problematic at all, because at least some well-designed general allocators should do quite well for the workload in question.

We suspect that in some cases, the perceptions are wrong, and that the costs of modern heap allocation are simply overestimated. In many cases, however, it appears that poorly-designed or poorly-implemented allocators have lead to a widespread and quite understandable belief that general heap allocation is necessarily expensive. Too many poor allocators have been supplied with widely-distributed operating systems and compilers, and too few practitioners are aware of the alternatives.

This appears to be changing, to some degree. Many operating systems now supply fairly good allocators, and there is an increasing trend toward marketing libraries that include general allocators which are at least claimed to be good, as a replacement for default allocators. It seems likely that there is simply a lag between the improvement in allocator technology and its widespread adoption, and another lag before programming style adapts. The combined lag is quite long, however, and we have seen several magazine articles in the last year on how to avoid using a general allocator. Postings praising *ad hoc* allocation schemes are very common in the Usenet newsgroups oriented toward real-world programming.

The slow adoption of better technology and the lag in changes in perceptions may not be the only problems, however. We have our doubts about how well allocators are really known to work, based on a fairly thorough review of the literature. We wonder whether some part of the perception is due to occasional pro-

[9] This is an unreliable estimate based on admittedly casual last-minute computations, approximately as follows: there are on the order of 100 million PC's in the world. If we assume that they have an average of 10 megabytes of memory at $30 per megabyte, there is 30 billion dollars worth of RAM at stake. (With the expected popularity of Windows 95, this seems like it will soon become a fairly conservative estimate, if it isn't already.) If just one fifth (6 billion dollars worth) is used for heap-allocated data, and one fifth of that is unnecessarily wasted, the cost is over a billion dollars.

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

grams that interact pathologically with common allocator designs, in ways that have never been observed by researchers.

This does not seem unlikely, because most experiments have used non-representative workloads, which are extremely unlikely to generate the same problematic request patterns as real programs. Sound studies using realistic workloads are too rare. The total number of real, nontrivial programs that have been used for good experiments is very small, apparently less than 20. A significant number of real programs could exhibit problematic behavior patterns that are simply not represented in studies to date.

Long-running processes such as operating systems, interactive programming environments, and networked servers may pose special problems that have not been addressed. Most experiments to date have studied programs that execute for a few minutes (at most) on common workstations. Little is known about what happens when programs run for hours, days, weeks or months. It may well be that some seemingly good allocators do not work well in the long run, with their memory efficiency slowly degrading until they perform quite badly. We don't know—and we're fairly sure that nobody knows. Given that long-running processes are often the most important ones, and are increasingly important with the spread of client/server computing, this is a potentially large problem.

The worst case performance of any general allocator amounts to complete failure due to memory exhaustion or virtual memory thrashing (Section 1.2). This means that any real allocator may have lurking "bugs" and fail unexpectedly for seemingly reasonable inputs.

Such problems may be hidden, because most programmers who encounter severe problems may simply code around them using *ad hoc* storage management techniques—or, as is still painfully common, by statically allocating "enough" memory for variable-sized structures. These ad-hoc approaches to storage management lead to "brittle" software with hidden limitations (e.g., due to the use of fixed-size arrays). The impact on software clarity, flexibility, maintainability, and reliability is quite important, but difficult to estimate. It should not be underestimated, however, because these hidden costs can incur major penalties in productivity and, to put it plainly, human costs in sheer frustration, anxiety, and general suffering.

A much larger and broader set of test applications and experiments is needed before we have any assurance that any allocator *works* reliably, in a crucial performance sense—much less works well. Given this caveat, however, it appears that some allocators are clearly better than others in most cases, and this paper will attempt to explain the differences.

1.2 What an Allocator Must Do

An allocator must keep track of which parts of memory are in use, and which parts are free. The goal of allocator design is usually to minimize wasted space without undue time cost, or vice versa. The ideal allocator would spend negligible time managing memory, and waste negligible space.

A conventional allocator cannot control the number or size of live blocks—these are entirely up to the program requesting and releasing the space managed by the allocator. A conventional allocator also cannot *compact* memory, moving blocks around to make them contiguous and free contiguous memory. It must respond immediately to a request for space, and once it has decided which block of memory to allocate, it cannot change that decision—that block of memory must be regarded as inviolable until the application[11] program chooses to free it. It can only deal with memory that is free, and only choose where in free memory to allocate the next requested block. (Allocators record the locations and sizes of free blocks of memory in some kind of hidden data structure, which may be a linear list, a totally or partially ordered tree, a bitmap, or some hybrid data structure.)

An allocator is therefore an *online* algorithm, which must respond to requests in strict sequence, immediately, and its decisions are irrevocable.

The problem the allocator must address is that the application program may free blocks in any order, creating "holes" amid live objects. If these holes are too numerous and small, they cannot be used to satisfy future requests for larger blocks. This problem is known as *fragmentation*, and it is a potentially disastrous one. For the general case that we have outlined—where the application program may allocate arbitrary-sized objects at arbitrary times and free them at any later time—there is no reliable algorithm for ensuring efficient memory usage, *and none*

[11] We use the term "application" rather generally; the "application" for which an allocator manages storage may be a system program such as a file server, or even an operating system kernel.

is possible. It has been proven that for any possible allocation algorithm, there will always be the possibility that some application program will allocate and deallocate blocks in some fashion that defeats the allocator's strategy, and forces it into severe fragmentation [Rob71, GGU72, Rob74, Rob77]. Not only are there no provably good allocation algorithms, there are proofs that any allocator will be "bad" for some possible applications.

The lower bound on worst case fragmentation is generally proportional to the amount of live data[12] multiplied by the logarithm of the ratio between the largest and smallest block sizes, i.e., $M \log_2 n$, where M is the amount of live data and n is the ratio between the smallest and largest object sizes [Rob71].

(In discussing worst-case memory costs, we generally assume that all block sizes are evenly divisible by the smallest block size, and n is sometimes simply called "the largest block size," i.e., in units of the smallest.)

Of course, for some algorithms, the worst case is much worse, often proportional to the simple *product* of M and n.

So, for example, if the minimum and maximum objects sizes are one word and a million words, then fragmentation in the worst case may cost an excellent allocator a factor of ten or twenty in space. A less robust allocator may lose a factor of a million, in its worst case, wasting so much space that failure is almost certain.

Given the apparent insolubility of this problem, it may seem surprising that dynamic memory allocation is used in most systems, and the computing world does not grind to a halt due to lack of memory. The reason, of course, is that there are allocators that are fairly good in practice, in combination with most actual programs. Some allocation algorithms have been shown in practice to work acceptably well with real programs, and have been widely adopted. If a particular program interacts badly with a particular allocator, a different allocator may be used instead. (The bad cases for one allocator may be very different from the bad cases for other allocators of different design.)

The design of memory allocators is currently something of a black art. Little is known about the interactions between programs and allocators, and which programs are likely to bring out the worst in which allocators. However, one thing is clear—most programs are "well behaved" in some sense. Most programs combined with most common allocators do not squander huge amounts of memory, even if they may waste a quarter of it, or a half, or occasionally even more.

That is, *there are regularities in program behavior that allocators exploit*, a point that is often insufficiently appreciated even by professionals who design and implement allocators. These regularities are exploited by allocators to prevent excessive fragmentation, and make it possible for allocators to work in practice.

These regularities are surprisingly poorly understood, despite 35 years of allocator research, and scores of papers by dozens of researchers.

1.3 Strategies, Placement Policies, and Splitting and Coalescing

The main technique used by allocators to keep fragmentation under control is *placement choice*. Two subsidiary techniques are used to help implement that choice: *splitting* blocks to satisfy smaller requests, and *coalescing* of free blocks to yield larger blocks.

Placement choice is simply the choosing of where in free memory to put a requested block. Despite potentially fatal restrictions on an allocator's online choices, the allocator also has a huge freedom of action—it can place a requested block anywhere it can find a sufficiently large range of free memory, and anywhere within that range. (It may also be able to simply request more memory from the operating system.) An allocator algorithm therefore should be regarded as the mechanism that implements a *placement policy*, which is motivated by a *strategy* for minimizing fragmentation.

Strategy, policy, and mechanism. The *strategy* takes into account regularities in program behavior, and determines a range of acceptable *policies* as to where to allocate requested blocks. The chosen policy is implemented by a *mechanism*, which is a set of algorithms and the data structures they use. This three-level distinction is quite important.

In the context of general memory allocation,

- a *strategy* attempts to exploit regularities in the request stream,

[12] We use "live" here in a fairly loose sense. Blocks are "live" from the point of view of the allocator if it doesn't know that it can safely reuse the storage—i.e., if the block was allocated but not yet freed. This is different from the senses of liveness used in garbage collection or in compilers' flow analyses.

- a *policy* is an implementable decision procedure for placing blocks in memory, and
- a *mechanism* is a set of algorithms and data structures that implement the policy, often over-simply called "an algorithm."[13]

An ideal strategy is "put blocks where they won't cause fragmentation later"; unfortunately that's impossible to guarantee, so real strategies attempt to heuristically approximate that ideal, based on assumed regularities of application programs' behavior. For example, one strategy is "avoid letting small long-lived objects prevent you from reclaiming a larger contiguous free area." This is part of the strategy underlying the common "best fit" family of policies. Another part of the strategy is "if you have to split a block and potentially waste what's left over, minimize the size of the wasted part."

The corresponding (best fit) policy is more concrete—it says "always use the smallest block that is at least large enough to satisfy the request."

The placement policy determines exactly where in memory requested blocks will be allocated. For the best fit policies, the general rule is "allocate objects in the smallest free block that's at least big enough to hold them." That's not a complete policy, however, because there may be several equally good fits; the complete policy must specify which of those should be chosen, for example, the one whose address is lowest.

The chosen policy is implemented by a specific mechanism, chosen to implement that policy efficiently in terms of time and space overheads. For best fit, a linear list or ordered tree structure might be used to record the addresses and sizes of free blocks, and a tree search or list search would be used to find the one dictated by the policy.

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

The strategy and policy are often very poorly-defined, as well, and the policy and mechanism are arrived at by a combination of educated guessing, trial and error, and (often dubious) experimental validation.[14]

[13] This set of distinctions is doubtless indirectly influenced by work in very different areas, notably Marr's work in natural and artificial visual systems [Mar82] and McClamrock's work in the philosophy of science and cognition [McC91, McC95]. The distinctions are important for understanding a wide variety of complex systems, however. Similar distinctions are made in many fields, including empirical computer science, though often without making them quite clear.

In "systems" work, mechanism and policy are often distinguished, but strategy and policy are usually not distinguished explicitly. This makes sense in some contexts, where the policy can safely be assumed to implement a well-understood strategy, or where the choice of strategy is left up to someone else (e.g., designers of higher-level code not under discussion).

In empirical evaluations of very poorly understood strategies, however, the distinction between strategy and policy is often crucial. (For example, errors in the implementation of a strategy are often misinterpreted as evidence that the expected regularities don't actually exist, when in fact they do, and a slightly different strategy would work much better.)

Mistakes are possible at each level; equally important, mistakes are possible *between* levels, in the attempt to "cash out" (implement) the higher-level strategy as a policy, or a policy as a mechanism.

[14] In case the important distinctions between strategy, policy, and mechanism are not clear, a metaphorical example may help. Consider a software company that has a strategy for improving productivity: reward the most productive programmers. It may institute a *policy* of rewarding programmers who produce the largest numbers of lines of program code. To implement this policy, it may use the *mechanisms* of instructing the managers to count lines of code, and providing scripts that count lines of code according to some particular algorithm.

This example illustrates the possible failures at each level, and in the mapping from one level to another. The strategy may simply be wrong, if programmers aren't particularly motivated by money. The policy may not implement the intended strategy, if lines of code are an inappropriate metric of productivity, or if the policy has unintended "strategic" effects, e.g., due to programmer resentment.

The mechanism may also fail to implement the specified policy, if the rules for line-counting aren't enforced by managers, or if the supplied scripts don't correctly implement the intended counting function.

This distinction between strategy and policy is oversimplified, because there may be multiple levels of strategy that shade off into increasingly concrete policies. At different levels of abstraction, something might be

Splitting and coalescing. Two general techniques for supporting a range of (implementations of) placement policies are *splitting* and *coalescing* of free blocks. (These mechanisms are important subsidiary parts of the larger mechanism that is the allocator implementation.)

The allocator may split large blocks into smaller blocks arbitrarily, and use any sufficiently-large subblock to satisfy the request. The remainders from this splitting can be recorded as smaller free blocks in their own right and used to satisfy future requests.

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

Completely general splitting and coalescing can be supported at fairly modest cost in space and/or time, using simple mechanisms that we'll describe later. This allows the allocator designer the maximum freedom in choosing a strategy, policy, and mechanism for the allocator, because the allocator can have a complete and accurate record of which ranges of memory are available at all times.

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

viewed as a strategy or policy.

The key point is that there are *at least* three qualitatively different *kinds* of levels of abstraction involved [McC91]; at the upper levels, there are is the general design goal of exploiting expected regularities, and a set of strategies for doing so; there may be subsidiary strategies, for example to resolve conflicts between strategies in the best possible way.

At at a somewhat lower level there is a general policy of where to place objects, and below that is a more detailed policy that exactly determines placement.

Below that there is an actual mechanism that is intended to implement the policy (and presumably effect the strategy), using whatever algorithms and data structures are deemed appropriate. Mechanisms are often layered, as well, in the usual manner of structured programming [Dij69]. Problems at (and between) these levels are the best understood—a computation may be improperly specified, or may not meet its specification. (Analogous problems occur at the upper levels occur as well—if expected regularities don't actually occur, or if they do occur but the strategy does't actually exploit them, and so on.)

2 A Closer Look at Fragmentation, and How to Study It

In this section, we will discuss the traditional conception of fragmentation, and the usual techniques used for studying it. We will then explain why the usual understanding is not strong enough to support scientific design and evaluation of allocators. We then propose a new (though nearly obvious) conception of fragmentation and its causes, and describe more suitable techniques used to study it. (Most of the experiments using sound techniques have been performed in the last few years, but a few notable exceptions were done much earlier, e.g., [MPS71] and [LH82], discussed in Section 4.)

2.1 Internal and External Fragmentation

Traditionally, fragmentation is classed as *external* or *internal* [Ran69], and is combatted by splitting and coalescing free blocks.

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

Internal fragmentation arises when a large-enough free block is allocated to hold an object, but there is a poor fit because the block is larger than needed. In some allocators, the remainder is simply wasted, causing internal fragmentation. (It's called *internal* because the wasted memory is inside an allocated block,

rather than being recorded as a free block in its own right.)

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

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

2.2 The Traditional Methodology: Probabilistic Analyses, and Simulation Using Synthetic Traces

(Note: readers who are uninterested in experimental methodology may wish to skip this section, at least on a first reading. Readers uninterested in the history of allocator research may skip the footnotes. The following section (2.3) is quite important, however, and should not be skipped.)

Allocators are sometimes evaluated using probabilistic analyses. By reasoning about the likelihood of certain events, and the consequences of those events for future events, it may be possible to predict what will happen on average. For the general problem of dynamic storage allocation, however, the mathematics are too difficult to do this for most algorithms and most workloads. An alternative is to do simulations, and find out "empirically" what really happens when workloads interact with allocator policies. This is more common, because the interactions are so poorly understood that mathematical techniques are difficult to apply.

Unfortunately, in both cases, to make probabilistic techniques feasible, important characteristics of the workload must be known—i.e., the probabilities of relevant characteristics of "input" events to the allocation routine. The relevant characteristics are not understood, and so the probabilities are simply unknown.

This is one of the major points of this paper. The paradigm of statistical mechanics[15] has been used in theories of memory allocation, but we believe that it is the wrong paradigm, at least as it is usually applied. Strong assumptions are made that frequencies of individual events (e.g., allocations and deallocations) are the base statistics from which probabilistic models should be developed, and we think that this is false.

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

This paradigm is inappropriate for memory allocation, for two reasons. The first is simply that the number of objects involved is usually too small for asymptotic analyses to be relevant, but this is not the most important reason.

The main weakness of the "statistical mechanics" approach is that there are important *systematic* interactions that occur in memory allocation, due to phase behavior of programs. No matter how large the system is, basing probabilistic analyses on individual events is likely to yield the wrong answers, if there are systematic effects involved which are not captured by the theory. Assuming that the analyses are appropriate for "sufficiently large" systems does not help here—the systematic errors will simply attain greater statistical significance.

Consider the case of evolutionary biology. If an overly simple statistical approach about individual animals' interactions is used, the theory will not capture predator/prey and host/symbiote relationships, sexual selection, or other pervasive evolutionary effects as niche filling.[16] Developing a highly predictive

[15] This usage of "statistical mechanics" should perhaps be regarded as metaphorical, since it is not really about simple interactions of large numbers of molecules in a gas or liquid. Several papers on memory allocation have used it loosely, however, to describe the analogous approach to analyzing memory allocation. Statistical mechanics has literally provided a *paradigm*—in the original, smaller sense of a "model" or "examplar," rather than in a larger Kuhnian sense—which many find attractive.

[16] Some of these effects *may* emerge from lower-level modeling, but for simulations to reliably predict them, *many* important lower-level issues must be modeled correctly, and sufficient data are usually not available, or suffi-

evolutionary theory is extremely difficult—and some would say impossible—because too many low-level (or higher-level) details matter,[17] and there may intrinsic unpredictabilities in the systems described [Den95].[18]

We are not saying that the development of a good theory of memory allocation is as hard as developing a predictive evolutionary theory—far from it. The problem of memory allocation seems far simpler, and we are optimistic that a useful predictive theory can be developed.[19]

Our point is simply that the paradigm of simple statistical mechanics must be evaluated relative to other alternatives, which we find more plausible in this domain. There are major interactions between workloads and allocator policies, which are usually ignored. No matter how large the system, and no matter how asymptotic the analyses, ignoring these effects seems likely to yield major errors—e.g., analyses will simply yield the wrong asymptotes.

A useful probabilistic theory of memory allocation may be possible, but if so, it will be based on a quite different set of statistics from those used so far—statistics which capture effects of systematicities, rather than assuming such systematicities can be ignored. As in biology, the theory must be tested against reality, and refined to capture systematicities that had previously gone unnoticed.

Random simulations. The traditional technique for evaluating allocators is to construct several *traces* (recorded sequences of allocation and deallocation requests) thought to resemble "typical" workloads, and use those traces to drive a variety of actual allocators. Since an allocator normally responds only to the request sequence, this can produce very accurate simulations of what the allocator would do if the workload were real—that is, if a real program generated that request sequence.

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

Given a set of object size and lifetime distributions, the small "driver" subprogram generates a sequence of requests that obeys those distributions. This driver is simply a loop that repeatedly generates requests, using a pseudo-random number generator; at any point in the simulation, the next data object is chosen by "randomly" picking a size and lifetime, with a bias that (probabilistically) preserves the desired distributions. The driver also maintains a table of objects that have been allocated but not yet freed, ordered by their scheduled death (deallocation) time. (That is, the step at which they were allocated, plus their randomly-chosen lifetime.) At each step of the simulation, the driver deallocates any objects whose death times indicate that they have expired. One convenient measure of simulated "time" is the volume of objects allocated so far—i.e., the sum of the sizes of objects that have been allocated up to that step of the simulation.[20]

An important feature of these simulations is that they tend to reach a "steady state." After running for a certain amount of time, the volume of live (simu-

ciently understood.

[17] For example, the different evolutionary strategies implied by the varying replication techniques and mutation rates of RNA-based vs. DNA-based viruses, or the impact of environmental change on host/parasite interactions [Gar94].

[18] For example, a single chance mutation that results in an adaptive characteristic in one individual may have a major impact on the subsequent evolution of a species and its entire ecosystem [Dar59].

[19] We are also not suggesting that evolutionary theory provides a good paradigm for allocator research; it is just an example of a good scientific paradigm that is very *different* from the ones typically seen in memory allocation research. It demonstrates the important and necessary interplay between high-level theories and detailed empirical work.

[20] In many early simulations, the simulator modeled real time, rather than just discrete steps of allocation and deallocation. Allocation times were chosen based on randomly chosen "arrival" times, generated using an "inter-arrival distribution" and their deaths scheduled in continuous time—rather than discrete time based on the number and/or sizes of objects allocated so far. We will generally ignore this distinction in this paper, because we think other issues are more important. As will become clear, in the methodology we favor, this distinction is not important because the actual sequences of actions are sufficient to guarantee exact simulation, and the actual sequence of events is recorded rather than being (approximately) emulated.

lated) objects reaches a level that is determined by the size and lifetime distributions, and after that objects are allocated and deallocated in approximately equal numbers. The memory usage tends to vary very little, wandering probabilistically (in a random walk) around this "most likely" level. Measurements are typically made by sampling memory usage at points after the steady state has presumably been reached, or by averaging over a period of "steady-state" variation. These measurements "at equilibrium" are assumed to be important.

There are three common variations of this simulation technique. One is to use a simple mathematical function to determine the size and lifetime distributions, such as uniform or (negative) exponential. Exponential distributions are often used because it has been observed that programs are typically more likely to allocate small objects than large ones,[21] and are more likely to allocate short-lived objects than long-lived ones.[22] (The size distributions are generally truncated at some plausible minimum and maximum object size, and discretized, rounding them to the nearest integer.)

The second variation is to pick distributions intuitively, i.e., out of a hat, but in ways thought to resemble real program behavior. One motivation for this is to model the fact that many programs allocate objects of some sizes and others in small numbers or not at all; we refer to these distributions as "spiky."[23]

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

In general, there has been something of a trend toward the use of more realistic distributions,[24] but this trend is not dominant. Even now, researchers often use simple and smooth mathematical functions to generate traces for allocator evaluation.[25] The use of smooth distributions is questionable, because it bears directly on issues of fragmentation—if objects of only a few sizes are allocated, the free (and uncoalescable) blocks are likely to be of those sizes, making it possible to find a perfect fit. If the object sizes are smoothly distributed, the requested sizes will almost always be slightly different, increasing the chances of fragmentation.

Probabilistic analyses. Since Knuth's derivation of the "fifty percent rule" [Knu73] (discussed later, in Section 4), there have been many attempts to reason probabilistically about the interactions between program behavior and allocator policy, and assess the overall cost in terms of fragmentation (usually) and/or CPU time.

These analyses have generally made the same assumptions as random-trace simulation experiments—e.g., random object allocation order, independence of size and lifetimes, steady-state behavior—and often stronger assumptions as well.

These simplifying assumptions have generally been made in order to make the mathematics tractable. In particular, assumptions of randomness and independence make it possible to apply well-developed theory

[21] Historically, uniform size distributions were the most common in early experiments; exponential distributions then became increasingly common, as new data became available showing that real systems generally used many more small objects than large ones. Other distributions have also been used, notably Poisson and hyper-exponential. Still, relatively recent papers have used uniform size distributions, sometimes as the only distribution.

[22] As with size distributions, there has been a shift over time toward non-uniform lifetime distributions, often exponential. This shift occurred later, probably because real data on size information was easier to obtain, and lifetime data appeared later.

[23] In general, this modeling has not been very precise. Sometimes the sizes chosen out of a hat are allocated in uniform proportions, rather than in skewed proportions reflecting the fact that (on average) programs allocate many more small objects than large ones.

[24] The trend toward more realistic distributions can be explained historically and pragmatically. In the early days of computing, the distributions of interest were usually the distribution of segment sizes in an operating system's workload. Without access to the inside of an operating system, this data was difficult to obtain. (Most researchers would not have been allowed to modify the implementation of the operating system running on a very valuable and heavily-timeshared computer.) Later, the emphasis of study shifted away from segment sizes in segmented operating systems, and toward data object sizes in the virtual memories of individual processes running in paged virtual memories.

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

of stochastic processes (Markov models, etc.) to derive analytical results about expected behavior. Unfortunately, these assumptions tend to be false for most real programs, so the results are of limited utility.

It should be noted that these are not merely convenient simplifying assumptions that allow solution of problems that closely resemble real problems. If that were the case, one could expect that with refinement of the analyses—or with sufficient empirical validation that the assumptions don't matter in practice—the results would come close to reality. There is no reason to expect such a happy outcome. These assumptions dramatically change the key features of the problem; the ability to perform the analyses hinges on the very facts that make them much less relevant to the general problem of memory allocation.

Assumptions of randomness and independence make the problem irregular, in a superficial sense, but they make it very smooth (hence mathematically tractable) in a probabilistic sense. This smoothness has the advantage that it makes it possible to derive analytical results, but it has the disadvantage that it turns a real and deep scientific problem into a mathematical puzzle that is much less significant for our purposes.

The problem of dynamic storage allocation is intractable, in the vernacular sense of the word. As an essentially data-dependent problem, *we do not have a grip on it*, because it because we simply do not understand the inputs. "Smoothing" the problem to make it mathematically tractable "removes the handles" from something that is fundamentally irregular, making it unlikely that we will get any real purchase or leverage on the important issues. Removing the irregularities removes some of the problems—and most of the opportunities as well.

A note on exponentially-distributed random lifetimes. Exponential lifetime distributions have become quite common in both empirical and analytic studies of memory fragmentation over the last two decades. In the case of empirical work (using random-trace simulations), this seems an admirable adjustment to some observed characteristics of real program behavior. In the case of analytic studies, it turns out to have some very convenient mathematical properties as well. Unfortunately, it appears that the apparently exponential appearence of real lifetime distributions is often an artifact of experimental methodology (as will be explained in Sections 2.3 and 4.1) and that the emphasis on distributions tends to distract researchers from the *strongly patterned* underlying processes that actually generate them (as will be explained in Section 2.4).

We invite the reader to consider a randomly-ordered trace with an exponential lifetime distribution. In this case there is no correlation at all between an object's age and its expected time until death—the "half-life" decay property of the distribution and the randomness ensure that allocated objects die *completely at random* with no way to estimate their death times from any of the information available to the allocator.[26] (An exponential random function exhibits only a half-life property, and no other pattern, much like radioactive decay.) In a sense, exponential lifetimes are thus the *reductio ad absurdum* of the synthetic trace methodology—all of the time-varying regularities have been systematically eliminated from the input. If we view the allocator's job as an online problem of detecting and exploiting regularities, we see that this puts the allocator in the awkward position of trying to extract helpful hints from pure noise.

This does not necessarily mean that all allocators will perform identically under randomized workloads, however, because there are regularities in size distributions, whether they are real distributions or simple mathematical ones, and some allocators may simply shoot themselves in the foot.

Analyses and experiments with exponentially distributed random lifetimes may say something revealing about what happens when an allocator's strategy is completely orthogonal to the actual regularities. We have no real idea whether this is a situation that occurs regularly in the space of possible combinations of real workloads and reasonable strategies.[27] (It's clear that it is not the usual case, however.) The terrain of that space is quite mysterious to us.

A note on Markov models. Many probabilistic studies of memory allocation have used first-order

[26] We are indebted to Henry Baker, who has made quite similar observations with respect to the use of exponential lifetime distributions to estimate the effectiveness of generational garbage collection schemes [Bak93].

[27] In particular, certain effects of randomized traces *may* (or may not) resemble the cumulative effect of allocator strategy errors over much longer periods. This resemblance cannot be *assumed*, however—there are good reasons to think it may occur in some cases, but not in others, and empirical validation is necessary.

Markov processes to approximate program and allocator behavior, and have derived conclusions based on the well-understood properties of Markov models.

In a first-order Markov model, the probabilities of state transitions are known and fixed. In the case of fragmentation studies, this corresponds to assuming that a program allocates objects at random, with fixed probabilities of allocating different sizes.

The space of possible states of memory is viewed as a graph, with a node for each configuration of allocated and free blocks. There is a start state, representing an empty memory, and a transition probability for each possible allocation size. For a given placement policy, there will be a known transition from a given state for any possible allocation or deallocation request. The state reached by each possible allocation is another configuration of memory.

For any given request distribution, there is a network of possible states reachable from the start state, via successions of more or less probable transitions. In general, for any memory above a very, very small size, and for arbitrary distributions of sizes and lifetimes, this network is inconceivably large. As described so far, it is therefore useless for any practical analyses.

To make the problem more tractable, certain assumptions are often made. One of these is that lifetimes are exponentially distributed as well as random, and have the convenient half-life property described above, i.e., they die completely at random as well as being born at random.

This assumption can be used to ensure that both the states and the transitions between states have definite probabilities in the long run. That is, if one were to run a random-trace simulation for a long enough period of time, all reachable states would be reached, and all of them would be reached many times—and the number of times they were reached would reflect the probabilities of their being reached again in the future, if the simulation were continued indefinitely. If we put a counter on each of the states to keep track of the number of times each state was reached, the ratio between these counts would eventually stabilize, plus or minus small short-term variations. The relative weights of the counters would "converge" to a stable solution.

Such a network of states is called an *ergodic* Markov model, and it has very convenient mathematical properties. In some cases, it's possible to avoid running a simulation at all, and analytically derive what the network's probabiblities would converge to.

Unfortunately, this is a very inappropriate model for real program and allocator behavior. An ergodic Markov model is a kind of (probabilistic) finite automaton, and as such the patterns it generates are very, very simple, though randomized and hence unpredictable. They're almost unpatterned, in fact, and hence very predictable in a certain probabilistic sense.

Such an automaton is extremely unlikely to generate many patterns that seem likely to be important in real programs, such as the creation of the objects in a linked list in one order, and their later destruction in exactly the same order, or exactly the reverse order.[28] There are much more powerful kinds of machines—which have more complex state, like a real program—which are capable of generating more realistic patterns. Unfortunately, the only machines that we are sure generate the "right kinds" of patterns are actual real programs.

We do not understand what regularities exist in real programs well enough to model them formally and perform probabilistic analyses that are directly applicable to real program behavior. The models we have are grossly inaccurate in respects that are quite relevant to problems of memory allocation.

There are problems for which Markov models are useful, and a smaller number of problems where assumptions of ergodicity are appropriate. These problems involve processes that are literally random, or can be shown to be effectively random in the necessary ways. The general heap allocation problem is *not* in either category. (If this is not clear, the next section should make it much clearer.)

Ergodic Markov models are also sometimes used for problems where the basic assumptions are known to be false in some cases—but they should only be used in this way if they can be *validated*, i.e., shown by extensive testing to produce the right answers most of the time, despite the oversimplifications they're based on. For some problems it "just turns out" that the differences between real systems and the mathematical models are not usually significant. For the general problem of memory allocation, this turns out to be false as well—recent results clearly *invalidate* the use

[28] Technically, a Markov model will eventually generate such patterns, but the probability of generating a particular pattern within a finite period of time is vanishingly small if the pattern is large and not very strongly reflected in the arc weights. That is, many quite probable kinds of patterns are extremely improbable in a simple Markov model.

of simple Markov models [ZG94, WJNB95].[29]

2.3 What Fragmentation Really Is, and Why the Traditional Approach is Unsound

> A single death is a tragedy. A million deaths is a statistic.
> —*Joseph Stalin*

We suggested above that the shape of a size distribution (and its smoothness) might be important in determining the fragmentation caused by a workload. However, even if the distributions are completely realistic, there is reason to suspect that randomized synthetic traces are likely to be grossly unrealistic.

As we said earlier, the allocator should embody a strategy designed to exploit regularities in program behavior—otherwise it cannot be expected to do particularly well. The use of randomized allocation order eliminates some regularities in workloads, and introduces others, and there is every reason to think that the differences in regularities will affect the performance of different strategies differently. To make this concrete, we must understand fragmentation and its causes.

The technical distinction between internal and external fragmentation is useful, but in attempting to design experiments measuring fragmentation, it is worthwhile to stop for a moment and consider what fragmentation *really* is, and how it arises.

Fragmentation is the inability to reuse memory that is free. This can be due to policy choices by the allocator, which may choose not to reuse memory that in principle could be reused. More importantly for our purposes, the allocator may not have a choice at the moment an allocation request must be serviced: there may be free areas that are too small to service the request *and whose neighbors are not free*, making it impossible to coalesce adjacent free areas into a sufficiently large contiguous block.[30]

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

We have not made the notion of fragmentation particularly clear or quantifiable here, and this is no accident. An allocator's inability to reuse memory depends not only on the number and sizes of holes, but on the future behavior of the program, and the future responses of the allocator itself. (That is, it is a complex matter of interactions between patterned workloads and strategies.)

For example, suppose there are 100 free blocks of size 10, and 200 free blocks of size 20. Is memory highly fragmented? It depends. If future requests are all for size 10, most allocators will do just fine, using the size 10 blocks, and splitting the size 20 blocks as necessary. But if the future requests are for blocks of size 30, that's a problem. Also, if the future requests are for 100 blocks of size 10 and 200 blocks of size 20, whether it's a problem may depend on the order in which the requests arrive and the allocator's moment-

[29] It might seem that the problem here is the use of *first-order* Markov models, whose states (nodes in the reachability graph) correspond directly to states of memory. Perhaps "higher-order" Markov models would work, where nodes in the graph represent sequences of concrete state transitions. We think this is false as well.

The important kinds of patterns produced by real programs are generally not simple very-short-term sequences of a few events, but large-scale patterns involving many events. To capture these, a Markov model would have to be of such high order that analyses would be completely infeasible. It would essentially have to be pre-programmed to generate specific *literal* sequences of events. This not only begs the essential question of what real programs do, but seems certain *not* to concisely capture the right regularities.

Markov models are simply not powerful enough—i.e., not abstract enough *in the right ways*—to help with this problem. They should not be used for this purpose, or any similarly poorly understood purpose, where complex patterns may be very important. (At least, not without extensive validation.) The fact that the regularities are complex and unknown is *not* a good reason to assume that they're effectively random [ZG94, WJNB95] (Section 4.2).

[30] Beck [Bec82] makes the only clear statement of this principle which we have found in our exhausting review of the literature. As we will explain later (in our chronological review, Section 4.1), Beck also made some important inferences from this principle, but his theoretical model and his empirical methodology were weakened by working within the dominant paradigm. His paper is seldom cited, and its important ideas have generally gone unnoticed.

by-moment decisions as to where to place them. Best fit will do well for this example, but other allocators do better for some other examples where best fit performs abysmally.

We leave the concept of fragmentation somewhat poorly defined, because in the general case the actual phenomenon is poorly defined.[31]

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

Fragmentation is caused by time-varying behavior. Fragmentation arises from *changes* in the way a program uses memory—for example, freeing small blocks and requesting large ones. This much is obvious, but it is important to consider patterns in the changing behavior of a program, such as the freeing of large numbers of objects and the allocation of large numbers of objects of different types. Many programs allocate and free different kinds of objects in different stereotyped ways. Some kinds of objects accumulate over time, but other kinds may be used in bursty patterns. (This will be discussed in more detail in Section 2.4.) The allocator's job is to exploit these patterns, if possible, or at least not let the patterns undermine its strategy.

Implications for experimental methodology. (Note: this section is concerned only with experimental techniques; uninterested readers may skip to the following section.)

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

It should be clear that random object deaths may systematically create serious fragmentation in ways that are unlikely to be realistic. Randomization also has a potentially large effect on large-scale *aggregate* behavior of large numbers of objects. In real programs, the total volume of objects varies over time, and often the relative volumes of objects of different sizes varies as well. This often occurs due to phase behavior—some phases may use many more objects than others, and the objects used by one phase may be of very different sizes than those used by another phase.

Now consider a randomized synthetic trace—the overall volume of objects is determined by a random walk, so that the volume of objects rises gradually until a steady state is reached. Likewise the volume of memory allocated to objects of a given size is a similar random walk. If the number of objects of a given size is large, the random walk will tend to be relatively smooth, with mostly gradual and small changes in overall allocated volume. This implies that *the proportions of memory allocated to different-sized objects*

[31] Our concept of fragmentation has been called "startlingly nonoperational," and we must confess that it is, to some degree. We think that this is a strength, however, because it is better to leave a concept somewhat vague than to define it prematurely and incorrectly. It is important to first identify the "natural kinds" in the phenomena under study, and then figure out what their most important characteristics are [Kri72, Put77, Qui77]. (We are currently working on developing operational measures of "fragmentation-related" program behavior.)

Later in the paper we will express experimental "fragmentation" results as percentages, but this should be viewed as an operational shorthand for the *effects* of fragmentation on memory usage at whatever point or points in program execution measurements were made; this should be clear in context.

tend to be relatively stable.

This has major implications for external fragmentation. External fragmentation means that there are free blocks of memory of some sizes, but those are the wrong sizes to satisfy current needs. This happens when objects of one size are freed, and then objects of another size are allocated—that is, when there is an unfortunate change in the relative proportions of objects of one size and objects of a larger size. (For allocators that never split blocks, this can happen with requests for smaller sizes as well.) For synthetic random traces, this is less likely to occur—they don't systematically free objects of one size and then allocate objects of another. Instead, they tend to allocate and free objects of different sizes in relatively stable proportions. This minimizes the need to coalesce adjacent free areas to avoid fragmentation; on average, a free memory block of a given size will be reused relatively soon. This may bias experimental results by hiding an allocator's inability to deal well with external fragmentation, and favor allocators that deal well with internal fragmentation at a cost in external fragmentation.

Notice that while random deaths cause fragmentation, the aggregate behavior of random walks may reduce the extent of the problem. For some allocators, this balance of unrealistically bad and unrealistically good properties may average out to something like realism, but for others it may not. Even if—by sheer luck—random traces turn out to yield realistic fragmentation "on average," over many allocators, they are inadequate for comparing different allocators, which is usually the primary goal of such studies.

2.4 Some Real Program Behaviors

> ...and suddenly the memory returns.
> —*Marcel Proust, Swann's Way*

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

There *are* strong regularities in many real programs, however, because similar techniques are applied (in different combinations) to solve many problems. Several common patterns have been observed.

Ramps, peaks, and plateaus. In terms of overall memory usage over time, three patterns have been observed in a variety of programs in a variety of contexts. Not all programs exhibit all of these patterns, but most seem to exhibit one or two of them, or all three, to some degree. Any generalizations based on these patterns must therefore be *qualitative* and *qualified*. (This implies that to understand the quantitative importance of these patterns, a small set of programs is not sufficient.)

- *Ramps.* Many programs accumulate certain data structures monotonically over time. This may be because they keep a log of events, or because the problem-solving strategy requires building a large representation, after which a solution can be found quickly.
- *Peaks.* Many programs use memory in bursty patterns, building up relatively large data structures which are used for the duration of a particular phase, and then discarding most or all of those data structures. Note that the "surviving" data structures are likely to be of different types, because they represent the results of a phase, as opposed to intermediate values which may be represented differently. (A peak is like a ramp, but of shorter duration.)
- *Plateaus.* Many programs build up data structures quickly, and then use those data structures for long periods (often nearly the whole running time of the program).

These patterns are well-known, from anecdotal experience by many people (e.g., [Ros67, Han90]), from research on garbage collection (e.g., [Whi80, WM89, UJ88, Hay91, Hay93, BZ95, Wil95]),[32] and from a recent study of C and C++ programs [WJNB95].

[32] It may be thought that garbage collected systems are sufficiently different from those using conventional storage management that these results are not relevant. It appears, however, that these patterns are common in both kinds of systems, because similar problem-solving strategies are used by programmers in both kinds of systems. (For any particular problem, different qualitative program behaviors may result, but the general categories seem to be common in conventional programs as well. See [WJNB95].)

(Other patterns of overall memory usage also occur, but appear less common. As we describe in Section 4, backward ramp functions have been observed [GM85]. Combined forward and backward ramp behavior has also been observed, with one data structure shrinking as another grows [Abr67].)

Notice that in the case of ramps and ramp-shaped peaks, looking at the statistical distributions of object lifetimes may be very misleading. A statistical distribution suggests a random decay process of some sort, but it may actually reflect sudden deaths of *groups* of objects that are *born* at different times. In terms of fragmentation, the difference between these two models is major. For a statistical decay process, the allocator is faced with isolated deaths, which are likely to cause fragmentation. For a phased process where many objects often die at the same time, the allocator is presented with an opportunity to get back a significant amount of memory all at once.

In real programs, these patterns may be composed in different ways at different scales of space and time. A ramp may be viewed as a kind of peak that grows over the entire duration of program execution. (The distinction between a ramp and a peak is not precise, but we tend to use "ramp" to refer to something that grows slowly over the whole execution of a program, and drops off suddenly at the end, and "peak" to refer to faster-growing volumes of objects that are discarded before the end of execution. A peak may also be flat on top, making it a kind of tall, skinny plateau.)

While the overall long-term pattern is often a ramp or plateau, it often has smaller features (peaks or plateus) added to it. This crude model of program behavior is thus recursive. (We note that it is *not* generally fractal[33]—features at one scale may bear no resemblance to features at another scale. Attempting to characterize the behavior of a program by a simple number such as fractal dimension is not appropriate, because program behavior is not that simple.[34])

Ramps, peaks, and plateus have very different implications for fragmentation.

An overall ramp or plateau profile has a very convenient property, in that if short-term fragmentation can be avoided, long term fragmentation is not a problem either. Since the data making up a plateau are stable, and those making up a ramp accumulate monotonically, inability to reuse freed memory is not an issue—nothing is freed until the end of program execution. Short-term fragmentation can be a cumulative problem, however, leaving many small holes in the mass of long lived-objects.

Peaks and tall, skinny plateaus can pose a challenge in terms of fragmentation, since many objects are allocated and freed, and many other objects are likely to be allocated and freed later. If an earlier phase leaves scattered survivors, it may cause problems for later phases that must use the spaces in between.

More generally, phase behavior is the major cause of fragmentation—if a program's needs for blocks of particular sizes *change* over time in an awkward way. If many small objects are freed at the end of a phase—but scattered objects survive—a later phase may run into trouble. On the other hand, if the survivors happen to have been placed together, large contiguous areas will come free.

Fragmentation at peaks is important. Not all periods of program execution are equal. The most important periods are usually those when the most memory is used. Fragmentation is less important at times of lower overall memory usage than it is when memory usage is "at its peak," either during a short-lived peak or near the end of a ramp of gradually increas-

[33] We are using the term "fractal" rather loosely, as is common in this area. Typically, "fractal" models of program behavior are not infinitely recursive, and are actually graftals or other finite fractal-like recursive entities.

[34] We believe that this applies to studies of locality of reference as well. Attempts to characterize memory referencing behavior as fractal-like (e.g., [VMH+83, Thi89]) are ill-conceived or severely limited—if only because memory allocation behavior is not generally fractal, and memory-referencing behavior depends on memory allocation policy. (We suspect that it's ill-conceived for understanding program behavior at the level of references to objects, as well as at the level of references to memory.) If the fractal concept is used in a strong sense, we believe it is simply wrong. If it is taken in a weak sense, we believe it conveys little useful information that couldn't be better summarized by simple statistical curve-fitting; using a fractal conceptual framework tends to obscure more issues than it clarifies. Average program behavior may resemble a fractal, because similar features can occur at different scales in different programs; however, an individual program's behavior is *not* fractal-like in general, any more than it is a simple Markov process. Both kinds of models fail to capture the "irregularly regular" and scale-dependent kinds of patterns that are most important.

ing memory usage. This means that average fragmentation is less important than peak fragmentation—scattered holes in the heap *most of the time* may not be a problem if those holes are well-filled *when it counts.*

This has implications for the interpretation of analyses and simulations based on steady-state behavior (i.e., equilibrium conditions). Real programs may exhibit some steady-state behavior, but there are usually ramps and/or peaks as well. It appears that *most programs never reach a truly steady state*, and if they reach a temporary steady state, *it may not matter much.* (It *can* matter, however, because earlier phases may result in a configuration of blocks that is more or less problematic later on, at peak usage.)

Overall memory usage is not the whole story, of course. Locality of reference matters as well. All other things being equal, however, a larger total "footprint" matters even for locality. In virtual memories, many programs never page at all, or suffer dramatic performance degradations if they do. Keeping the overall memory usage lower makes this less likely to happen. (In a time-shared machine, a larger footprint is likely to mean that a *different* process has its pages evicted when the peak is reached, rather than its own less-recently-used pages.)

Exploiting ordering and size dependencies. If the allocator can exploit the phase information from the request stream, it may be able to place objects that will die at about the same time in a contiguous area of memory. This may suggest that the allocator should be adaptive,[35] but much simpler strategies also seem likely to work [WJNB95]:

- Objects allocated at about the same time are likely to die together at the end of a phase; if consecutively-allocated objects are allocated in contiguous memory, they will free contiguous memory.
- Objects of different types may be likely to serve different purposes and die at different times. Size is likely to be related to type and purpose, so avoiding the intermingling of different sizes (and likely types) of objects may reduce the scattering of long-lived objects among short-lived ones.

[35] Barrett and Zorn have recently built an allocator using profile information to heuristically separate long-lived objects from short-lived ones [BZ93]. (Section 4.2.)

This suggests that objects allocated at about the same time should be allocated adjacent to each other in memory, with the possible amendment that different-sized objects should be segregated [WJNB95].[36]

Implications for strategy. The phased behavior of many programs provides an opportunity for the allocator to reduce fragmentation. As we said above, if successive objects are allocated contiguously and freed at about the same time, free memory will again be contiguous. We suspect that this happens with many existing allocators—even though they were not designed with this principle in mind, as far as we can tell. It may well be that this accidental "strategy" is the major way that good allocators keep fragmentation low.

Implications for research. A major goal of allocator research should be to determine which patterns are common, and which can be exploited (or at least guarded against). Strategies that work well for one program may work poorly for another, but it may be possible to combine strategies in a single robust policy that works well for almost all programs. If that fails, it may be possible to have a small set of allocators with different properties, at least one of which works well for the vast majority of real problems.

We caution against blindly experimenting with different combinations of programs and complex, optimized allocators, however. It is more important to determine what regularities exist in real program behavior, and only then decide which strategies are most

[36] We have not found any other mention of these heuristics in the literature, although somewhat similar ideas underlie the "zone" allocator of Ross [Ros67] and Hanson's "obstack" system (both discussed later). Beck [Bec82], Demers et al. [DWH+90], and and Barrett and Zorn [BZ93] have developed systems that predict the lifetimes of objects for similar purposes.

We note that for our purposes, it is not necessary to predict *which* groups of objects will die *when*. It is only necessary to predict which groups of objects will die at similar times, and which will die at dissimilar times, without worrying about which group will die first. We refer to this as "death time discrimination." This simpler discrimination seems easier to achieve than lifetime prediction, and possibly more robust. Intuitively, it also seems more directly related to the causes of fragmentation.

appropriate, and which good strategies can be combined successfully. This is not to say that experiments with many variations on many designs aren't useful—we're in the midst of such experiments ourselves—but that the goal should be to identify fundamental interactions rather than just "hacking" on things until they work well for a few test applications.

Profiles of some real programs. To make our discussion of memory usage patterns more concrete, we will present profiles of memory use for some real programs. Each figure plots the overall amount of live data for a run of the program, and also the amounts of data allocated to objects of the five most popular sizes. ("Popularity" here means most volume allocated, i.e., sum of sizes, rather than object counts.) These are profiles of *program* behavior, independent of any particular allocator.

GCC. Figure 1 shows memory usage for GCC, the GNU C compiler, compiling the largest file of its own source code (`combine.c`). (A high optimization switch was used, encouraging the compiler to perform extensive inlining, analyses, and optimization.) We used a trace processor to remove "obstack" allocation from the trace, creating a trace with the equivalent allocations and frees of individual objects; obstacks are heavily used in this program.[37] The use of obstacks may affect programming style and memory usage patterns; however, we suspect that the memory usage patterns would be similar without obstacks, and that obstacks are simply used to exploit them.[38]

This is a heavily phased program, with several strong and similar peaks. These are two-horned peaks, where one (large) size is allocated and deallocated, and much smaller size is allocated and deallocated, out of phase.[39] (This is an unusual feature, in our limited experience.) Notice that this program exhibits very different usage profiles for different sized objects. The use of one size is nearly steady, another is strongly peaked, and others are peaked, but different.

Grobner. Figure 2 shows memory usage for the Grobner program[40] which decomposes complex expressions into linear combinations of polynomials (Gröbner bases).[41] As we understand it, this is done by a process of expression rewriting, rather like term rewriting or rewrite-based theorem proving techniques.

Overall memory usage tends upward in a general ramp shape, but with minor short-term variations, especially small plateaus, while the profiles for usage of different-sized objects are roughly similar, their ramps start at different points during execution and have different slopes and irregularities—the proportions of different-sized objects vary somewhat.[42]

Hypercube. Figure 3 shows memory usage for a hypercube message-passing simulator, written by Don Lindsay while at CMU. It exhibits a large and simple plateau.

This program allocates a single very large object near the beginning of execution, which lives for almost the entire run; it represents the nodes in a hypercube and their interconnections.[43] A very large number of other objects are created, but they are small and very short-lived; they represent messages

[37] See the discussion of [Han90] (Section 4.1) for a description of obstacks.

[38] We've seen similarly strong peaks in a profile of a compiler of our own, which relies on garbage collection rather than obstacks.

[39] Interestingly, the first of the horns usually consists of a size that is specific to that peak—different peaks use *different-sized* large objects, but the out-of-phase partner horn consists of the same small size each time. The differences in sizes used by the first horn explains why only three of these horns show up in the plot, and they show up for the largest peaks—for the other peaks' large sizes, the total memory used does not make it into the top five.

[40] This program (and the hypercube simulator described below) were also used by Detlefs in [Det92] for evaluation of a garbage collector. Based on several kinds of profiles, we now think that Detlefs' choice of test programs may have led to an overestimation of the costs of his garbage collector for C++. Neither of these programs is very friendly to a simple GC, especially one without compiler or OS support.

[41] The function of this program is rather analogous to that of a Fourier transform, but the basis functions are polynomials rather than sines and cosines, and the mechanism used is quite different.

[42] Many of the small irregularities in overall usage come from sizes that don't make it into the top five—small but highly variable numbers of these objects are used.

[43] In these plots, "time" advances at the end of each allocation. This accounts for the horizontal segments visible after the allocatons of large objects—no other objects are allocated or deallocated between the beginning and end of the allocation of an individual object, and allocation time advances by the size of the object.

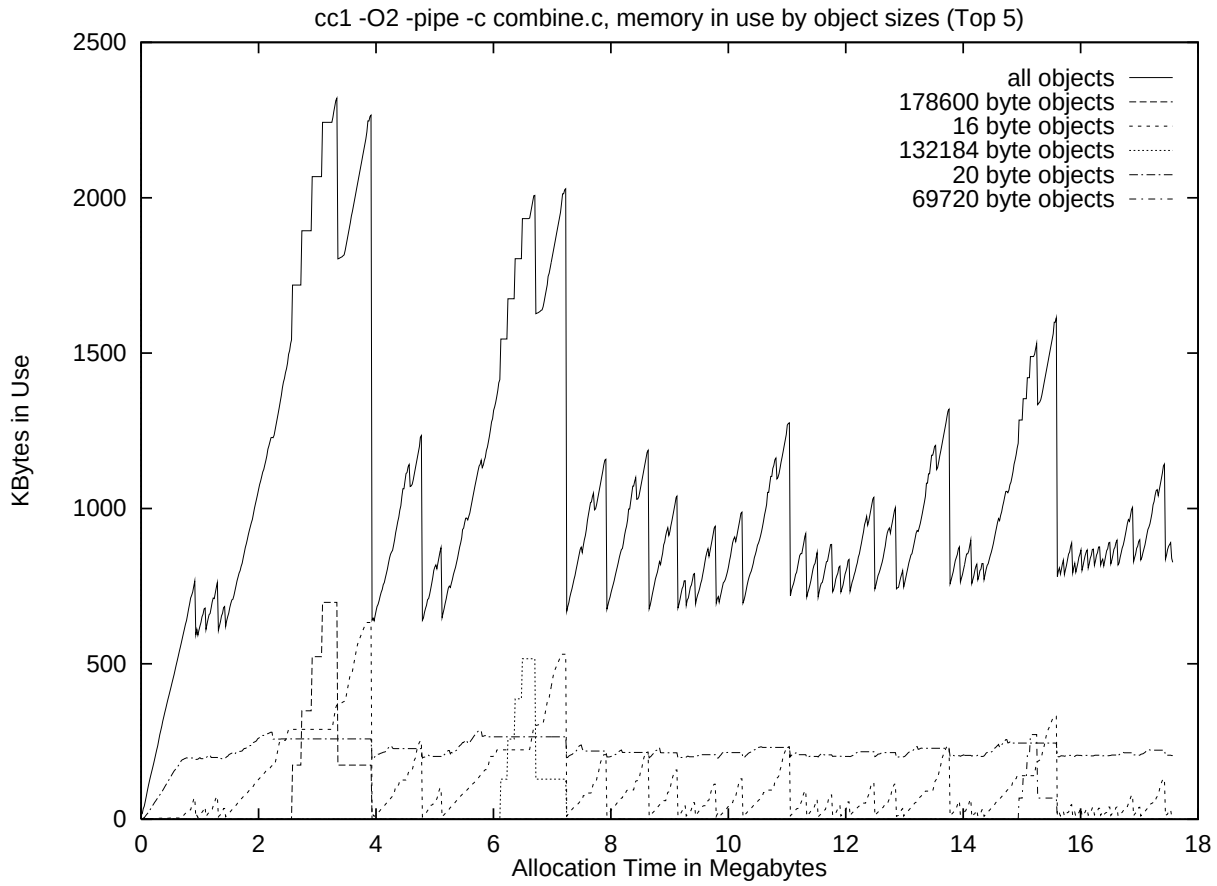

Fig. 1. Profile of memory usage in the GNU C compiler.

sent between nodes randomly.[44] This program quickly reaches a steady state, but the steady state is quite different from the one reached by most randomized allocator simulations—a very few sizes are represented, and lifetimes are both extremely skewed and strongly correlated with sizes.

Perl. Figure 4 shows memory usage for a script (program) written in the Perl scripting language. This program processes a file of string data. (We're not sure exactly what it is doing with the strings, to be honest; we do not really understand this program.) This program reaches a steady state, with heavily skewed usage of different sizes in relatively fixed proportions. (Since Perl is a fairly general and featureful programming language, its memory usage may vary tremendously depending on the program being executed.)

LRUsim. Figure 5 shows memory usage for a locality profiler written by Doug van Wieren. This program processes a memory reference trace, keeping track of how recently each block of memory has been touched and a accumulating a histogram of hits to blocks at different recencies (LRU queue positions). At the end of a run, a PostScript grayscale plot of the time-varying locality characteristics is generated. The recency queue is represented as a large modified AVL tree, which dominates memory usage—only a single object size really matters much. At the parameter setting used for this run, no blocks are ever discarded, and the tree grows monotonically; essentially no heap-allocated objects are ever freed, so memory usage is a

[44] These objects account for the slight increase and irregularitiy in the overall lifetime curve at around 2MB, after the large, long-lived objects have been allocated.

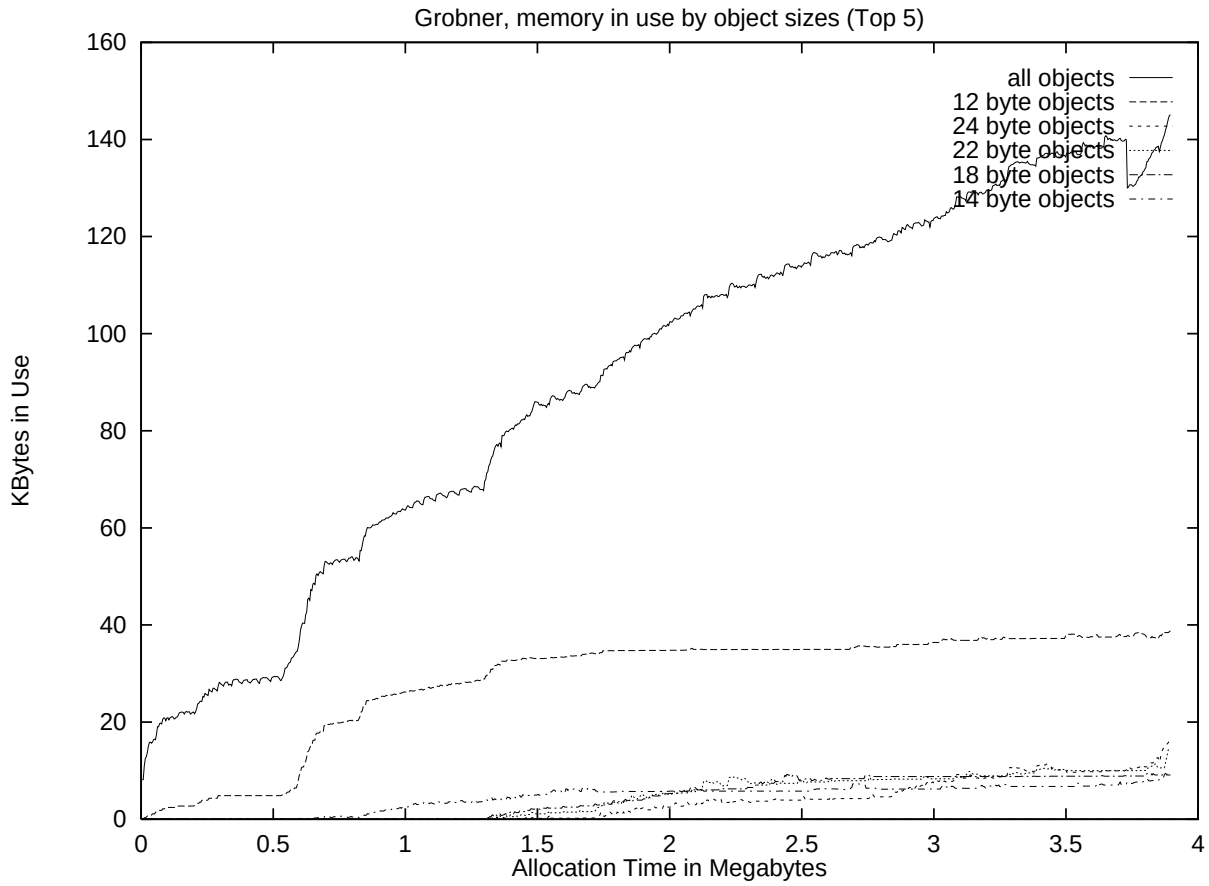

Fig. 2. Profile of memory usage in the Grobner program.

simple ramp. At other settings, only a bounded number of items are kept in the LRU tree, so that memory usage ramps up to a very stable plateau. This program exhibits a kind of dynamic stability, either by steady accumulation (as shown) or by exactly replacing the least-recently-used objects within a plateau (when used with a fixed queue length).

This is a small and simple program, but a very real one, in the sense that we have used it to tie up many megabytes of memory for about a trillion instruction cycles.[45]

[45] We suspect that in computing generally, a large fraction of CPU time and memory usage is devoted to programs with more complex behavior, but another significant fraction is dominated by highly regular behavior of simple useful programs, or by long, regular phases of more complex programs.

Espresso. Figure 6 shows memory usage for a run of Espresso, an optimizer for programmable logic array designs.

Espresso appears to go through several qualitatively different kinds of phases, using different sizes of objects in quite different ways.

Discussion of program profiles. In real programs, memory usage is usually quite different from the memory usage of randomized traces. Ramps, peaks, and plateaus are common, as is heavily skewed usage of a few sizes. Memory usage is neither Markov nor interestingly fractal-like in most cases. Many programs exhibit large-scale and small-scale patterns which may be of any of the common feature types, and different at different scales. Usage of different sizes may be strongly correlated, or it may not be, or may be

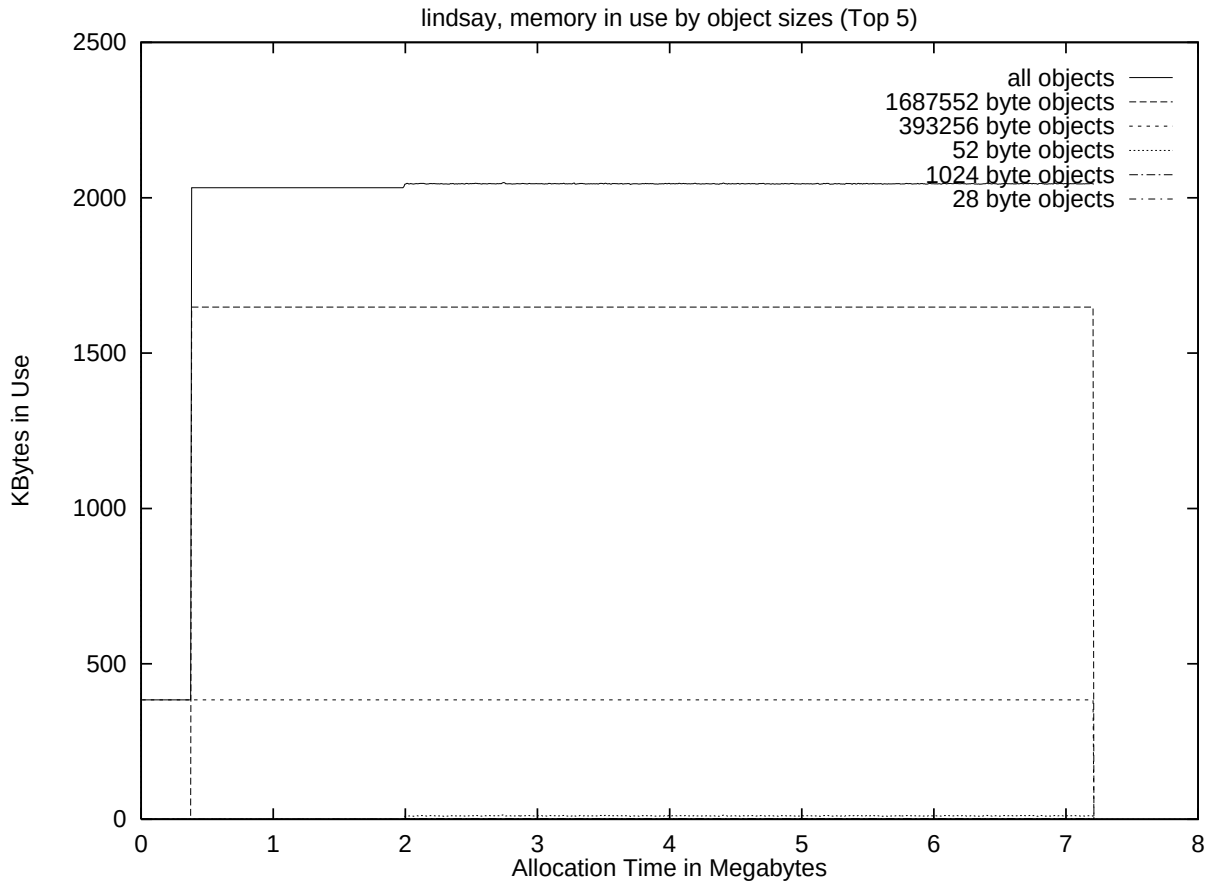

Fig. 3. Profile of memory usage in Lindsay's hypercube simulator.

related in more subtle time-varying ways. Given the wide variation within this small sample, it is clear that more programs should be profiled to determine which other patterns occur in a significant number of programs, and how often various patterns are likely to occur.

Summary. In summary, this section makes six related points:

- Program behavior is usually time-varying, not steady,
- Peak memory usage is important; fragmentation at peaks is more important than at intervening points,
- Fragmentation is caused by time-varying behavior, especially peaks using different sizes of objects.
- Known program behavior invalidates previous experimental and analytical results,
- Nonrandom behavior of programs can be exploited, and
- Different programs may display characteristically different nonrandom behavior.

2.5 Deferred Coalescing and Deferred Reuse

Deferred coalescing. Many allocators attempt to avoid coalescing blocks of memory that may be repeatedly reused for short-lived objects of the same size. This *deferred coalescing* can be added to any allocator, and usually avoids coalescing blocks that will soon be split again to satisfy requests for small objects. Blocks of a given size may be stored on a simple free list, and reused without coalescing, splitting, or formatting (e.g., putting in headers and/or footers).

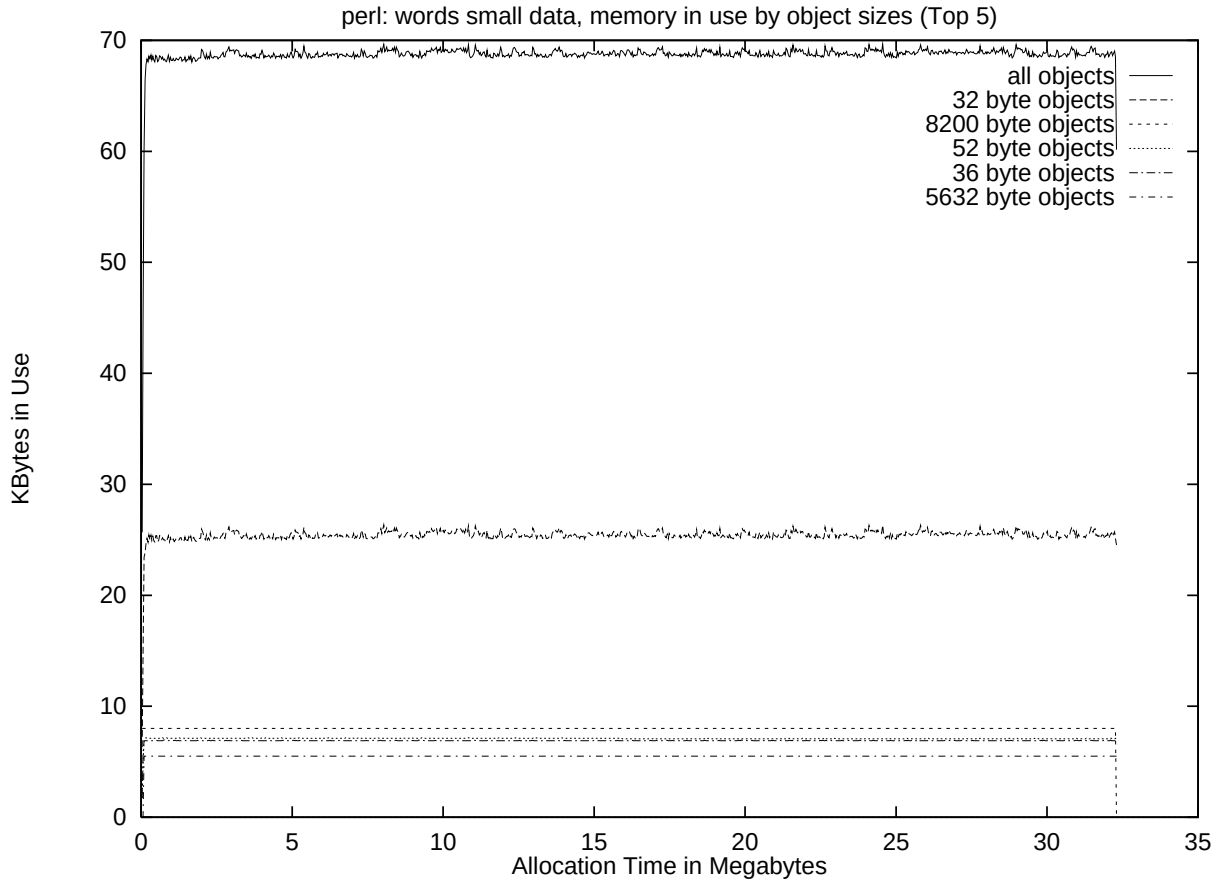

Fig. 4. Profile of memory usage in Perl running a string-processing script.

If the application requests the same size block soon after one is freed, the request can be satisfied by simply popping the pre-formatted block off of a free list in very small constant time.

While deferred coalescing is traditionally thought of as a speed optimization, it is important to note that fragmentation considerations come into play, in three ways.[46]

- The lower fragmentation is, the more important deferred coalescing will be in terms of speed—if adjacent objects generally die at about the same time, aggressive coalescing and splitting will be particularly expensive, because large areas will be coalesced together by repeatedly combining adjacent blocks, only to be split again into a large number of smaller blocks. If fragmentation is low, deferred coalescing may be especially beneficial.
- Deferred coalescing may have significant effects on fragmentation, by changing the allocator's decisions as to which blocks of memory to use to hold which objects. For example, blocks cannot be used to satisfy requests for larger objects while they remain uncoalesced. Those larger objects may therefore be allocated in different places than they would have been if small blocks were coalesced immediately; that is, deferred coalescing can affect placement policy.
- Deferred coalescing may decrease locality of reference for the same reason, because recently-freed small blocks will usually not be reused to

[46] To our knowledge, none of these effects has been noted previously in the literature, although it's likely we've seen at least the first but forgotten where. In any event, these effects have received little attention, and don't seem to have been studied directly.

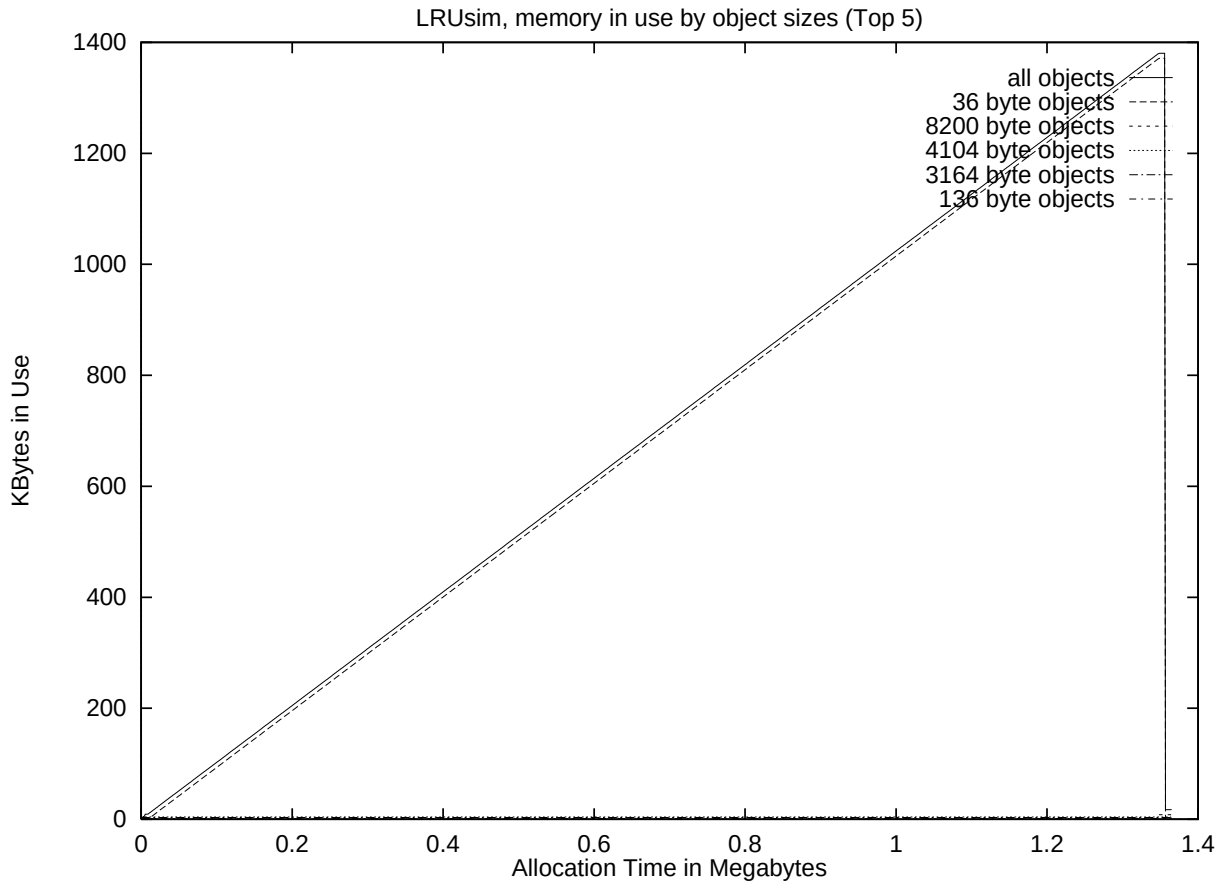

Fig. 5. Profile of memory usage in van Wieren's locality profiler.

hold larger objects. This may force the program to touch more different areas of memory than if small blocks were coalesced immediately and quickly used again. On the other hand, deferred coalescing is very likely to *increase* locality of reference if used with an allocator that otherwise would not reuse most memory immediately—the deferred coalescing mechanism will ensure that most freed blocks are reused soon.

Deferred reuse. Another related notion—which is equally poorly understood—is *deferred reuse*.[47] Deferred reuse is a property of some allocators that recently-freed blocks tend not to be the soonest reused. For many allocators, free memory is managed in a mostly stack-like way. For others, it is more queue-like, with older free blocks tending to be reused in preference to newly-freed blocks.

Deferred reuse may have effects on locality, because the allocator's choices affect which parts of memory are used by the program—the program will tend to use memory briefly, and then use *other* memory before reusing that memory.

Deferred reuse may also have effects on fragmentation, because newly-allocated objects will be placed in holes left by old objects that have died. This may make fragmentation worse, by mixing objects created by different phases (which may die at different times) in the same area of memory. On the other hand, it may be very beneficial because it may gradually pack the "older" areas of memory with long-lived objects, or because it gives the neighbors of a freed block more time to die before the freed block is reused. That

[47] Because it is not generally discussed in any systematic way in the literature, we coined this term for this paper.

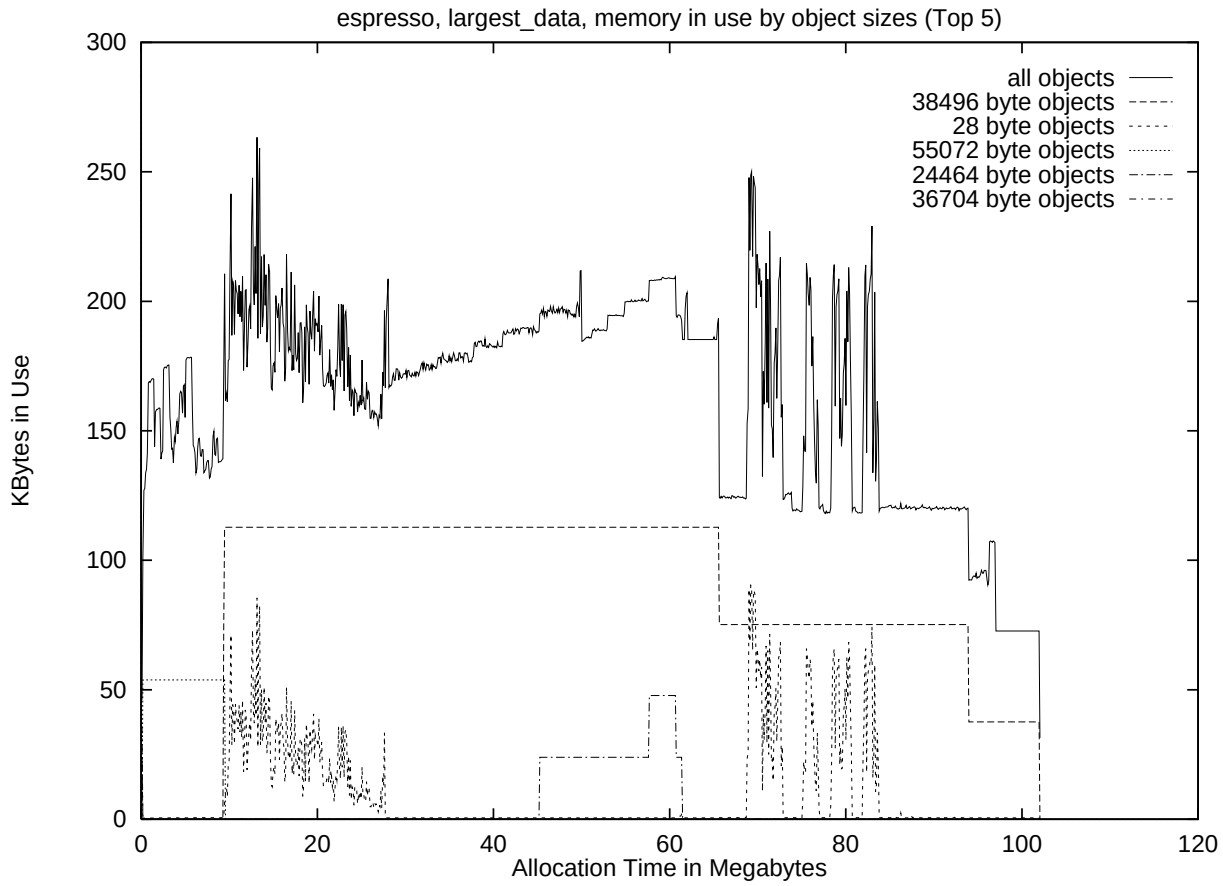

Fig. 6. Profile of memory usage in the Espresso PLA Optimizer.

may allow slightly longer-lived objects to avoid causing much fragmentation, because they will die relatively soon, and be coalesced with their neighbors whose reuse was deferred.

2.6 A Sound Methodology: Simulation Using Real Traces

The traditional view has been that programs' fragmentation-causing behavior is determined only by their object size and lifetime distributions. Recent experimental results show that this is false ([ZG94, WJNB95], Section 4.2), because orderings of requests have a large effect on fragmentation. Until a much deeper understanding of program behavior is available, and until allocator strategies and policies are as well understood as allocator mechanisms, the only reliable method for allocator simulation is to use real traces—i.e., the actual record of allocation and deallocation requests from real programs.

Tracing and simulation. Allocation traces are not particularly difficult to obtain (but see the caveats about program selection in Section 5.5). A slightly modified allocator can be used, which writes information about each allocation and deallocation request to a file—i.e., whether the request is an allocation or deallocation, the address of the block, and (for allocations) the requested block size. This allocator can be linked with a program of interest and used when running the program. These traces tend to be long, but they can be stored in compressed form, on inexpensive serial media (e.g., magnetic tape), and later processed serially during simulation. (Allocation traces are generally very compressible, due to the strong regularities

in program behavior.[48]) Large amounts of disk space and/or main memory are not required, although they are certainly convenient.

To use the trace for a simulation, a driver routine reads request records out of the file, and submits them to the allocator being tested by calling the allocator in the usual way. The driver maintains a table of objects that are currently allocated, which maps the object identifier from the trace file to the address where it is allocated during simulation; this allows it to request the deallocation of the block when it encounters the deallocation record in the trace.

This simulated program doesn't actually do anything with the allocated blocks, as a real program would, but it imitates the real program's request sequences exactly, which is sufficient for measuring the memory usage. Modern profiling tools [BL92, CK93] can also be used with the simulation program to determine how many instruction cycles are spent in the allocator itself.

An alternative strategy is to actually link the program with a variety of allocators, and actually re-run the program for each "simulation". This has the advantage that the traces needn't be stored. It has the disadvantages that it requires being able to re-run the program at will (which may depend on having similar systems, input data sets being available and in the right directories, environment variables, etc.) and doesn't allow convenient sharing of traces between different experimenters for replication of experiments. It also has the obvious disadvantage that instructions spent executing the actual program are wasted, but on fast machines this may be preferable to the cost of trace I/O, for many programs.

Locality studies. While locality is mostly beyond the scope of this paper, it is worth making a few comments about locality studies. Several tools are available to make it relatively easy to gather memory-reference traces, and several cache and virtual memory simulators are available for processing these traces.

Larus' QPT tool (a successor to the earlier AE system [BL92]) modifies an executable program to make it self-tracing. The Shade tool from SunLabs [CK93] is essentially a CPU emulator, which runs a program in emulation and records various kinds of events in an extremely flexible way. For good performance, it uses dynamic compilation techniques to increase speed relative to a straightford interpretive simulator.

Either of these systems can save a reference trace to a file, but the file is generally very large for long-running programs. Another alternative is to perform incremental simulation, as the trace is recorded—event records are saved to a fairly small buffer, and batches of event records are passed to a cache simulator which consumes them on the fly.

Efficient cache simulators are available for processing reference traces, including Mark Hill's Tycho and Dinero systems [HS89].[49]

3 A Taxonomy of Allocators

Allocators are typically categorized by the *mechanisms* they use for recording which areas of memory are free, and for merging adjacent free blocks into

[48] Conventional text-string-oriented compression algorithms [Nel91] (e.g, UNIX `compress` or GNU `gzip`) work quite well, although we suspect that sophisticated schemes could do significantly better by taking advantage of the numerical properties of object identifiers or addresses; such schemes have been proposed for use in compressed paging and addressing [WLM91, FP91]. (Text-oriented compression generally makes Markov-like modeling assumptions, i.e., that literal sequences are likely to recur. This is clearly true to a large degree for allocation and reference traces, but other regularities could probably be exploited as well [WB95].)

Dain Samples [Sam89] used a simple and effective approach for compressing memory-reference traces; his "Mache" trace compactor used a simple preprocessor to massage the trace into a different format, making the the relevant regularities easier for standard string-oriented compression algorithms to recognize and exploit. A similarly simple system may work well for allocation traces.

[49] Before attempting locality studies, however, allocation researchers should become familiar with the rather subtle issues in cache design, in particular the effects and interactions of associativity, fetch and prefetch policies, write buffers, victim buffers, and subblock placement.

Such details have been shown to be important in assessing the impact of locality of allocation on performance; a program with apparently "poor" locality for a simple cache design may do quite well in a memory hierarchy well-suited to its behavior. The literature on garbage collection is considerably more sophisticated in terms of locality studies than the literature on memory allocation, and should not be overlooked. (See, e.g., [Bae73, KLS92, Wil90, WLM92, DTM93, Rei94, GA95, Wil95].) Many of the same issues must arise in conventionally-managed heaps as well.

larger free blocks (coalescing). Equally important are the *policy* and *strategy* implications—i.e., whether the allocator properly exploits the regularities in real request streams.

In this section, we survey the policy issues and mechanisms in memory allocation; since deferred coalescing can be added to any allocator, it will be discussed after the basic general allocator mechanisms have been covered, in Section 3.11.

3.1 Allocator Policy Issues

We believe that there are several important policy issues that must be made clear, and that real allocators' performance must be interpreted with regard to them:

- *Patterns of Memory Reuse.* Are recently-freed blocks reused in preference to older free areas? Are free blocks in an area of memory preferentially reused for objects of the same size (and perhaps type) as the live objects nearby? Are free blocks in some areas reused in preference to free blocks in other areas (e.g., preferentially reusing free blocks toward one end of the heap area)?
- *Splitting and Coalescing.* Are large free blocks split into smaller blocks to satisfy requests for smaller objects? Are adjacent free blocks merged into larger areas at all? Are all adjacent free areas coalesced, or are there restrictions on when coalescing can be done because it simplifies the implementation? Is coalescing always done when it's possible, or is it *deferred* to avoid needless merging and splitting over short periods of time?
- *Fits.* When a block of a particular size is reused, are blocks of about the same size used preferentially, or blocks of very different sizes? Or perhaps blocks whose sizes are related in some other useful way to the requested size?
- *Splitting thresholds.* When a too-large block is used to satisfy a request, is it split and the remainder made available for reuse? Or is the remainder left unallocated, causing *internal fragmentation*, either for implementation simplicity or as part of a policy intended to trade internal fragmentation for reduced external fragmentation?

All of these issues may affect overall fragmentation, and should be viewed as *policies*, even if the reason for a particular choice is to make the mechanism (implementation) simpler or faster. They may also have effects on locality; for example, reusing recently-freed blocks may increase temporal locality of reference by reusing memory that is still cached in high-speed memory, in preference to memory that has gone untouched for a longer while. (Locality is beyond the scope of this paper, but it is an important consideration. We believe that the best policies for reducing fragmentation are good for locality as well, by and large, but we will not make that argument in detail here.[50])

3.2 Some Important Low-Level Mechanisms

Several techniques are used in different combinations with a variety of allocators, and can help make sophisticated policies surprisingly easy to implement efficiently. We will describe some very low-level mechanisms that are pieces of several "basic" (higher-level) mechanisms, which in turn implement a policy.

(The casual reader may wish to skim this section.)

Header fields and alignment. Most allocators use a hidden "header" field within each block to store useful information. Most commonly, the size of the block is recorded in the header. This simplifies freeing, in many algorithms, because most standard allocator interfaces (e.g., the standard C `free()` routine) do not require a program to pass the size of the freed block to the deallocation routine at deallocation time.

Typically, the allocation function (e.g., C's `malloc()` memory allocation routine) passes only the requested size, and the allocator returns a pointer to the block allocated; the free routine is only passed that address, and it is up to the allocator to infer the size if necessary. (This may not be true in some systems with stronger type systems, where the sizes of objects are usually known statically. In that case, the compiler may generate code that supplies the object size to the freeing routine automatically.)

Other information may be stored in the header as well, such as information about whether the block is in use, its relationship to its neighbors, and so on. Having information about the block stored with the block makes many common operations fast.

[50] Briefly, we believe that the allocator should heuristically attempt to cluster objects that are likely to be used at about the same times and in similar ways. This should improve locality [Bae73, WLM91]; it should also increase the chances that adjacent objects will die at about the same time, reducing fragmentation.

Header fields are usually one machine word; on most modern machines, that is four 8-bit bytes, or 32 bits. (For convenience, we will assume that the word size is 32 bits, unless indicated otherwise.) In most situations, there is enough room in one machine word to store a size field plus two or three one-bit "flags" (boolean fields). This is because most systems allocate all heap-allocated objects on whole-word or doubleword address boundaries, but most hardware is byte-addressable.[51] (This constraint is usually imposed by compilers, because hardware issues make unaligned data slower—or even illegal—to operate on.)

This alignment means that partial words cannot be allocated—requests for non-integral numbers of words are rounded up to the nearest word. The rounding to word (or doubleword) boundaries ensures that the low two (or three) bits of a block address are always zero.

Header fields are convenient, but they consume space—e.g., a word per block. It is common for block sizes in many modern systems to average on the order of 10 words, give or take a factor of two or so, so a single word per header may increase memory usage by about 10% [BJW70, Ung86, ZG92, DDZ93, WJNB95].

Boundary tags. Many allocators that support general coalescing are implemented using *boundary tags* (due to Knuth [Knu73]) to support the coalescing of free areas. Each block of memory has a both header and a "footer" field, both of which record the size of the block and whether it is in use. (A footer, as the name suggests, is a hidden field within the block, at the opposite end from the header.) When a block is freed, the footer of the preceding block of memory is examined to see if it is free; likewise, the header of the following block is examined. Adjacent free areas are merged to form larger free blocks.

Header and footer overhead are likely to be significant—with an average object size of about ten words, for example, a one-word header incurs a 10% overhead and a one-word footer incurs another 10%.

Luckily there is a simple optimization that can avoid the footer overhead.[52] Notice that when an block is in use (holding a live object), the size field in the footer is not actually needed—all that is needed is the flag bit saying that the storage is unavailable for coalescing. The size field is only needed when the block is free, so that its header can be located for coalescing. The size field can therefore be taken out of the last word of the block of memory—when the block is allocated, it can be used to hold part of the object; when the object is freed, the size field can be copied from the header into the footer, because that space is no longer needed to hold part of the object.

The single bit needed to indicate whether a block is in use can be stolen from the *header word* of the *following block* without unduly limiting the range of the size field.[53]

Link fields within blocks. For allocators using free lists or indexing trees to keep track of free blocks, the list or tree nodes are generally embedded in the free blocks themselves. Since only free blocks are recorded, and since their space would otherwise be wasted, it is usually considered reasonable to use the space within the "empty" blocks to hold pointers linking them together. Space for indexing structures is therefore "free" (almost).

Many systems use doubly-linked linear lists, with a "previous" and "next" pointer taken out of the free area. This supports fast coalescing; when objects are merged together, at least one of them must be removed from the linked list so that the resulting block will appear only once in the list. Having pointers to both the predecessor and successor of a block makes it possible to quickly remove the block from the list, by adjusting those objects' "next" and "previous" pointers to skip the removed object.

Some other allocators use trees, with space for the "left child" and "right child" (and possibly "parent") pointers taken out of the free area.

The hidden cost of putting link fields within blocks is that the block must be big enough to hold them, along with the header field and footer field, if any. This imposes a *minimum block size* on the allocator imple-

[51] For doubleword aligned systems, it is still possible to use a one-word header while maintaining alignment. Blocks are allocated "off by one" from the doubleword boundary, so that the part of the block that actually stores an object is properly aligned.

[52] This optimization is described in [Sta80], but it appears not to have been noticed and exploited by most implementors of actual systems, or by researchers in recent years.

[53] Consider a 32-bit byte-addressed system where blocks may be up to 4GB. As long as blocks are word-aligned, the least significant bits of a block address are always zero, so those two "low bits" can be used to hold the two flags. In a doubleword-aligned system, three "low bits" are available.

mentation, and any smaller request must be rounded up to that size. A common situation is having a header with a size field and boundary tags, plus two pointers in each block. This means that the smallest block size must be at least three words. (For doubleword alignment, it must be four.)

Assuming only the header field is needed on allocated blocks, the effective object size is three words for one-, two-, or three-word objects. If many objects are only one or two words long—and two is fairly common—significant space may be wasted.

Lookup tables. Some allocators treat blocks within ranges of sizes similarly—rather than indexing free blocks by their exact size, they lump together blocks of roughly the same size. The size range may also be important to the coalescing mechanism. Powers of two are often used, because it is easy to use bit selection techniques on a binary representation of the size to figure out which power-of-two range it falls into. Powers of two are coarse, however, and can have drawbacks, which we'll discuss later.

Other functions (such as Fibonacci series) may be more useful, but they are more expensive to compute at run time. A simple and effective solution is to use a *lookup table*, which is simply an array, indexed by the size, whose values are the numbers of the ranges. To look up which range a size falls into, you simply index into the array and fetch the value stored there. This technique is simple and very fast.

If the values used to index into the table are potentially large, however, the lookup table itself may be too big. This is often avoided by using lookup tables only for values below some threshold (see below).

Special treatment of small objects. In most systems, many more small objects are allocated than large ones. It is therefore often worthwhile to treat small objects specially, in one sense or another. This can usually be done by having the allocator check to see if the size is small, and if so, use an optimized technique for small values; for large values, it may use a slower technique.

One application of this principle is to use a fast allocation technique for small objects, and a space-efficient technique for large ones. Another is to use fast lookup table techniques for small values, and slower computations for large ones, so that the lookup tables don't take up much space. In this case, consider the fact that it is very difficult for a program to use a large number of large objects in a short period of time—it generally must *do something* with the space it allocates, e.g., initialize the fields of the allocated objects, and presumably do something more with at least some of their values. For some moderate object size and above, the possible frequency of allocations is so low that a little extra overhead is not significant. (Counterexamples are possible, of course, but we believe they are rare.) The basic idea here is to ensure that the time spent allocating a block is small relative to the computations on the data it holds.

Special treatment of the end block of the heap. The allocator allocates memory to programs on request, but the allocator itself must get memory from somewhere. The most common situtation in modern systems is that the heap occupies a range of virtual addresses and grows "upward" through the address space. To request more (virtual) memory, a system call such as the UNIX `brk()`[54] call is used to request that storage be mapped to that region of address space, so that it can be used to hold data.[55] Typically, the allocator keeps a "high-water mark" that divides memory into the part that is backed by storage and the part that is not.

(In systems with a fixed memory, such as some non-virtual memory systems, many allocators maintain a similar high-water mark for their own purposes, to keep track of which part of memory is in use and which part is a large contiguous free space.)

We will generally assume that a paged virtual memory is in use. In that case, the system call that obtains more memory obtains some integral number of pages, (e.g., 4KB, 8KB, 12KB, or 16KB on a machine with 4KB pages.) If a larger block is requested, a larger request (for as many pages as necessary) is made.

Typically the allocator requests memory from the operating system when it cannot otherwise satisfy a memory request, but it actually only needs a small amount of memory to satisfy the request (e.g., 10 words). This raises the question of what is done with the rest of the memory returned by the operating system.

[54] `brk()` is often called indirectly, via the library routine `sbrk()`.

[55] Other arrangements are possible. For example, the heap could be backed by a (growable) memory-mapped file, or several files mapped to non-contiguous ranges of address space.

While this seems like a trivial bookkeeping matter, it appears that the treatment of this "end block" of memory may have significant policy consequences under some circumstances. (We will return to this issue in Section 3.5.)

3.3 Basic Mechanisms

We will now present a relatively conventional taxonomy of allocators, based mostly on mechanisms, but along the way we will point out policy issues, and alternative mechanisms that can implement similar policies. (We would prefer a strategy-based taxonomy, but strategy issues are so poorly understood that they would provide little structure. Our taxonomy is therefore roughly similar to some previous ones (particularly Standish's [Sta80]), but more complete.)

The basic allocator mechanisms we discuss are:

- *Sequential Fits*, including first fit, next fit, best fit, and worst fit,
- *Segregated Free Lists*, including simple segregated storage and segregated fits,
- *Buddy Systems*, including conventional binary, weighted, and Fibonacci buddies, and double buddies,
- *Indexed Fits*, which use structured indexes to implement a desired fit policy, and
- *Bitmapped Fits*, which are a particular kind of indexed fits.

The section on sequential fits, below, is particularly important—many basic policy issues arise there, and the policy discussion is applicable to many different mechanisms.

After describing these basic allocators, we will discuss deferred coalescing techniques applicable to all of them.

3.4 Sequential Fits

Several classic allocator algorithms are based on having a single linear list of all free blocks of memory. (The list is often doubly-linked and/or circularly-linked.) Typically, sequential fits algorithms use Knuth's *boundary tag* technique, and a doubly-linked list to make coalescing simple and fast.

In considering sequential fits, it is probably most important to keep strategy and policy issues in mind. The classic linear-list implementations may not *scale* well to large heaps, in terms of time costs; as the number of free blocks grows, the time to search the list may become unacceptable.[56] More efficient and scalable techniques are available, using totally or partially ordered trees, or segregated fits (see Section 3.6).[57]

Best fit. A best fit sequential fits 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. This strategy might backfire in practice, if the fits are *too* good, but not perfect—in that case, most of each block will be used, and the remainder will be quite small and perhaps unusable.[58]

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

Best fit generally exhibits quite good memory usage (in studies using both synthetic and real traces). Various scalable implementations have been built using balanced binary trees, self-adjusting trees, and segregated fits (discussed later).

The worst-case performance of best fit is poor, with its memory usage proportional to the product of the amount of allocated data and the ratio between the largest and smallest object size (i.e., Mn) [Rob77]. This appears not to happen in practice, or at least not commonly.

First fit. First fit simply searches the list from the beginning, and uses the first free block large enough to

[56] This is not *necessarily* true, of course, because the average search time may be much lower than the worst case. For robustly good performance, however, it appears that simple linear lists should generally be avoided for large heaps.

[57] The confusion of mechanism with strategy and policy has sometimes hampered experimental evaluations; even after obviously scalable implementations had been discussed in the literature, later researchers often excluded sequential fit policies from consideration due to their apparent time costs.

[58] This potential accumulation of small fragments (often called "splinters" or "sawdust") was noted by Knuth [Knu73], but it seems not to be a serious problem for best fit, with either real or synthetic workloads.

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 sequential first fit is that the larger blocks near the beginning of the list tend to be split first, and the remaining fragments result in having a lot of small blocks near the beginning of the list. These "splinters" can increase search times because many small free blocks accumulate, and the search must go past them each time a larger block is requested. Classic (linear) first fit therefore may scale poorly to systems in which many objects are allocated and many different-sized free blocks accumulate.

As with best fit, however, more scalable implementations of first fit are possible, using more sophisticated data structures. This is somewhat more difficult for first fit, however, because a first fit search must find the *first* block that is *also* large enough to hold the object being allocated. (These techniques will be discussed under the heading of Indexed Fits, in Section 3.8.)

This brings up an important policy question: what ordering is used so that the "first" fit can be found? When a block is freed, at what position is it inserted into the ordered set of free blocks? The most obvious ordering is probably to simply push the block onto the front of the free list. Recently-freed blocks would therefore be "first," and tend to be reused quickly, in LIFO (last-in-first-out) order. In that case, freeing is very fast but allocation requires a sequential search. Another possibility is to insert blocks in the list in address order, requiring list searches when blocks are freed, as well as when they are allocated.

An advantage of address-ordered first fit is that the address ordering encodes the adjacency of free blocks; this information can be used to support fast coalescing. No boundary tags or double linking (backpointers) are necessary. This can decrease the minimum object size relative to other schemes.[59]

In experiments with both real and synthetic traces, it appears that address-ordered first fit may cause significantly less fragmentation than LIFO-ordered first fit (e.g., [Wei76, WJNB95]); the address-ordered variant is the most studied, and apparently the most used.

Another alternative is to simply push freed blocks onto the *rear* of a (doubly-linked) list, opposite the end where searches begin. This results in a FIFO (first-in-first-out) queue-like pattern of memory use. This variant has not been considered in most studies, but recent results suggest that it can work quite well—better than the LIFO ordering, and perhaps as well as address ordering [WJNB95].

A first fit policy may tend over time toward behaving rather like best fit, because blocks near the front of the list are split preferentially, and this may result in a roughly size-sorted list.[60] Whether this happens for real workloads is unknown.

Next fit. A common "optimization" of first fit is to use a *roving pointer* for allocation [Knu73]. The pointer records the position where the last search was satisfied, and the next search begins from there. Successive searches cycle through the free list, so that searches do not always begin in the same place and result in an accumulation of splinters. The usual rationale for this is to decrease average search times when using a linear list, but this implementation technique has major effects on the policy (and effective strategy) for memory reuse.

Since the roving pointer cycles through memory regularly, objects from different phases of program execution may become interspersed in memory. This may affect fragmentation if objects from different phases have different expected lifetimes. (It may also seriously affect locality. The roving pointer itself may have bad locality characteristics, since it examines each free block before touching the same block again. Worse, it may affect the locality *of the program it allocates for*, by scattering objects used by certain phases and intermingling them with objects used by other phases.)

In several experiments using both real traces [WJNB95] and synthetic traces (e.g., [Bay77, Wei76, Pag84, KV85]), next fit has been shown to cause more

[59] Another possible implementation of address-ordered first fit is to use a linked list of *all* blocks, allocated or free, and use a size field in the header of each block as a "relative" pointer (offset) to the beginning of the next block. This avoids the need to store a separate link field, making the minimum object size quite small. (We've never seen this technique described, but would be surprised if it hasn't been used before, perhaps in some of the allocators described in [KV85].) If used straightforwardly, such a system is likely to scale *very* poorly, because live blocks must be traversed during search, but this technique might be useful in combination with some other indexing structure.

[60] This has also been observed by Ivor Page [Pag82] in randomized simulations, and similar (but possibly weaker) observations were made by Knuth and Shore and others in the late 1960's and 1970's. (Section 4.)

fragmentation than best fit or address-ordered first fit, and the LIFO-order variant may be significantly worse than address order [WJNB95].

As with the other sequential fits algorithms, scalable implementations of next fit are possible using various kinds of trees rather than linear lists.

3.5 Discussion of Sequential Fits and General Policy Issues.

The sequential fits algorithms have many possible variations, which raise policy issues relevant to most other kinds of allocators as well.

List order and policy. The classic first fit or next fit mechanisms may actually implement very different policies, depending on exactly how the free list is maintained. These policy issues are relevant to many other allocation mechanisms as well, but we will discuss them in the context of sequential fits for concreteness.

LIFO-ordered variants of first fit and next fit push freed blocks onto the front of the list, where they will be the next considered for reuse. (In the case of next fit, this immediate reuse only happens if the next allocation request can be satisfied by that block; otherwise the roving pointer will rove past it.)

If a FIFO-ordered free list is used, freed blocks may tend not to be reused for a long time. If an address-ordered free list is used, blocks toward one end of memory will tend to be used preferentially. Seemingly minor changes to a few of lines of code may change the placement policy dramatically, *and in effect implement a whole new strategy with respect to the regularities of the request stream.*

Address-ordered free lists may have an advantage in that they tend to pack one end of memory with live objects, and gradually move upward through the address space. In terms of clustering related objects, the effects of this strategy are potentially complex. If adjacent objects tend to die together, large contiguous areas of memory will come free, and later be carved up for consecutively-allocated objects. If deaths are scattered, however, scattered holes will be filled with related objects, perhaps decreasing the chances of contiguous areas coming free at about the same time. (Locality considerations are similarly complex.)

Even for best fit, the general strategy does not determine an exact policy. If there are multiple equally-good best fits, how is the tie broken? We do not know whether this choice actually occurs often in practice. It may be that large blocks tend to come free due to clustered deaths. If free blocks become scattered, however, it choosing among them may be particularly significant.

Splitting. A common variation is to impose a *splitting threshold*, so that blocks will not be split if they are already small. Blocks generally can't be split if the resulting remainder is smaller than the minimum block size (big enough to hold the header (and possibly a footer) plus the free list link(s)). In addition, the allocator may choose not to split a block if the remainder is "too small," either in absolute terms [Knu73] or relative to the size of the block being split [WJNB95].

This policy is intended to avoid allocating in the remainder a small object that may outlive the large object, and prevent the reclamation of a larger free area. Splitting thresholds do not appear to be helpful in practice, unless (perhaps) they are very small.

Splitting raises other policy questions; when a block is split, where is the remainder left in the free list? For address-ordered variants, there is no choice, but for others, there are several possibilities—leave it at the point in the list where the split block was found (this seems to be common), or put it on one end or the other of the free list, or anywhere in between.[61] And when the block is split, is the first part used, or the last, or even the middle?[62]

Other policies. Sequential fits techniques may also be used to intentionally implement unusual policies.

One policy is *worst fit*, where the largest free block is always used, in the hope that small fragments will not accumulate. The idea of worst fit is to avoid creating small, unusable fragments by making the remainder as *large* as possible. This extreme policy seems

[61] Our guess is that putting it at the head of the list would be advantageous, all other things being equal, to increase the chances that it would be used soon. This might tend to place related objects next to each other in memory, and decrease fragmentaton if they die at about the same time. On the other hand, if the remainder is too small and only reusable for a different size, this might make it likely to be used for a different purpose, and perhaps it should *not* be reused soon.

[62] Using the last part has the minor speed advantage that the first part can be left linked where it is in the free list—if that is the desired policy—rather than unlinking the first part and having to link the remainder back into the list.

to work quite badly (in synthetic trace studies, at least)—probably because of its tendency to ensure that there are *no* very large blocks available. The general idea may have some merit, however, as part of a combination of strategies.

Another policy is so-called "optimal fit," where a limited search of the list is usually used to "sample" the list, and a further search finds a fit that is as good or better [Cam71].[63]

Another policy is "half fit" [FP74], where the allocator preferentially splits blocks twice the requested size, in hopes that the remainder will come in handy if a similar request occurs soon.

Scalability. As mentioned before, the use of a sequentially-searched list poses potentially serious scalability problems—as heaps become large, the search times can in the worst case be proportional to the size of the heap. The use of balanced binary trees, self-adjusting ("splay") trees,[64] or partially ordered trees can reduce the worst-case performance so that it is logarithmic in the number of free blocks, rather than linear.[65]

Scalability is also sensitive to the degree of fragmentation. If there are many small fragments, the free list will be long and may take much longer to search.

Plausible pathologies. It may be worth noting that LIFO-ordered variants of first fit and next fit can suffer from severe fragmentation in the face of certain simple and plausible patterns of allocation and deallocation. The simplest of these is when a program repeatedly does the following:

1. allocates a (short-lived) large object,
2. allocates a long-lived small object, and
3. allocates another short-lived large object of the same size as the freed large object.

In this case, each time a large block is freed, a small block is soon taken out of it to satisfy the request for the small object. When the next large object is allocated, the block used for the previously-deallocated large object is now too small to hold it, and more memory must be requested from the operating system. The small objects therefore end up effectively wasting the space for large objects, and fragmentation is proportional to the ratio of their sizes. This may not be a common occurrence, but it has been observed to happen in practice more than once, with severe consequences.[66]

A more subtle possible problem with next fit is that clustered deallocations of different-sized objects may result in a free list that has runs of similar-sized blocks, i.e., batches of large blocks interspersed with batches of small blocks. The occasional allocation of a large object may often force the free pointer past many small blocks, so that subsequent allocations are more likely to carve small blocks out of large blocks. (This is a generalization of the simple kind of looping behavior that has been shown to be a problem for some programs.)

We do not yet know whether this particular kind of repetitive behavior accounts for much of the fragmentation seen for next fit in several experiments.

Treatment of the end block. As mentioned before, the treatment of the last block in the heap—at the point where more memory is obtained from the operating system, or from a preallocated pool—can be quite important. This block is usually rather large, and a mistake in managing it can be expensive. Since such blocks are allocated whenever heap memory grows, consistent mistakes could be disastrous [KV85]—all of the memory obtained by the allocator could get "messed up" soon after it comes under the allocator's control.

There is a philosophical question of whether the end block is "recently freed" or not. On the one hand, the block just became available, so perhaps it should be put on whichever end of the free list freed blocks are put on. On the other hand, it's *not* being *freed*—in a

[63] This is not really optimal in any useful sense, of course. See also Page's critique in [Pag82] (Section 4.1).

[64] Splay trees are particularly interesting for this application, since they have an adaptive characteristic that may adjust well to the patterns in allocator requests, as well as having amortized complexity within a constant factor of optimal [ST85].

[65] We suspect that earlier researchers often simply didn't worry about this because memory sizes were quite small (and block sizes were often rather large). Since this point was not generally made explicit, however, the obvious applicability of scalable data structures was simply left out of most discussions, and the confusion between policy and mechanism became entrenched.

[66] One example is in an early version of the large object manager for the Lucid Common Lisp system (Jon L. White, personal communication, 1991); another is mentioned in [KV85] (Section 4.1).

sense, the end block has been there all along, ignored until needed. Perhaps it should go on the opposite end of the list because it's conceptually the *oldest* block—the very large block that contains all as-yet-unused memory.

Such philosophical fine points aside, there is the practical question of how to treat a virgin block of significant size, to minimize fragmentation. (This block is sometimes called "wilderness" [Ste83] to signify that it is as yet unspoiled.)

Consider what happens if a first fit or next fit policy is being used. In that case, the allocator will most likely carve many small objects out of it immediately, greatly increasing the chances of being unable to recover the contiguous free memory of the block. On the other hand, putting it on the opposite end of the list will tend to leave it unused for at least a while, perhaps until it gets used for a larger block or blocks. An alternative strategy is to keep the wilderness block out of the main ordering data structure entirely, and only carve blocks out of it when no other space can be found. (This "wilderness" block can also be extended to include more memory by expanding the heap segment, so that the entire area above the high-water mark is viewed as a single huge block.[67]) Korn and Vo call this a "wilderness preservation heuristic," and report that it is helpful for some allocators [KV85] (No quantitative results are given, however.)

For policies like best fit and address-ordered first fit, it seems natural to simply put the end block in the indexing structure like any other block. If the end block is viewed as part of the (very large) block of as-yet-unused memory, this means that a best fit or address-ordered first fit policy will always use any other available memory before carving into the wilderness. If it is not viewed this way, the end block will usually be a little less than a page (or whatever unit is used to obtain memory from the operating system); typically, it will not be used to satisfy small requests unless there are no other similarly-large blocks available.

We therefore suspect—but do not know—that it does not matter much whether the block is viewed as the beginning of a huge block, or as a moderate-sized block in its own right, as long as the allocator tends to use smaller or lower-addressed blocks in preference to larger or higher-addressed blocks.[68]

Summary of policy issues. While best fit and address-ordered first fit seem to work well, it is not clear that other policies can't do quite as well; FIFO-ordered first fit may be about as good.

The sensitivity of such results to slight differences in mechanical details suggests that we do not have a good model of program behavior and allocator performance—at this point, it is quite unclear which seemingly small details will have significant policy consequences.

Few experiments have been performed with novel policies and real program behavior; research has largely focused on the obvious variations of algorithms that date from the early 1960's or before.[69]

Speculation on strategy issues. We have observed that best fit and address-ordered first fit perform quite similarly, for both real and synthetic traces.

Page [Pag82] has observed that (for random traces using uniform distributions), the short-term placement choices made by best fit and address-ordered

[67] In many simple UNIX and roughly UNIX-like systems, the allocator should be designed so that other routines can request pages from the operating system by extending the (single) "data segment" of the address space. In that case, the allocator must be designed to work with a potentially non-contiguous set of pages, because there may be intervening pages that belong to different routines. (For example, our Texas persistent store allows the data segment to contain interleaved pages belonging to a persistent heap and a transient heap [SKW92].)

Despite this possible interleaving of pages used by different modules, extending the heap will typically just extend the "wilderness block," because it's more likely that successive extensions of the data segment are due to requests by the allocator, than that memory requests from different sources are interleaved.

[68] It is interesting to note, however, that the *direction* of the address ordering matters for first fit, if the end block is viewed as the beginning of a very large block of all unused memory. If reverse-address-order is used, it becomes pathological. It will simply march through all of "available" memory—i.e., all memory obtainable from the operating system—without reusing any memory. This suggests to us that address-ordered first fit (using the usual preference order) is somehow more "right" than its opposite, at least in a context where the size of memory can be increased.

[69] Exceptions include Fenton and Payne's "half fit" policy (Section 4.1), and Beck's "age match" policy (Section 4.1). Barrett and Zorn's "lifetime prediction" allocator (Section 4.2) is the only recent work we know of (for conventional allocators) that adopts a novel and explicit strategy to exploit interesting regularities in real request streams.

first fit were usually identical. That is, if one of these policies was used up to a certain point in a trace, switching to the other for the next allocation request usually did not change the placement decision made for that request.

We speculate that this reflects a fundamental similarity between best fit and address-ordered first fit, in terms of how they exploit regularities in the request stream. These allocators seem to perform well—and *very similarly*—for both real and randomized workloads. In some sense, perhaps, each is an approximation of the other.

But a more important question is this: *what is the successful strategy that both of these policies implement?*

One possibility is something we might call the "open space preservation" heuristic, i.e., *try not to cut into relatively large unspoiled areas.*[70] At some level, of course, this is obvious—it's the same general idea that was behind best fit in the first place, over three decades ago.

As we mentioned earlier, however, there are at least two ideas behind best fit, at least in our view:

- *Minimize the remainder*—i.e., if a block must be split, split the block that will leave the smallest remainder. If the remainder goes unused, the smaller it is, the better.
- *Don't break up large free areas unnnecessarily*—preferentially split areas that are already small, and hence less likely to be flexibly usable in the future.

In some cases, the first principle may be more important, while the second may be more important in other cases. Minimizing the remainder may have a tendency to result in small blocks that are unlikely to be used soon; the *result* may be similar to having a splitting threshold, and to respect the second principle.[71]

These are very different strategies, at least on the surface. It's possible that these strategies can be combined in different ways—and perhaps they *are* combined in different ways by best fit and address-ordered first fit.

Shore [Sho75] designed and implemented a hybrid best fit/first fit policy that outperformed either plain first fit or plain best fit for his randomized workloads. (Discussed in Section 4.1.) The *strategic* implications of this hybrid policy have not been explored, and it is unclear whether they apply to real workloads. Shore's results should be interpreted with considerable caution, because real workloads exhibit regularities (e.g., plateaus and ramps) that seem likely to interact with these strategies in subtle ways.[72]

Address-ordered first fit seems likely to have other strategic implications as well. The use of address ordering seems likely to result in clustering of related data under some circumstances, increasing the chances that contiguous areas will come free, if the related objects die together. However, in cases where free blocks are small, of varied sizes, and widely scattered, first fit may tend to decluster related objects, as will best fit. Amending these policies may allow better clustering, which could be important for long-run fragmentation.

It should now be quite unclear why best fit and address-ordered first fit work well in practice, and whether they work *for the same reasons* under randomized workloads as for real workloads.

For randomized workloads, which cause more scattered random deaths, there may be very few placement choices, and little contiguous free memory. In that case, the strategy of minimizing the remainder may be crucial. For real workloads, however, large contiguous areas may come free at the ends of phases, and tend to be carved up into small blocks by later phases as live data accumulate. This may often result in contiguous allocation of successively-allocated blocks, which will again create large free blocks when they die together at the end of the later phase. In that case, the effects of small "errors" due to unusually long-lived objects may be important; they may lead to cumulative fragmentation for long-running programs, or fragmentation may stabilize after a while. We simply don't know.

There are many possible subtle interactions and strategic implications, all of which are quite poorly

[70] Korn and Vo's "wilderness preservation heuristic" can be seen as a special case or variant of the "open space preservation heuristic."

[71] This could explain why explicit splitting thresholds don't seem to be very helpful—policies like best fit may *already* implement a similar strategy indirectly, and adding an explicit splitting threshold may be overkill.

[72] For example, address-ordered first fit has a tendency to pack one end of memory with live data, and leave larger holes toward the other end. This seems particularly relevant to programs that allocate large and very long-lived data structures near the beginning of execution.

understood for these seemingly simple and very popular policies.

3.6 Segregated Free Lists

One of the simplest allocators uses an array of free lists, where each list holds free blocks of a particular size [Com64]. 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.

It is important to note that blocks in such schemes are *logically* segregated in terms of indexing, but usually *not* physically segregated in terms of storage. Many segregated free list allocators support general splitting and coalescing, and therefore must allow mixing of blocks of different sizes in the same area of memory.

One common variation is to use *size classes* to lump similar sizes together for indexing purposes, and use free blocks of a given size to satisfy a request for that size, or for any size that is slightly smaller (but still larger than any smaller size class). A common size-class scheme is to use sizes that are a power of two apart (e.g., 4 words, 8 words, 16 words...) and round the requested size up to the nearest size class; however, closer size class spacings have also been used, and are usually preferable.

Simple segregated storage. In this variant, no splitting of free blocks is done to satisfy requests for smaller sizes. When a request for a given size is serviced, and the free list for the appropriate size class is empty, more storage is requested from the underlying operating system (e.g., using UNIX `sbrk()` to extend the heap segment); typically one or two virtual memory pages are requested at a time, and split into same-sized blocks which are then strung together and put on the free list. We call this *simple segregated storage* because the result is that pages (or some other relatively large unit) contain blocks of only one size class. (This differs from the traditional terminology in an important way. "Segregated storage" is commonly used to refer both to this kind of scheme and what we call *segregated fits* [PSC71]. We believe this terminology has caused considerable confusion, and will generally avoid it; we will refer to the larger class as "segregated free list" schemes, or use the more specific terms "simple segregated storage" and "segregated fits."[73])

An advantage of this simple scheme is that no headers are required on allocated objects; the size information can be recorded for a page of objects, rather than for each object individually. This may be important if the average object size is very small. Recent studies indicate that in modern programs, the average object size *is* often quite small by earlier standards (e.g., around 10 words [WJNB95]), and that header and footer overheads alone can increase memory usage by ten percent or twenty percent [ZG92, WJNB95]. This is comparable to the "real" fragmentation for good allocators [WJNB95].

Simple segregated storage is quite fast in the usual case, especially when objects of a given size are repeatedly freed and reallocated over short periods of time. The freed blocks simply wait until the next allocation of the same size, and can be reallocated without splitting. Allocation and freeing are both fast constant-time operations.

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

There is some tradeoff between expected internal fragmentation and external fragmentation; if the spacing between size classes is large, more different sizes will fall into each size class, allowing space for some sizes to be reused for others. (In practice, very coarse size classes generally lose more memory to internal fragmentation than they save in external fragmentation.) In the worst case, memory usage is proportional to the product of the maximum amount of live data (plus worst-case internal fragmentation due to the rounding up of sizes) and the number of size classes.

A crude but possibly effective form of coalescing for

[73] Simple segregated storage is sometimes incorrectly called a buddy system; we do not use that terminology because simple segregated storage does not use a buddy rule for coalescing—no coalescing is done at all. (Standish [Sta80] refers to simple segregated storage as "partitioned storage.")

simple segregated storage (used by Mike Haertel in a fast allocator [GZH93, Vo95], and in several garbage collectors [Wil95]) is to maintain a count of live objects for each page, and notice when a page is entirely empty. If a page is empty, it can be made available for allocating objects in a different size class, preserving the invariant that all objects in a page are of a single size class.[74]

Segregated fits. This variant uses an array of free lists, with each array holding free blocks within a 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 fits search, and many significant variations are possible (see below). Typically first fit or next fit 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 sometimes *not* appreciated is that this also affects the placement in a very important way—the use of segregated lists excludes blocks of very different sizes, meaning *good* fits are usually found—the policy therefore embodies a *good fit* or even *best fit* strategy, despite the fact that it's often described as a variation on first fit.

If there is not a free block in the appropriate free list, segregated fits algorithms try to find a larger block and split it to satisfy the request. This usually proceeds by looking in the list for the next larger size class; if it is empty, the lists for larger and larger sizes are searched until a fit is found. If this search fails, more memory is obtained from the operating system to satisfy the request. For most systems using size classes, this is a logarithmic-time search in the worst case. (For example for powers-of-two size classes, the total number of lists is equal to the logarithm of the maximum block size. For a somewhat more refined series, it is still generally logarithmic, but with a larger constant factor.)

In terms of policy, this search order means that smaller blocks are used in preference to larger ones, as with best fit. In some cases, however, the details of the size class system and the searching of size-class lists may cause deviations from the best fit policy.

Note that in a segregated fits scheme, coalescing may increase search times. When blocks of a given size are freed, they may be coalesced and put on different free lists (for the resulting larger sizes); when the program requests more objects of that size, it may have to find the larger block and split it, rather than still having the same small blocks on the appropriate free list. (Deferred coalescing can reduce the extent of this problem, and the use of multiple free lists makes segregated fits a particularly natural context for deferred coalescing.)

Segregated fits schemes fall into three general categories:

1. *Exact Lists.* In exact lists systems, where there is (conceptually) a separate free list for each possible block size [Com64]. This can result in a very large number of free lists, but the "array" of free lists can be represented sparsely. Standish and Tadman's "Fast Fits" scheme[75] uses an array of free lists for small size classes, plus a binary tree of free lists for larger sizes (but only the ones that actually occur) [Sta80, Tad78].[76]
2. *Strict Size Classes with Rounding.* When sizes are grouped into size classes (e.g., powers of two), one approach is to maintain an invariant that all blocks on a size list are exactly of the same size. This can be done by rounding up requested sizes to one of the sizes in the size class series, at some cost in internal fragmentation. In this case, it is also necessary to ensure that the size class series is carefully designed so that split blocks always result in a size that is also in the series; otherwise blocks will result that aren't the right size for any free list. (This issue will be discussed in more detail when we come to buddy systems.)
3. *Size Classes with Range Lists.* The most common way of dealing with the ranges of sizes that

[74] This invariant can be useful in some kinds of systems, especially systems that provide persistence [SKW92] and/or garbage collection for languages such as C or C++ [BW88, WDH89, WJ93], where pointers may point into the interior parts of objects, and it is important to be able to find the object headers quickly. In garbage-collected systems, it is common to segregated objects by type, or by implementation-level characteristics, to facilitate optimizations of type checking and/or garbage collection [Yua90, Del92, DEB94].

[75] Not to be confused with Stephenson's better-known indexed fits scheme of the same name.

[76] As with most tree-based allocators, the nodes of the tree are embedded in the blocks themselves. The tree is only used for larger sizes, and the large blocks are big enough to hold left and right child pointers, as well as a doubly linked list pointers. One block of each large size is part of the tree, and it acts as the head of the doubly-linked list of same-sized blocks.

fall into size classes is to allow the lists to contain blocks of slightly different sizes, and search the size lists sequentially, using the classic best fit, first fit, or next fit technique [PSC71]. (The choice affects the policy implemented, of course, though probably much less than in the case of a single free list.) This could introduce a linear component to search times, though this does not seem likely to be a common problem in practice, at least if size classes are closely spaced.[77] [78] If it is, then exact list schemes are preferable.

An efficient segregated fits scheme with general coalescing (using boundary tags) was described and shown to perform well in 1971 [PSC71], but it did not become well-known; Standish and Tadman's apparently better scheme was published (but only in a textbook) in 1980, and similarly did not become particularly well known, even to the present. Our impression is that these techniques have received too little attention, while considerably more attention has been given to techniques that are inferior in terms of scalability (sequential fits) or generality (buddy systems).

Apparently, too few researchers realized the full significance of Knuth's invention of boundary tags for a wide variety of allocation schemes—boundary tags can support fast and general splitting and coalescing, independently of the basic indexing scheme used by the allocator. This frees the designer to use more sophisticated higher-level mechanisms and policies to implement almost any desired strategy. (It seems likely that the original version of boundary tags was initially viewed as too costly in space, in a time when memory was a very scarce resource, and the footer optimization [Sta80] simply never became well-known.)

3.7 Buddy Systems

Buddy systems [Kno65, PN77] are a variant of segregated lists that supports a limited but efficient kind of splitting and coalescing. In the simple buddy schemes, the entire heap area is conceptually split into two large areas, and those areas are further split into two smaller areas, and so on. This hierarchical division of memory is used to constrain where objects are allocated, what their allowable sizes are, and how they may be coalesced into larger free areas. For each allowable size, a separate free list is maintained, in an array of free lists. Buddy systems are therefore actually *a special case of segregated fits*, using size classes with rounding, and a peculiar limited technique for splitting and coalescing.

Buddy systems therefore implement an approximation of a best fit policy, but with potentially serious variations due to peculiarities in splitting and coalescing.

(In practical terms, buddy systems appear to be distinctly inferior to more general schemes supporting arbitrary coalescing; without heroic efforts at optimization and/or hybridization, their cost in internal fragmentation alone seems to be comparable to the total fragmentation costs of better schemes.)

A free block may only be merged with its *buddy*, which is its unique neighbor at the same level in the binary 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 may only be merged with the following block, which follows it in memory; conversely, the second block 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 splitting.

(This is perhaps best understood by example; the reader may wish to skip ahead to the description of binary buddies, which are the simplest kind of buddy systems.)

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 block, or an unavailable block. An unavailable block 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 beginning of the buddy—it will never find the middle of an allocated object. The buddy will be either a whole (allocated or free) block of a determinate size, or the beginning of a block of that size that has been split in a determinate way. If (and only if) it turns out to be the header of a free block, and the block is the whole buddy, the buddies

[77] Lea's allocator uses very closely spaced size classes, dividing powers of two linearly into four uniform ranges.

[78] Typical size distributions appear to be both spiky and heavily skewed, so it seems likely that for small size ranges, only zero or one actual sizes (or popular sizes) will fall into a given range. In that case, a segregated fits scheme may approximate a best fit scheme very closely.

can be merged. If the buddy is entirely or partly allocated, the buddies cannot be merged—even if there is an adjacent free area within the (split) buddy.

Buddy coalescing is relatively fast, but perhaps the biggest advantage in some contexts is that it requires little space overhead per object—only one bit is required per buddy, to indicate whether the buddy is a contiguous free area. This can be implemented with a single-bit header per object or free block. Unfortunately, for this to work, *the size of the block being freed must be known*—the buddy mechanism itself does not record the sizes of the blocks. This is workable in some statically-typed languages, where object sizes are known statically and the compiler can supply the size argument to the freeing routine. In most current languages and implementations, however, this is not the case, due to the presence of variable-sized objects and/or because of the way libraries are typically linked. Even in some languages where the sizes of objects are known, the "single" bit ends up up costing an entire word per object, because a single bit cannot be "stolen" from the space for an allocated object—objects must be aligned on word boundaries for architectural reasons, and there is no provision for stealing a bit from the space allocated to an object.[79] Stealing a bit from each object can be avoided, however, by keeping the bits in a separate table "off to the side" [IGK71], but this is fairly awkward, and such a bit table could probably be put to better use with an entirely different basic allocation mechanism.

In practical terms, therefore, buddy systems usually require a header word per object, to record the type and/or size. Other, less restrictive schemes can get by with a word per object as well. Since buddy systems also incur internal fragmentation, this apparently makes buddy systems unattractive relative to more general coalescing schemes such as segregated fits.[80]

In experiments using both real and synthetic traces, buddy systems generally exhibit significantly more fragmentation than segregated fits and indexed fits schemes using boundary tags to support general coalescing. (Most of these results come from synthetic trace studies, however; it appears that only two buddy systems have ever been studied using real traces [WJNB95].)

Several significant variations on buddy systems have been devised:

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

On the other hand, systems based on closer size class spacings may be similarly efficient if lookup tables are used to perform size class mappings quickly.

A major problem with binary buddies is that internal fragmentation is usually relatively high—the expected case is (very roughly) about 28% [Knu73, PN77],[81] because any object size must be rounded up to the nearest power of two (minus a word for the header, if the size field is stored).

Fibonacci buddies. This variant of the buddy scheme uses a more closely-spaced set of size classes, based on a Fibonacci series, to reduce internal fragmentation [Hir73]. Since each number in the Fibonacci series is the sum of the two previous numbers, a block can always be split (unevenly) to yield two blocks whose sizes are also in the series. As with binary buddies, the increasing size of successive size ranges limits the number of free lists required.

A further refinement, called *generalized Fibonacci buddies* [Hir73, Bur76, PN77] uses a Fibonacci-like number series that starts with a larger number and generates a somewhat more closely-spaced set of sizes.

A possible disadvantage of Fibonacci buddies is that when a block is split to satisfy a request for a particular size, the remaining block is of a *different*

[79] In some implementations of some languages, this is less of a problem, because all objects have headers that encode type information, and one bit can be reserved for use by the allocator and ignored by the language implementation. This complicates the language implementation, but may be worthwhile if a buddy system is used.

[80] Of course, buddy systems could become more attractive if it were to turn out that the buddy *policy* has significant beneficial interactions with actual program behavior, and unexpectedly reduced external fragmentation or increased locality. At present, this does not appear to be the case.

[81] This figure varies somewhat depending on the expected range and skew of the size distribution [PN77].

size, which is less likely to be useful if the program allocates many objects of the same size [Wis78].

Weighted buddies. Weighted buddy systems [SP74] use a different kind of size class series than either binary or Fibonacci buddy systems. Some size classes can be split only one way, while other size classes can be split in two ways. The size classes include the powers of two, but in between each pair of successive sizes, there is also a size that is three times a power of two. The series is thus 2, 3, 4, 6, 8, 12... (words). (Often, the series actually starts at 4 words.)

Sizes that are powers of two may only be split evenly in two, as in the binary buddy system. This always yields another size in the series, namely the next lower power of two.

Sizes that are three times a power of two can be split in two ways. They may be split evenly in two, yielding a size that is another three-times-a-power-of-two size. (E.g., a six may be split into two threes.) They may also be split unevenly into two sizes that are one third and two thirds of the original size; these sizes are always a power of two. (E.g., six may be split into two and four.).

Double buddies. Double buddy systems use a different technique to allow a closer spacing of size classes [Wis78, PH86, WJNB95]. They use two different binary buddy systems, with staggered sizes. For example, one buddy system may use powers-of-two sizes (2, 4, 8, 16...) while another uses a powers-of-two spacing starting at a different size, such as 3. (The resulting sizes are 3, 6, 12, 24 ...). This is the same set of sizes used in weighted buddies, but the splitting rule is quite different. Blocks may only be split in half, as in the binary buddy system, so the resulting blocks are always in the same binary buddy series.

Request sizes are rounded up to the nearest size class in either series. This reduces the internal fragmentation by about half, but means that space used for blocks in one size series can only coalesced or split into sizes in that series. That is, splitting a size whose place in the combined series is odd always produces another size whose place is odd; likewise, splitting an even-numbered size always produces an even-numbered size. (E.g., a block of size 16 can be split into 8's and 4's, and a block of size 24 can be split into 12's and 6's, but *not* 8's or 4's.)

This may cause external fragmentation if blocks in one size series are freed, and blocks in the other are requested. As an optimization, free areas of a relatively large size (e.g., a whole free page) may be made available to the other size series and split according to that size series' rules. (This complicates the treatment of large objects, which could be treated entirely differently, or by another buddy system for large units of free storage such as pages.)

Naturally, more than two buddy systems could be combined, to decrease internal fragmentation at a possible cost in external fragmentation due to limitations on sharing free memory between the different buddy systems.

As with simple segregated storage, it is possible to keep per-page counts of live objects, and notice when an entire page is empty. Empty pages can be transferred from one buddy series to another. To our knowledge, such an optimization has never been implemented for a double buddy scheme.

Buddy systems can easily be enhanced with deferred coalescing techniques, as in "recombination delaying" buddy systems [Kau84]. Another optimization is to tailor a buddy system's size class series to a particular program, picking a series that produces little internal fragmentation for the object sizes the program uses heavily.

3.8 Indexed Fits

As we saw in Section 3.4 simple linear list mechanisms can be used to implement a wide variety of policies, with general coalescing.

An alternative is to use a more sophisticated indexing data structure, which indexes blocks by exactly the characteristics of interest to the desired policy, and supports efficient searching according to those characteristics. We call this kind of mechanism *indexed fits*. (This is really an unsatisfying catch-all category, showing the limitations of a mechanism-based taxonomy.)

The simplest example of an indexed fit scheme was mentioned earlier, in the discussion of sequential fits: a best fit policy implemented using a balanced or self-adjusting binary tree ordered by block size. (Best fit policies may be easier to implement scalably than address-ordered first fit policies.)

Another example was mentioned in the section on segregated free lists (3.6); Standish and Tadman's exact lists scheme is the limiting case of a segregated fits scheme, where the indexing is precise enough that no linear searching is needed to find a fit. On the other

hand, it is also a straightforward two-step optimization of the simple balanced-tree best fit. (The first optimization is to keep a tree with only one node per size that occurs, and hang the extra blocks of the same sizes off of those nodes in linear lists. The second optimization is to keep the most common size values in an array rather than the tree itself.) Our mechanism-based taxonomy is clearly showing it seams here, because the use of hybrid data structures blurs the distinctions between the basic classes of allocators.

The best-known example of an indexed fits scheme is probably Stephenson's "Fast Fits" allocator [Ste83], which uses a Cartesian tree sorted on both size and address. A Cartesian tree [Vui80] encodes two-dimensional information in a binary tree, using two constraints on the tree shape. It is effectively sorted on a primary key and a secondary key. The tree is a normal totally-ordered tree with respect to the primary key. With respect to the secondary key, it is a "heap" data structure, i.e., a partially ordered tree whose nodes each have a value greater than their descendants. This dual constraint limits the ability to rebalance the tree, because the shape of the tree is highly constrained by the dual indexing keys.

In Stephenson's system, this indexing data structure is embedded in the free blocks of memory themselves, i.e., the blocks become the tree nodes in much the same way that free blocks become list nodes in a sequential fits fits scheme. The addresses of blocks are used as the primary key, and the sizes of blocks are used as the secondary key.

Stephenson uses this structure to implement either an address-ordered first fit policy (called "leftmost fit") or a "better fit" policy, which is intended to approximate best fit. (It is unclear how good an approximation this is.)

As with address-ordered linear lists, the address ordering of free blocks is encoded directly in the tree structure, and the indexing structure can be used to find adjacent free areas for coalescing, with no additional overhead for boundary tags. In most situations, however, a size field is still required, so that blocks being freed can be inserted into the tree in the appropriate place.

While Cartesian trees give logarithmic expected search times for random inputs, they may become unbalanced in the face of patterned inputs, and in the worst case provide only linear time searches.[82]

Discussion of indexed fits. In terms of implementation, it appears that size-based policies may be easier to implement efficiently than address-based policies; a tree that totally orders all actual block sizes will typically be fairly small, and quick to search. If a FIFO- or LIFO- ordering of same-sized blocks implements an acceptable policy, then a linear list can be used and no searching among same-sized blocks is required.[83] Size-based policies also easier to optimize the common case, namely small sizes.

A tree that totally orders all block addresses may be very much larger, and searches will take more time. On the other hand, adaptive structures (e.g., splay trees) may make these searches fast in the common case, though this depends on subtleties of the request stream and the policy that are not currently understood.

Deferred coalescing may be able to reduce tree searches to the point where the differences in speed are not critical, making the fragmentation implications of the policy more important than minor differences in speed.

Totally ordered trees may not be necessary to implement the best policy, whatever that should turn out to be. Partial orders may work just as well, and lend themselves to very efficient and scalable implementations. At this point, the main problem does not seem to be time costs, but understanding what policy will yield the least fragmentation and the best locality.

Many other indexed fits policies and mechanisms are possible, using a variety of data structures to accelerate searches. One of these is a set of free lists segregated by size, as discussed earlier, and another is a simple bitmap, discussed next.

3.9 Bitmapped Fits

A particularly interesting form of indexed fits is *bitmapped fits*, where a *bitmap* is used to record which

[82] Data from [Zor93] suggest that actual performance is reasonable for real data, being among the faster algorithms used in that study, and having good memory usage. On the other hand, data from a different experiment [GZ93] show it being considerably slower than a set of allocators designed primarily for speed. Very recent data [Vo95] show it being somewhat slower than some other algorithms with similar memory usage, on average.

[83] If an algorithm relies on an awkward secondary key, e.g., best fit with address-ordered tie breaking, then it may not make much difference what the ordering function is—one total ordering of blocks is likely to cost about as much as another.

parts of the heap area are in use, and which parts are not. A bitmap is a simple vector of one-bit flags, with one bit corresponding to each word of the heap area. (We assume here that heap memory is allocated in word-aligned units that are multiples of one word. In some systems, double-word alignment is required for architectural reasons. In that case, the bitmap will include one bit for each double-word alignment boundary.)

To our knowledge, bitmapped allocation has never been used in a conventional allocator, but it is quite common in other contexts, particularly mark-sweep garbage collectors (notably the conservative collectors of Boehm, et al. from Xerox PARC [BW88, BDS91, DWH+90][84]) and file systems' disk block managers. We suspect that the main reason it has not been used for conventional memory allocation is that it is perceived as too slow.

We believe that bitmap operations can be made fast enough to use in allocators by the use of clever implementation techniques. For example, a bitmap can be quickly scanned a byte at a time using a 256-way lookup table to detect whether there are any runs of a desired length.[85]

If object sizes are small, bitmapped allocation may have a space advantage over systems that use whole-word headers. A bit per word of heap memory only incurs a 3% overhead, while for object sizes averaging 10 words, a header incurs a 10% overhead. In the most obvious scheme, two bitmaps are required (one to encode the boundaries of blocks, and another to encode whether blocks are in use), but we believe there are ways around that.[86]

Bitmapped allocators have two other advantages compared to conventional schemes. One is that they support searching the free memory indexed by address order, or localized searching, where the search may begin at a carefully-chosen address. (Address-ordered searches may result in less fragmentation than similar policies using some other orderings.) Another advantage is that bitmaps are "off to the side," i.e., not interleaved with the normal data storage area. This may be exploitable to improve the locality of searching itself, as opposed to traversing lists or trees embedded in the storage blocks themselves. (It may also reduce checkpointing costs in systems that checkpoint heap memory, by improving the locality of writes; freeing an object does not modify heap memory, only the bitmap.) Bitmapped techniques therefore deserve further consideration.

It may appear that bitmapped allocators are slow, because search times are linear, and to a first approximation this may be true. But notice that if a good heuristic is available to decide which area of the bitmap to search, searching is linear in the size of the area searched, rather than the number of free blocks. The cost of bitmapped allocation may then be proportional to the rate of allocation, rather than the number of free blocks, and may scale *better* than other indexing schemes. If the associated constants are low enough, bitmapped allocation may do quite well. It may also be valuable in conjunction with other indexing schemes.

3.10 Discussion of Basic Allocator Mechanisms.

By now it should be apparent that our conventional taxonomy is of only very limited utility, because the implementation focus obscures issues of policy. At a sufficiently high level of abstraction, *all* of these allocators are really "indexed" fits—they record which areas of memory are free in some kind of data structure—but they vary in terms of the policies they implement, how efficiently their mechanisms support the desired policy, and how flexible the mechanisms are in supporting policy variations. Even in its own mechanism-based terms, the taxonomy is collapsing under its own weight due to the use of hybrid algorithms that can be categorized in several ways.

[84] Actually, these systems use bitmaps to detect contiguous areas of free memory, but then accumulate free lists of the detected free blocks. The advantage of this is that a single scan through a region of the bitmap can find blocks of all sizes, and make them available for fast allocation by putting them on free lists for those sizes.

[85] This can be enhanced in several ways. One enhancement allows the fast detection of longer runs that cross 8-bit boundaries by using a different lookup tables to compute the number of leading and trailing zeroes, so that a count can be maintained of the number of zeroes seen so far. Another is to use redundant encoding of the size by having headers in large objects, obviating long scans when determining the size of a block being freed.

[86] It is increasingly common for allocators to ensure double-word alignment (even on 32-bit machines), padding requests as necessary, for architectural reasons. In that case, half as many bits are needed. There may also be clever encodings that can make some of the bits in a bitmap do double duty, especially if the minimum object size is more than two alignment units.

Simple segregated storage is simple and quite fast—allocation and deallocation usually take only a few instructions each—but lacks freedom to split and coalesce memory blocks to support later requests for different-sized objects. It is therefore subject to serious external fragmentation, as well as internal fragmentation, with some tradeoff between the two.

Buddy systems support fairly flexible splitting, but significantly restricted coalescing.

Sequential fits support flexible splitting and (with boundary tags) general coalescing, but cannot support most policies without major scalability concerns. (More precisely, the boundary tag implementation technique supports completely general coalescing, but the "index" is so simple that searches may be very expensive for some policies.)

This leaves us with the more general indexed storage techniques, which include tree-structured indexes, segregated fits using boundary tags, and bitmapped techniques using bitmaps for both boundary tags and indexing. All of these can be used to implement a variety of policies, including exact or approximate best fit. None of them require more space overhead per object than buddy systems, for typical conventional language systems, and all can be expected to have lower internal fragmentation.

In considering any indexing scheme, issues of strategy and policy should be considered carefully. Scalability is a significant concern for large systems, and may become increasingly important.

Constant factors should not be overlooked, however. Alignment and header and footer costs may be just as significant as actual fragmentation. Similarly, the speed of common operations is quite important, as well as scalability to large heaps. In the next section, we discuss techniques for increasing the speed of a variety of general allocators.

3.11 Quick Lists and Deferred Coalescing

Deferred coalescing can be used with any of the basic allocator mechanisms we have described. The most common way of doing this is to keep an array of free lists, often called "quick lists" or "subpools" [MPS71], one for each size of block whose coalescing is to be deferred. Usually, this array is only large enough to have a separate free list for each individual size up to some maximum, such as 10 or 32 words; only those sizes will be treated by deferred coalescing [Wei76]. Blocks larger than this maximum size are simply returned directly to the "general" allocator, of whatever type.

The following discussion describes what seems to be a typical (or at least reasonable) arrangement. (Some allocators differ in significant details, notably Lea's segregated fits scheme.)

To the general allocator, a block on a quick list appears to be allocated, i.e., uncoalescable. For example, if boundary tags are used for coalescing, the flag indicates that the block is allocated. The fact that the block is free is encoded only in its presence on the quick list.

When allocating a small block, the quick list for that size is consulted. If there is a free block of that size on the list, it is removed from the list and used. If not, the search may continue by looking in other quick lists for a larger-sized block that will do. If this fails, the general allocator is used, to allocate a block from the general pool. When freeing a small block, the block is simply added to the quick list for that size. Occasionally, the blocks in the quick lists are removed and added to the general pool using the general allocator to coalesce neighboring free blocks.

The quick lists therefore act as caches for the location and size information about free blocks for which coalescing has not been attempted, while the general allocator acts as a "backing store" for this information, and implements general coalescing. (Most often, the backing store has been managed using an unscalable algorithm such as address-ordered first fit using a linear list.) Using a scalable algorithm for the general allocator seems preferable.

Another alternative is to use an allocator which in its usual operation maintains a set of free lists for different sizes or size classes, and simply to defer the coalescing of the blocks on those lists. This may be a buddy system (as in [Kau84]) or a segregated lists allocator such as segregated fits.[87]

Some allocators, which we will call "simplified quick fit" allocators, are structured similarly but don't do any coalescing for the small blocks on the quick lists. In effect, they simply use a non-coalescing segregated lists allocator for small objects and an entirely different allocator for large ones. (Examples include Weinstock and Wulf's simplification of their own Quick Fit allocator [WW88], and an allocator developed by Grunwald and Zorn, using Lea's allocator as the general allocator[GZH93].) One of the advantages of such

[87] The only deferred coalescing segregated fits algorithm that we know of is Doug Lea's allocator, distributed freely and used in several recent studies (e.g., [GZH93, Vo95, WJNB95]).

a scheme is that the minimum block size can be very small—only big enough to hold a header and and a single link pointer. (Doubly-linked lists aren't necessary, since no coalescing is done for small objects.)

These simplified designs are not true deferred coalescing allocators, except in a degenerate sense. (With respect to small objects, they are non-coalescing allocators, like simple segregated storage.)

True deferred coalescing schemes vary in significant ways besides what general allocator is used, notably in how often they coalesce items from quick lists, and which items are chosen for coalescing. They also may differ in the order in which they allocate items from the quick lists, e.g., LIFO or FIFO, and this may have a significant effect on placement policies.

Scheduling of coalescing. Some allocators defer all coalescing until memory runs out, and then coalesce all coalescable memory. This is most common in early designs, including Comfort's original proposal [Com64][88] and Weinstock's "Quick Fit" scheme [Wei76].

This is not an attractive strategy in most modern systems, however, because in a virtual memory, the program never "runs out of space" until backing store is exhausted. If too much memory remains uncoalesced, wasting virtual memory, locality may be degraded and extra paging could result. Most systems therefore attempt to limit the amount of memory that may be wasted because coalescing has not been attempted.

Some systems wait until a request cannot be satisfied without either coalescing or requesting more memory from the operating system. They then perform some coalescing. They may perform all possible coalescing at that time, or just enough to satisfy that request, or some intermediate amount.

Another possibility is to periodically flush the quick lists, returning all of the items on the quick lists to the general store for coalescing. This may be done incrementally, removing only the older items from the quick lists.

In Margolin et al.'s scheme [MPS71], the lengths of the free lists are bounded, and those lengths are based on the expected usage of different sizes. This ensures that only a bounded amount of memory can be wasted due to deferred coalescing, but if the estimates of usage are wrong, deferred coalescing may not work as well—memory may sit idle on some quick lists when it could otherwise be used for other sizes.

In Oldehoeft and Allan's system [OA85], the number of quick lists varies over time, according to a FIFO or Working Set policy. This has an adaptive character, especially for the Working Set policy, in that sizes that have not been freed recently are quickly coalesced, while "active" sizes are not. This adaptation may not be sufficient to ensure that the memory lost to deferred coalescing remains small, however; if the system only frees blocks of a few sizes over a long period of time, uncoalesced blocks may remain on another quick list indefinitely. (This appears to happen for some workloads in a similar system developed by Zorn and Grunwald [ZG94], using a fixed-length LRU queue of quick lists.)

Doug Lea's segregated fits allocator uses an unusual and rather complex policy to perform coalescing in small increments. (It is optimized as much for speed as for space.) Coalescing is only performed when a request cannot otherwise be satisfied without obtaining more memory from the operating system, and only enough coalescing is done to satisfy that request. This incremental coalescing cycles through the free lists for the different size classes. This ensures that coalescable blocks will not remain uncoalesced indefinitely, unless the heap is not growing.

In our view, the best policy for minimizing space usage without undue time costs is probably an adaptive one that limits the volume of uncoalesced blocks—i.e. the actual amount of potentially wasted space—and adapts the lengths of the free lists to the recent usage patterns of the program. Simply flushing the quick lists periodically (after a bounded amount of allocation) may be sufficient, and may not incur undue costs if the general allocator is reasonably fast.[89] [90]

[88] In Comfort's proposed scheme, there was no mechanism for immediate coalescing. (Boundary tags had not been invented.) The only way memory could be coalesced was by examining all of the free lists, and this was considered a awkward and expensive.

[89] The issues here are rather analogous to some issues in the design and tuning of generational garbage collectors, particularly the setting of generation sizes and advancement thresholds [Wil95].

[90] If absolute all-out speed is important, Lea's strategy of coalescing only when a search fails may be more attractive—it does not require incrementing or checking an allocation total at each allocation or deallocation. (Another possibility would be to use a timer interrupt, but this is quite awkward. Most allocator designers do not wish to depend on using interrupts for what is otherwise a fairly simple library, and it also raises ob-

On the other hand, it may be preferable to avoid attempting to coalesce very recently-freed blocks, which are very likely to be usable for another request soon. One possible technique is to use some kind of "high-water mark" pointer into each list to keep track of which objects were freed after some point in time, such as the last allocate/coalesce cycle. However, it may be easier to accomplish by keeping two lists, one for recently-freed blocks and one for older blocks. At each attempt at coalescing, the older blocks are given to the general allocator, and the younger blocks are promoted to "older" status.[91] (If a more refined notion of age is desired, more than two lists can be used.)

What to coalesce. As mentioned earlier, several systems defer the coalescing of small objects, but not large ones. If allocations of large objects are relatively infrequent—and they generally are—immediately coalescing them is likely to be worthwhile, all other things being equal. (This is true both because the time costs are low and the savings in potentially wasted memory are large.) Deferred coalescing usually affects the placement policy, however, and the effects of that interaction are not understood.

Discussion. There are many possible strategies for deferred coalescing, and any of them may affect the general allocator's placement policy and/or the locality of the program's references to objects. For example, it appears that for normal free lists, FIFO ordering may produce less fragmentation than LIFO ordering, but it is unknown whether that applies to items on quick lists in a deferred coalescing scheme.[92] Similarly, when items are removed from the quick list and returned to the general allocator, it is unknown which items should be returned, and which should be kept on the quick lists.

To date, only a few sound experiments evaluating deferred coalescing have been performed, and those that have been performed are rather limited in terms of identifying basic policy issues and the interactions between deferred coalescing and the general allocator.

Most experiments before 1992 used synthetic traces, and are of very dubious validity. To understand why, consider a quick list to be a buffer that absorbs variations in the number of blocks of a given size. If variations are small, most allocation requests can be satisfied from a small buffer. If there are frequent variations in the *sizes* in use, however, many buffers (quick lists) will be required in order to absorb them.

Randomization may reduce clustered usage of the same sizes, spreading all requested sizes out over the whole trace. This may make the system look bad, because it could increase the probability that the buffers (i.e., the set of quick lists) contain objects of the wrong sizes. On the other hand, the smoothed (random walk) nature of a synthetic trace may flatter deferred coalescing by ensuring that allocations and frees are fairly balanced over small periods of time; real phase behavior could overwhelm a too-small buffer by performing many frees and later many allocations.

3.12 A Note on Time Costs

An allocator can be made extremely fast if space costs are not a major issue. Simple segregated storage can be used to allow allocation or deallocation in a relatively small number of instructions—a few for a table lookup to find the right size class, a few for indexing into the free list array and checking to ensure the free list is not empty, and a few for the actual unlinking or linking of the allocated block.[93]

This scheme can be made cosiderably faster if the allocator can be compiled together with the applica-

scure issues of reentrancy—the interrupt handler must be careful not to do anything that would interfere with an allocation or deallocation that is interrupted.)

[91] This is similar to the "bucket brigade" advancement technique used in some generational garbage collectors [Sha88, WM89, Wil95]. A somewhat similar technique is used in Lea's allocator, but for a different purpose. Lea's allocator has a quick list (called the "dirty" list) for each size class used by the segregated fits mechanism, rather than for every small integer word size. (This means that allocations from the quick list have to search for a block that fits, but a close spacing of size classes ensures that there is usually only one popular size per list; the searches are usually short.) The quick lists are stored in the same array as the main ("clean") free lists.

[92] Informal experiments by Lea suggest that FIFO produces less fragmentation, at least for his scheme. (Lea, personal communication 1995.)

[93] For a closely-spaced series of size classes, it may be necessary to spend a few more instructions on checking the size to ensure that (in the usual case) it's small enough to use table lookup, and occasionally use a slower computation to find the appropriate list for large-sized requests.

tion program, rather than linked as a library in the usual way. The usual-case code for the allocator can be compiled as an "inline" procedure rather than a runtime procedure call, and compile-time analyses can perform the size-class determination at compile time. In the usual case, the runtime code will simply directly access the appropriate free list, check that it is not empty, and link or unlink a block. This inlined routine will incur no procedure call overhead. (This kind of alloction inlining is quite common in garbage collected systems. It can be a little tricky to code the inlined allocation routine so that a compiler will optimize it appropriately, but it is not too hard.)

If space is an issue, naturally things are more complicated—space efficient allocators are more complicated than simple segregated storage. However, deferred coalescing should ensure that a complex allocator behaves like simple segregated storage most of the time; with some space/time tradeoff. If extreme speed is desired, coalescing can be deferred for a longer period, to ensure that quick lists usually have free blocks on them and allocation is fast.[94] Adjusting this space-time tradeoff is a topic for future research, however.

4 A Chronological Review of The Literature

Given the background presented by earlier sections, we will chronologically review the literature, paying special attention to methodological considerations that we believe are important. To our knowledge, this is by far the most thorough review to date, but it should not be considered detailed or exhaustive; valuable points or papers may have escaped our notice.[95] We have left out work on concurrent and parallel allocators (e.g., [GW82, Sto82, BAO85, MK88, EO88, For88, Joh91, JS92, JS92, MS93, Iye93]), which are beyond the scope of this paper. We have also neglected mainly analytical work (e.g., [Kro73, Bet73, Ree79, Ree80, McI82, Ree82, BCW85]) to some degree, because we are not yet familiar enough with all of this literature to do it justice.

The two subsections below cover periods before and after 1991. The period from 1960 to 1990 was dominated by the gradual development of various allocator designs and by the synthetic trace methodology. The period after 1990 has (so far) shown that that methodology is in fact unsound and biased, and that much still needs to be done, both in terms of reevaluating old designs and inventing new ones on the basis of new results. (Readers who are uninterested in the history of allocator design and evaluation may wish to skip to Section 4.2.)

In much of the following, empirical results are presented qualitatively (e.g., allocator A was found to use space more efficiently than allocator B). In part, this is due to the fact that early results used figures of merit that are awkward to explain in a brief review, and difficult to relate to measures that current readers are likely to find most interesting. In addition, workloads have changed so much over the last three decades that precise numbers would be of mostly historical interest. (Early papers were mostly about managing operating system segments (or overlays) in fixed main memories,[96] while recent papers are mostly about managing small objects within the memory of a single process.) The qualitative presentation is also due in part to our skepticism of the methodology underlying most of the results before 1991; citing precise numbers would lend undue weight to quantities we consider questionable.

4.1 The first three decades: 1960 to 1990

Structure of this section. Our review of the work in this period is structured chronologically, and divided into three parts, roughly a decade each. Each of the three sections begins with an overview; the casual reader may want to read the overviews first, and skim the rest. We apologize in advance for a certain amount of redundancy—we have attempted to make this section relatively free-standing, so that it can be read straight through (by a reader with sufficient fortitude) given the basic concepts presented by earlier sections.

[94] This is not quite necessarily true. For applications that do little freeing, the initial carving of memory requested from the operating system will be a significant fraction of the allocation cost. This can be made quite fast as well, however.

[95] A few papers have not escaped our attention but seem to have escaped our libary. In particular, we have had to rely on secondary sources for Graham's influential work in worst-case analyses.

[96] Several very early papers (e.g., [Mah61, IJ62]) discussed memory fragmentation, but in systems where segments could be compacted together or swapped to secondary storage when fragmentation became a problem; these papers generally do not give any quantitative results at all, and few qualitative results comparing different allocation strategies.

1960 to 1969.

Overview. Most of the basic designs still in use were conceived in the 1960's, including sequential fits, buddy systems, simple segregated storage, and segregated lists using exact lists, and sequential fits. (Some of these, particularly sequential fits, already existed in the late 50's, but were not well described in the literature. Knuth [Knu73] gives pointers to early history of linked list processing.) In the earliest days, interest was largely in managing memory overlays or segments in segmented operating systems, i.e., managing mappings of logical (program and data) segments to physical memory.[97] By the mid-1960's, the problem of managing storage for different-sized objects within the address space of a single process was also recognized as an important one, largely due to the increasing use (and sophistication) of list processing techniques and languages [Ros61, Com64, BR64].[98]

Equally important, the 1960's saw the invention of the now-traditional methodology for allocator evaluation. In early papers, the assumptions underlying this scheme were explicit and warned against, but as the decade progressed, the warnings decreased in frequency and seriousness.

Some of the assumptions underlying this model made more sense then than they do now, at least for some purposes. For example, most computers were based on segmented memory systems, and highly loaded. In these systems, the memory utilization was often kept high, by long-term scheduling of jobs. (In some cases, segments belonging to a process might be evicted to backing storage to make room when a request couldn't otherwise be satisfied.) This makes steady-state and independence assumptions somewhat more plausible than in later decades, when the emphasis had shifted from managing segments in an operating system to managing individual program objects within the virtual memory of a single process.

On the other hand, in retrospect this assumption can be seen to be unwarranted even for such systems. For example, multitasking may *introduce* phase behavior, since the segments belonging to a process are usually only released when that process is running, or when it terminates. Between time slices, a program does not generally acquire or release segments. Operations on the segments associated with a process may occur periodically.

Other assumptions that became common during the 1960's (and well beyond) also seem unwarranted in retrospect. It was widely assumed that segment sizes were independent, perhaps because most systems were used by many users at the same time, so that most segments were typically "unrelated." On reflection, even in such a system there is good reason to think that particular segment sizes may be quite common, for at least three reasons. First, if the same program is run in different processes simultaneously, the statically-allocated data segment sizes of frequently-used programs may appear often. Second, some important programs may use data segments of particular characteristic sizes. (Consider a sort utility that uses a fixed amount of memory chosen to make internal sorting fast, but using merging from external storage to avoid bringing all of the data into memory.) Third, some segment sizes may be used in unusually large numbers due to peculiarities of the system design, e.g., the minimum and/or maximum segment size. (Segments or overlays were also typically fairly large compared to total memory, so statistical mechanics would not be particularly reliable even for random workloads.)

> The original paraphernalia for the lottery had been lost long ago, and the black box...had been put into use even before Old Man Warner, the oldest man in town, was born. Mr. Summers spoke frequently to the villagers about making a new box, but no one liked to upset even as much tradition as was represented by the black box. There was a story that the present box had been made with some pieces of the box that had preceded it, the one that had been constructed when the first people settled down to make a village here.
> —*Shirley Jackson*, "The Lottery"

Collins [Col61] apparently originated the random-trace methodology, and reported on experiments with best fit, worst fit, first fit, and random fit.

[97] This is something of an oversimplification, because in the earliest days operating systems were not well developed, and "user" programs often performed "system-level" tasks for themselves.

[98] Early list processing systems used only list nodes of one or two sizes, typically containing only two pointers, but later systems supported nodes of arbitrary sizes, to directly support structures that had multiple links. (Again, see Knuth [Knu73] for more references.)

Collins described his simulations as a "game," in the terminology of game theory. The application program and the allocator are players; the application makes moves by requesting memory allocations or deallocations, and the allocator responds with moves that are placement decisions.[99]

Collins noted that this methodology required further validation, and that experiments with real workloads would be better. Given this caveat, best fit worked best, but first fit (apparently address-ordered) was almost equally good. No quantitative results were reported, and the distributions used were not specified.

Comfort, in a paper about list processing for different-sized objects [Com64], briefly described the segregated lists technique with splitting and coalescing, as well as address-ordered first fit, using an ordered linear list.[100] (The address order would be used to support coalescing without any additional space overhead.) Comfort did not mention that his "multiple free lists" technique (segregated fits with exact lists) was an implementation of a best fit policy, or something very similar; later researchers would often overlook this scheme. Comfort also proposed a simple form of deferred coalescing, where no coalescing was done until memory was exhausted, and then it was all done at once. (Similar coalescing schemes seem to have been used in some early systems, with process swapping or segment eviction used when coalescing failed to obtain enough contiguous free memory.) No empirical results were reported.

Totschek [Tot65] reported the distribution of job sizes (i.e., memory associated with each process) in the SDC (Systems Development Corporation) timesharing system. Later papers refer to this as "the SDC distribution". Naturally, the "block" sizes here were rather large. Totschek found a roughly *trimodal* distribution, with most jobs being either around 20,000 words, or either less than half or more than twice that. He did not find a significant correlation between job size and running time.

Knowlton [Kno65] published the first paper on the (binary) buddy system, although Knuth [Knu73] reports that same idea was independently invented and used by H. Markowitz in the Simscript system around 1963. Knowlton also suggested the use of deferred coalescing to avoid unneeded overheads in the common case where objects of the same size were frequently used.

Ross, in [Ros67] described a sophisticated storage management system for the AED engineering design support system. While no empirical results were reported, Ross describes different patterns of memory usage that programs may exhibit, such as mostly monotonic accumulation (ramps), and fragmentation caused by different characteristic lifetimes of different-sized objects.

The storage allocation scheme divided available memory into "zones," which could be managed by different allocators suitable to different application's usual behavior.[101] Zones could be nested, and the system was extensible—a zone could use one of the default allocators, or provide its own allocation and deallocation routines. It was also possible to free an entire zone at once, rather than freeing each object individually. The default allocators included first fit and simple segregated storage. (This is the first published mention of simple segregated storage that we have found, though Comfort's multiple free list scheme is similar.)

Graham, in an unpublished technical report [Gra], described the problem of analyzing the worst-case memory use of allocators, and presented lower bounds on worst case fragmentation.[102] (An earlier memo by Doug McIlroy may have motivated this work, as well as Robson's later work.)

Graham characterized the problem metaphorically as a board game between an "attacker," who knows the exact policy used by the allocator ("defender") and submits requests ("makes moves") that will force the defender's policy to do as badly as possible. (This is a common metaphor in "minimax" game theory; such an omniscient, malevolent opponent is commonly called a "devil" or "evil demon.")

Knuth surveyed memory allocation techniques in Volume One of *The Art of Computer Programming*

[99] We suspect that the history of allocator research might have been quite different if this metaphor had been taken more seriously—the application program in the randomized methodology is a very unstable individual, or one using a very peculiar strategy.

[100] Knuth [Knu73] reports that this paper was written in 1961, but unpublished until 1964.

[101] Comparable schemes were apparently used in other early systems, including one that was integrated with overlaying in the IBM PL/I compiler [Boz84].

[102] We do not have a copy of this report at this writing. Our information comes from secondary sources.

([Knu73], first edition 1968), which has been a standard text and reference ever since. It has been particularly influential in the area of memory allocation, both for popularizing existing ideas and for presenting novel algorithms and analyses.

Knuth introduced next fit (called "modified first fit" in many subsequent papers), the boundary tag technique, and splitting thresholds. In an exercise, he suggested the Fibonacci buddy system (Ex. 2.5.31) In another exercise, he suggests using balanced binary trees for best fit (Answer to Ex. 2.5.9).

Knuth adopted Collins' random-trace simulation methodology to compare best fit, first fit, next fit, and binary buddy. Three size distributions were used, one smooth (uniform) and two spiky.[103] The published results are not very detailed. First fit was found to be better than best fit in terms of space, while next fit was better in terms of time. The (binary) buddy system worked better than expected; its limited coalescing usually worked. Simple segregated storage worked very poorly.[104]

Knuth also presented the "fifty-percent rule" for first fit, and its derivation. This rule states that under several assumptions (effectively random allocation request order, steady-state memory usage, and block sizes infrequently equal to each other) the length of the free list will tend toward being about half the number of blocks actually in use. (All of these assumptions now appear to be false for most programs, as we will explain later in the discussions of [MPS71], [ZG94] and [WJNB95]. Shore would later show that Knuth's simplifying assumptions about the lack of systematicity in the allocator's placement were also unwarranted.[105] Betteridge [Bet82] provides a somewhat different critique of the fifty percent rule.)

In a worst-case analysis, Knuth showed that the binary buddy system requires at most $2M \log_2 n$ memory.

After Knuth's book appeared, many papers showed that (in various randomized simulations) best fit had approximately the same memory usage as address-ordered first fit, and sometimes better, and that next fit had significantly more fragmentation. Nonetheless, next fit became quite popular in real systems. It is unclear whether this is because next fit seems more obviously scalable, or simply because Knuth seemed to favor it and his book was so widely used.

Randell [Ran69] defined internal and external fragmentation, and pointed out that internal fragmentation can be traded for reduced external fragmentation by allocating memory in multiples of some grain size g; this reduces the effective number of sizes and increases the chances of finding a fit.

Randell also reported on simulation experiments with three storage allocation methods: best fit, random fit, and an idealized method that compacts memory continually to ensure optimal memory usage. (All of these methods used a random free list order.) He used the synthetic trace methodology, basing sizes on an exponential distribution and on Totschek's SDC distribution. He found that the grain size g must be very small, or the increase in external fragmentation would outweigh the decrease in internal fragmentation.[106] (Given the smoothing effects of the randomization of requests, and its possibly different effects on internal and external fragmentation, this result should be interpreted with caution.)

Randell used three different placement algorithms. The first (called RELOC) was an idealized algorithm that continually compacted memory to obtain the best possible space usage. The other two (non-compacting) algorithms were best fit (called MIN) and random. Comparisons between these two are not given. The only quantitative data obtainable from the paper are from figures 2 and 3, which show that for best fit, the SDC distribution exhibits less fragmen-

[103] One consisted of the six powers of two from 1 to 32, chosen with probability inversely proportional to size, and the other consisted of 22 sizes from 10 to 4000, chosen with equal probability. The latter distribution appears (now) to be unrealistic in that most real programs' size distributions are not only spiky, but skewed toward a few heavily-used sizes.

[104] This contrasts strongly with our own recent results for synthetic traces using randomized order (but real sizes and lifetimes), described later. We are unsure why this is, but there are many variables involved, including the relative sizes of memories, pages, and objects, as well as the size and lifetime distributions.

[105] Nonetheless, his fifty-percent rule (and others' corollaries) are still widely quoted in textbooks on data structures and operating systems. (To our minds, the fault for this does not lie with Knuth, who presented eminently reasonable first-cut analyses in the course of writing a tremendously ambitious, valuable and general series of books.)

[106] On first reading, Randell's grain sizes seem quite large—the smallest (nonzero) value used was 16 words. Examining Totschek's distribution, however, it is clear that this *is* quite small relative to the average "object" (segment) size [Tot65].

tation (about 11 or 12 percent) than an exponential distribution (about 17 or 18 percent), and both suffer considerably as the grain size is increased.

Minker et al. [M+69] published a technical report which contained a distribution of "buffer sizes" in the University of Maryland UNIVAC Exec 8 system.[107] Unfortunately, these data are imprecise, because they give counts of buffers within ranges of sizes, not exact sizes.

These data were later used by other researchers, some of whom described the distribution as roughly exponential. The distribution is clearly not a simple exponential, however, and the use of averaging over ranges may conceal distinct spikes.[108]

1970 to 1979.

Overview. The 1970's saw a few significant innovations in allocator design and methodology. However, most research was focused on attempts to refine known allocator designs (e.g., the development of various buddy systems), on experiments using different combinations of distributions and allocators, or on attempts to derive analytical formulae that could predict the performance of actual implementations for (randomized) workloads.

Analytic techniques had much greater success within a certain limited scope. Bounds were found for worst-case fragmentation, both for specific algorithms and for all algorithms. The results were not encouraging. Building on Graham's analysis framework, Robson's 1971 paper dashed any hope of finding an allocator with low fragmentation in the worst case.

Most empirical studies used synthetic trace techniques, which were refined as more information about real lifetime and size distributions became available, and as it became obvious that the relative performance of different algorithms depended on those factors. Exponential distributions became the most common size distribution, and a common lifetime distribution, because empirical data showed that allocations of small and short-lived objects were frequent. The fact that these distributions were often spiky—or effectively smoothed in the statistics-gathering process—was often overlooked, as was the non-independence of requests.

Perhaps the most innovative and empirical paper of this period was Margolin's, which used sound methodology, and evaluated a new form of deferred coalescing.

Fenton and Payne's "half fit" policy is also novel and interesting; it is based on a very different strategy from those used in other allocators. Wise's (unpublished) double buddy design is also well-motivated. Purdom, Stigler and Cheam introduced the segregated fits mechanism, which did not receive the attention it was due.

Batson and Brundage's statistics for Algol-60 segment sizes and lifetimes were quite illuminating, and their commentary insightfully questioned the plausibility of the usual assumptions of randomness and independence. They underscored the difficulty of predicting allocator performance. Unfortunately, though their results and commentary were available in 1974 in a technical report, they were not published in a journal until 1977.

Denning [Den70] used Knuth's fifty percent rule to derive an "unused memory rule", which states that under assumptions of randomness and steady-state behavior, fragmentation generally increases memory usage by about half; he also pointed out that sequential free list searches tend to be longer when memory is heavily loaded. **Gelenbe** also derived a similar "two thirds rule" [Gel71] in a somewhat different way. (These essentially identical rules are both subject to the same criticisms as Knuth's original rule.)

Purdom and Stigler [PS70] performed statistical analyses of the binary buddy system, and argued that limitations on buddy system coalescing were seldom a problem. Their model was based on strong assumptions of independence and randomness in the workload, including exponentially distributed random lifetimes.

Batson, Ju and Wood [BJW70] reported segment size and lifetime distributions in the Univer-

[107] We have not yet obtained a copy of this report—our information is taken from [Rus77] and other secondary sources. We are unclear on exactly what sense of "buffer" is meant, but believe that it means memory used to cache logical segments for processes; we suspect that the sizes reported are ranges because the system used a set of fixed buffer sizes, and recorded those, rather than the exact sizes of segments allocated in those buffers. We are also unsure of the exact units used.

[108] Our tentative interpretation of the data is that the distribution is at least bimodal, with modes somewhere around roughly 5 units (36% of all requests) and roughly 20 units (30% of all requests).

sity of Virginia B5500 system. Most segments were "small"—about 60 percent of the segments in use were 40 (48-bit) words or less in length.

About 90 percent of the programs run on this system, including system programs, were written in Algol, and the sizes of segments often corresponded to the sizes of individual program objects, e.g., Algol arrays. (In many other systems, e.g., Totschek's SDC system, segments were usually large and might contain many individual program objects.) The data were obtained by sampling at various times, and reflect the actual numbers of segments in use, not the number of allocation requests.

This distribution is weighted toward small objects, but Batson et al. note that it is not well described as an exponential. Unfortunately, their results are presented only in graphs, and in roughly exponentially spaced bins (i.e., more precise for smaller objects than large ones). This effectively smooths the results, making it unclear what the actual distribution is, e.g., whether it is spiky. The general shape (*after* smoothing) has a rounded peak for the smaller sizes, and is roughly exponential after that. (In a followup study [BB77], described later, Batson and Brundage would find spikes.)

A note about Algol-60 is in order here. Algol-60 does *not* support general heap allocation—all data allocations are associated with procedure activations, and have (nested) dynamic extents. (In the case of statically allocated data, that extent is the entire program run.) In the B5500 Algol system, scalar variables associated with a procedure were apparently allocated in a segment; arrays were allocated in separate segments, and referenced via an indirection. Because of the B5500's limit of 1023 words per segment, large arrays were represented as a set smaller arrays indexed by an array of descriptors (indirections).[109]

Because of this purely block-structured approach to storage allocation, Algol-60 data lifetimes may be more closely tied to the phase structure of the program than would be expected for programs in more modern languages with a general heap. On the other hand, recent data for garbage-collected systems [Wil95] and for C and C++ programs [WJNB95] suggest that the majority of object lifetimes in modern programs are also tied to the phase structure of programs, or to the single large "phase" that covers the whole duration of execution.

[109] Algol-60's dynamically sized arrays may complicate this scenario somewhat, requiring general heap allocation, but apparently a large majority of arrays were statically sized and stack-like usage predominated.

Campbell introduced an "optimal fit" policy, which is a variant of next fit intended to improve the chances of a good fit without too much cost in extra searching [Cam71]. (It is not optimal in any useful sense.) The basic idea is that the allocator looks forward through the linear list for a bounded number of links, recording the best fit found. It then proceeds forward looking for another fit at least as good as what it found in that (sample) range. If it fails to find one before traversing the whole list, it uses the best fit it found in the sample range. (That is, it degenerates into exhaustive best fit search when the sample contains the best fit.)

Campbell tested this technique with a real program (a physics problem), but the details of his design and experiment were strongly dependent on unusual coordination between the application program and the memory allocator. After an initial phase, the application can estimate the number of blocks of different sizes that will be needed later. Campbell's algorithm exploited this information to construct a randomized free list containing a good mix of block sizes.

While Campbell's algorithm worked well in his experiment, it seems that his results are not applicable to the general allocation problem, and other techniques might have worked as well or better. (For example, constructing multiple free lists segregated by size, rather than a random unified free list that must be searched linearly. See also the discussion of [Pag82], later in this section.)

Purdom, Stigler, and Cheam [PSC71] introduced segregated fits using size classes with range lists (called "segregated storage" in their paper). The nature and importance of this efficient mechanism for best-fit-like policies was not generally appreciated by later researchers (an exception being Standish [Sta80]). This may be because their paper's title gave no hint that a novel algorithm was presented.

Purdom et al. used the random trace methodology to compare first fit, binary buddy, and segregated fits. (It is unclear which kind of first fit was used, e.g., LIFO-ordered or address-ordered). Their segregated fits scheme used powers-of-two size classes.

They reported that memory usage for segregated fits was almost identical to that of first fit, while binary buddy's was much worse.

> Every year, after the lottery, Mr. Summers began talking again about a new box, but every year the subject was allowed to fade off without anything's being done. The black box grew shabbier each year; by now it was no longer completely black but splintered badly among one side to show the original wood color, and in some places faded or stained.
> —*Shirley Jackson*, "The Lottery"

Margolin et al. used real traces to study memory allocation in the CP-67 control program of an IBM System/360 mainframe [MPS71]. (Note that this allocator allocated storage used by the operating system itself, not for application programs.)

They warned that examination of their system showed that several assumptions underlying the usual methodology were false, for their system's workload: uncorrelated sizes and lifetimes, independence of successive requests, and well-behaved distributions. Unfortunately, these warnings were to go generally unheeded for two decades, despite the fact that some later researchers used the distributions they reported to generate randomly-ordered synthetic traces. (We suspect that their careful analysis of a single system was not given the attention it deserved because it seemed too *ad hoc*.)

Their size distribution was both spiky and skewed, with several strong modes of different sizes. Nearly half (46.7%) of all objects were of size 4 or 5 doublewords; sizes 1 and 8 (doublewords) accounted for about 11% each, and size 29 accounted for almost 16% of the remainder. Many sizes were never allocated at all.

Margolin et al. began with an address-ordered first fit scheme, and added deferred coalescing. Their major goal was to decrease the time spent in memory management inside the CP-67 control program, without an undue increase in memory usage. Their deferred coalescing subpools (quick lists) pre-allocated some fraction (50% or 95%) of the expected maximum usage of objects of those sizes. (This scheme does not appear to adapt to changes in program behavior.) Deferred coalescing was only used for frequently-allocated sizes.

For their experiments, they used several traces from the same machine, but gathered at different times and on different days. They tuned the free list sizes using one subset of the traces, and evaluated them using another. (Their system was thus tuned to a particular installation, but not a particular run.)

They found that using deferred coalescing increased memory usage by approximately zero to 25%, while generally decreasing search traversals to a small fraction of the original algorithm's. In actual tests in the real system, time spent in memory management was cut by about a factor of six.

Robson [Rob71] showed that the worst-case performance of a worst-case-optimal algorithm is bounded from below by a function that rises logarithmically with the ratio n (the ratio of the largest and smallest block sizes), i.e., $M \log_2 n$ times a constant.

Isoda, Goto and Kimura [IGK71] introduced a bitmapped technique for keeping track of allocated and unallocated buddies in the (binary) buddy system. Rather than taking a bit (or several, as in Knowlton's original scheme) out of the storage for each block, their scheme maintains a bit vector corresponding to the words of memory. The bit for the last word of each block, and the bit for the last word occupied by a block is set. The buddy placement constraint lets these be used as "tail lamps" to look efficiently look through memory to find the ends of preceding blocks.

Hirschberg [Hir73] followed Knuth's suggestion and devised a Fibonacci buddy system; he compared this experimentally to a binary buddy. His experiment used the usual synthetic trace methodology, using a real distribution of block sizes (from the University of Maryland UNIVAC Exec 8 system [M$^+$69]) and exponential lifetime distribution. His results agreed well with the analytically derived estimates; Fibonacci buddy's fragmentation increased memory usage by about 25%, compared to binary buddy's 38%.

Hirschberg also suggested a generalization of the buddy system allowing Fibonacci-like series where each size was the sum of the previous size and a size a fixed distance further down in the size series. (For some fixed integer k, the ith size in the series may be split into two blocks of sizes $i-1$ and $i-k$.)

Robson [Rob74] put a fairly tight upper and lower bounds on the worst-case performance of the best possible allocation algorithm. He showed that a worst-case-optimal strategy's worst-case memory usage was somewhere between $0.5M \log_2 n$ and about $0.84M \log_2 n$.

Shen and Peterson introduced the weighted buddy method [SP74], whose allowable block sizes are either powers of two, or three times a power of two.

They compared this scheme to binary buddy, using the synthetic trace methodology; they used only a uniform lifetime distributions, and only two size distributions, both smooth (uniform and exponential). This is unfortunate, because skew in object size request may affect the effectiveness of different block-splitting schemes.

They found that for a uniform size distribution, weighted buddy lost more memory to fragmentation than binary buddy, about 7%. For an exponential distribution (which is apparently more realistic) this was reversed—weighted buddy did better by about 7%. By default, they used FIFO-ordered free lists. With LIFO-ordered free lists, memory usage was about 3% worse.

Using a variation of the random trace methodology intended to approximate a segment-based multiprogramming system,[110] **Fenton and Payne** [FP74] compared best fit (called "least fit"), first fit, next fit, worst fit, and "half fit." The half fit policy allocator attempts to find a block about twice the desired size, in the hopes that if there is a bias toward particular sizes, remainders from splitting will be more likely to be a good fit for future requests. They found that best fit worked best, followed by first fit, half fit, next fit, and worst fit, in that order. Half fit was almost as good as first fit, with next fit performing significantly worse, and worst fit much worse.

All of the size distributions used in their experiments were smooth. For many of their experiments, they used a smooth distribution based on generalizations about Totschek's SDC distribution and Batson, Ju, and Wood's B5500 distribution. (This is a "deformed exponential" distribution, which rises quickly, rounds off at the top, and then descends in a roughly exponential fashion.) Fenton and Payne apparently didn't consider the possibility that smooth distributions (and randomized order) might make their half-fit policy work worse than it would in practice, by decreasing the chance that a request for a particular size would be repeated soon.

[110] In this model, each object (segment) is assumed to be associated with a different process. When a request cannot be satisfied, that process blocks (i.e., the death time of the segment is delayed, but time advances so that other segments may die). This models embodies an oversimplification relative to most real systems, in that processes in most systems may have multiple associated segments whose death times cannot be postponed independently.

Hinds [Hin75] presented a fast scheme for recombination in binary and generalized Fibonacci buddy systems. Each block has a "left buddy count" indicating whether it is a right buddy at the lowest level (in which case the LBC is zero), or indicating for how many levels above the lowest it is a left buddy. This supports splitting and merging nearly as quickly as in the binary buddy scheme.

Cranston and Thomas [CT75] presented a method for quickly finding the buddy of a block in various buddy systems, using only three bits per block. This reduces the time cost of splitting and merging relative to Hirschberg's scheme, as well as incurring minimal space cost.

Shore [Sho75] compared best fit and address-ordered first fit more thoroughly than had been done previously, and also experimented with worst-fit and a novel hybrid of best fit and first fit. He used the then-standard methodology, generating random synthetic traces with (only) uniformly distributed lifetimes. Size distributions were uniform, normal, exponential, and hyperexponential. He also performed limited experiments with "partial populations" (i.e., spiky distributions). The figure of merit was the space-time product of memory usage over time. (This essentially corresponds to the average memory usage, rather than peak usage.)

This study was motivated in part by Wald's report of the "somewhat puzzling success" of best fit in actual use in the Automatic Operating and Scheduling Program of the Burroughs D-825 system [Wal66]. (Fragmentation was expected to be a problem; plans were made for compaction, but none was needed.)

Shore found that best fit and (address-ordered) first fit worked about equally well, but that first fit had an advantage when the distribution included block sizes that were relatively large compared to the memory size. Following Knuth [Knu73], he hypothesized that this was due to its tendency to fit small objects into holes near one end of memory, accumulating larger free areas toward the other end.[111]

For partial populations, Shore found that increasing degrees of spikiness seemed to favor best fit over first

[111] We are actually unsure what Shore's claim is here. It is not clear to us whether he is making the general claim that first fit tends to result in a free list that is approximately size-ordered, or only the weaker claim that first fit more often has unusually large free blocks in the higher address range, and that this is important for distributions that include occasional very large blocks.

fit slightly, but that the variance increased so quickly that this result was not reliable.[112]

Shore noted that while first fit and best fit policies are roughly similar, they seem to have somewhat different strengths and weaknesses; he hypothesized that these might be combinable in a hybrid algorithm that would outperform either.

Shore experimented with a novel parameterized allocator, combining features of first fit and best fit. At one extreme setting of the parameter, it behaved like address-ordered first fit, and at the other extreme it behaved like best fit. He found that an intermediate parameter setting showed less fragmentation than either standard algorithm. If this were to be shown to work for real workloads, it could be a valuable result. It suggests that best fit and address-ordered first fit may be exploiting different regularities, and that the two strategies can be combined to give better performance. (Since the inputs were randomly ordered, however, it is not clear whether these regularities exist in real program behavior, or whether they are as important as other regularities.)

Shore also experimented with worst-fit, and found that it performed very poorly.[113]

Shore warned that his results "must be interpreted with caution," and that some real distributions are not well behaved. Citing Margolin, he noted tht "such simplifying assumptions as well-behaved distributions, independence of successive requests, and independence of request sizes and duration are questionable." These warnings apparently received less attention than his thorough (and influential) experimentation within the random trace paradigm.

Burton introduced a generalization of the Fibonacci buddy system [Bur76] which is more general than Hirschberg's. Rather than using a fixed function for generating successive sizes (such as always adding size $i-1$ and $i-3$ to generate size i), Burton points out that different sizes in the series can be used. (For example, adding sizes $i-1$ and $i-2$ to generate i, but adding sizes $j-1$ and $j-4$ to generate size j.) Burton's intended application was for disk storage management, where it is desirable to ensure that the block size, track size, and cylinder size are all in the series. The result is fairly general, however, and has usually been overlooked; it could be used to generate application-specific buddy systems tailored to particular programs' request sizes.

> "You didn't give him time enough to take any paper he wanted. I saw you. It wasn't fair!"
> "Be a good sport, Tessie," Mrs Delacroix called, and Mrs. Graves said, "All of us took the same chance."
> —*Shirley Jackson*, "The Lottery"

Batson and Brundage [BB77] reported segment sizes and lifetimes in 34 varied Algol-60 programs. Most segments were small, and the averaged size distribution was somewhat skewed and spiky. (Presumably the distributions for individual programs were even less well-behaved, with individual spikes being reduced considerably by averaging across multiple programs.)

Lifetime distributions were somewhat better-behaved, but still irregular.[114] When lifetimes were normalized to program running times, evidence of plateau and ramp usage appeared. (In our interpretation of the data, that is. As mentioned earlier, however, Algol-60 associates segment lifetimes with the block structure of the program.)

Batson and Brundage pointed out that lifetimes are *not* independent of size, because some blocks are entered many times, and others only once; most entries

[112] Wald had hypothesized that best fit worked well in his system because of the spiky distribution of requests. Shore notes that "Because there were several hundred possible requests" in that system, the result "was due more probably to a nonsaturating workload." The latter makes sense, because Wald's system was a real-time system and generally not run at saturation. The former is questionable, however, because the distribution of actual requests (and of live data) is more important than the distribution of possible requests.

[113] He drew the (overly strong) conclusion that good fits were superior to poor fits; we suggest that this isn't always the case, and that the strengths of worst fit and best-fit-like policies might be combinable. Worst fit has the advantage that it tends to not to create small remainders, as best fit does. It has the disadvantage that it tends to ensure that there are *no* very large free areas—it systematically whittles away at the largest free block until it is no longer the largest. A hybrid strategy might use poor fits, but preserve some larger areas as well.

[114] Recall that looking at distributions is often misleading, because sudden deaths of objects born at different times will result in a range of lifetimes. (Section 2.4) Small irregularities in the lifetime distribution may reflect large dynamic patterns.

to the same block allocate exactly the same number and sizes of segments. They stated that they had no success fitting any simple curve to their data, and that this casts doubts on analyses and experiments assuming well-behaved distributions.

They also suggested that the experiments of Randell, Knuth, and Shore could be redone be using realistic distributions, but warned that "we must wait for a better understanding" of "the dynamics of the way in which the allocated space is *used*—before we can make reasonable predictions about the comparative performance of different mechanisms." They go on to say that "there is no reason to suppose that stochastic processes could possibly generate the observed request distributions."

Though based on a 1974 technical report, this paper was not published until 1977, the same year that saw publication of a flurry of papers based on random traces with well-behaved distributions. (Described below.)

Weinstock [Wei76] surveyed most of the important work in allocators before 1976, and presented new empirical results. He also introduced the "QuickFit" algorithm, a deferred coalescing scheme using size-specific lists for small block sizes, backed by LIFO-ordered first fit as the general allocator.[115] (Weinstock reported that this scheme was invented several years earlier for use in the Bliss/11 compiler [WJW+75], and notes that a similar scheme was independently developed and used in the Simscript II.5 language [Joh72]. Margolin's prior work was overlooked, however.)

Weinstock used the conventional synthetic trace methodology; randomly-ordered synthetic traces were generated, using two real size distributions and four artificial ones. One of the real size-and-lifetime distributions came from the Bliss/11 compiler [WJW+75], and the other was from Batson and Brundage's measurements of the University of Virginia B5500 system [BB77], described above. The four artificial size distributions were uniform, exponential, Poisson, and a two-valued distribution designed to be a bad case for first fit and best fit. (The two-valued distribution was not used in the final evaluation of allocators.)

The Bliss/11 distribution is heavily weighted toward small objects, but is not well-described by an exponential curve. It has distinct spikes at 2 words (44% of all objects) and 9 words (14%). In between those spikes is another elevation at 5 words and 6 words (9% each).

The figures of merit for space usage in this study were probabilities of failure in different-sized memories. (That is, how likely it was that the synthetic program would exhaust memory and fail, given a particular limited memory size.) This makes the results rather difficult reading, but the use of fixed memory sizes allows experimentation with allocators which perform (deferred) coalescing only when memory is otherwise exhausted.

Weinstock experimented with QuickFit, best fit, first fit, next fit, and binary buddies. Variations of best fit used address-ordered or size-ordered free lists. Variations of first fit and next fit used address-ordered and LIFO-ordered free lists. The address-ordered versions of best, first, and next fit were also tried with immediate coalescing and deferred coalescing. Two binary buddy systems were used, with immediate and deferred coalescing. (In all cases, deferred coalescing was only performed when memory was exhausted; no intermediate strategies were used.)

In general, Weinstock found that address-ordered best fit had the best space usage, followed closely by address-ordered first fit. (Both did about equally well under light loadings, i.e., when memory was more plentiful.)

After address-ordered best fit came a cluster of algorithms whose ranking changed depending on the loading and on the distributions used: address-ordered first fit, address-ordered best fit with deferred coalescing, size-ordered best fit, and Quick Fit.

After that came a cluster containing address-ordered first fit with deferred coalescing and address-ordered next fit. This was followed by address ordered next fit with deferred coalescing, followed in turn by LIFO-ordered first fit. Binary buddies performed worst, with little difference between the immediate and deferred coalescing variants.

In summary, address-ordered variants tended to outperform other variants, and deferred coalescing (in the extreme form used) usually increased fragmentation. FIFO-ordered lists were not tried, however.

In terms of speed, QuickFit was found to be fastest, followed by binary buddy with deferred coalescing. Then came binary buddy with immediate coalescing. Rankings are given for the remaining allocators, but these are probably not particularly useful; the remain-

[115] This is not to be confused with the later variant of QuickFit [WW88], which does no coalescing for small objects, or Standish and Tadman's indexed fits allocator.

ing algorithms were based on linear list implementations, and could doubtless be considerably improved by the use of more sophisticated indexing systems such as splay trees or (in the case of best fit) segregated fits.

Weinstock made the important point that seemingly minor variations in algorithms could have a significant effect on performance; he therefore took great care in the describing of the algorithms he used, and some of the algorithms used in earlier studies.

In a brief technical communication, **Bays** [Bay77] replicated some of Shore's results comparing first fit and best fit, and showed that next fit was distinctly inferior when average block sizes were small. When block sizes were large, all three methods degraded to similar (poor) performance. (Only uniformly distributed lifetimes and exponentially distributed sizes were used.)

> "Seems like there's no time at all between lotteries any more," Mrs. Delacroix said to Mrs. Graves in the back row.
> —*Shirley Jackson*, "The Lottery"

Peterson and Norman [PN77] described a very general class of buddy systems, and experimentally compared several varieties of buddy systems: binary, Fibonacci, a generalized Fibonacci [HS64, Fer76], and weighted. They used the usual random trace methodology, with both synthetic (uniform and exponential) and real size distributions. Their three size distributions were Margolin's CP-67 distribution, the University of Maryland distribution, and a distribution from an IBM 360 OS/MVT system at Brigham Young University. (This "BYU" distribution was also used in several later studies.) They point out that the latter two distributions were imprecise, grouping sizes into ranges; they generated sizes randomly within those ranges. (The implication of this is that these distributions were smoothed somewhat; only the CP-67 distribution is truly natural.)

(The BYU distribution is clearly not exponential, although some later researchers would describe it that way; while it is skewed toward small sizes, it is at least bimodal. Given that it is reported in averages over ranges, there may be other regularities that have been smoothed away, such as distinct spikes.)

We are unsure what lifetime distribution was used.

Peterson and Norman found that these buddy systems all had similar memory usage; the decreases in internal fragmentation due to more-refined size series were usually offset by similar increases in external fragmentation.

Robson [Rob77] showed that the worst-case performance of address-ordered first fit is about $M \log_2 n$, while best fit's is far worse, at about Mn. He also noted that the roving pointer optimization made next fit's worst case similarly bad—both best fit and next fit can suffer about as much from fragmentation as any allocator with general splitting and coalescing.

Nielsen [Nie77] studied the performance of memory allocation algorithms for use in simulation programs. His main interest was in finding fast allocators, rather than memory-efficient allocators. He used a variation of the usual random trace methodology intended to model the workloads generated by discrete-event simulation systems. A workload was modeled as a set of streams of event objects; each stream generated only requests of a single size, but these requests were generated randomly according to size and inter-arrival time distributions associated with the streams. To construct a workload, between 3 and 25 request streams were combined to simulate a simulation with many concurrent activities.

Eighteen workloads (stream combinations) were used. Of these, only two modeled any phase behavior, and only one modeled phases that affected different streams (and object sizes) in correlated ways.[116]

Nielsen's experiments were done in two phases. In the first phase a single workload was used to test 35 variants of best fit, first fit, next fit, binary buddies, and segregated fits. (This workload consisted of 10 streams, and modeled no phase behavior.) Primarily on the basis of time costs, all but seven of the ini-

[116] In our view, this does not constitute a valid cross-section of discrete event simulation programs, for several reasons. (They may better reflect the state of the art in simulation at the time the study was done, however.) First, in many simulations, events are not generated at random, but in synchronized pulses or other patterns. Second, many events in some simulations are responses to emergent interactions of other events, i.e., patterns in the domain-level systems being simulated. Third, many simulation programs have considerable state local to simulated objects, in addition to the event records themselves. Fourth, many simulation systems include analysis facilities which may create objects with very different lifetime characteristics than the simulation objects themselves; for example, an event log that accumulates monotonically until the simulation terminates.

tial set of allocators were eliminated from consideration. (This is unfortunate, because different implementation strategies could implement many of the same policies more efficiently. Best fit and address-ordered first fit were among the policies eliminated.) Of the surviving seven allocators, six had poor memory usage. The seventh allocator, which performed quite well in terms of both speed and memory usage, was "multiple free lists," i.e., segregated fits with exact lists.

In [Sho77], **Shore** analyzed address-ordered first fit theoretically, and showed that the allocator itself violates a statistical assumption underlying Knuth's fifty percent rule. He argued that systematicity in the placement of objects interacts with "the statistics of the release process" to affect the length of the the free list under equilibrium conditions.

Shore demonstrated that the relative performance of best fit and (address-ordered) first fit depended on the shape of the lifetime distribution.

Shore was primarily concerned with simple, well behaved distributions, however, and made the usual assumptions of randomness (e.g., independence of successive allocations, independence of size and lifetime). He did not consider possible systematicities in the application program's allocations and releases, such as patterned births and deaths. (He did aptly note that "the dynamics of memory usage comprise complicated phenomena in which observable effects often have subtle causes.")

Russell [Rus77] attempted to derive formulas for expected fragmentation in a Fibonacci and a generalized Fibonacci buddy system,[117] based on the assumption that size distributions followed a generalization of Zipf's law (i.e., a decreasing function inversely related to the sizes). Based on this assumption, he derived estimated lower and upper bounds, as well as estimated average performance. He compared this to simulation results, using the conventional synthetic trace methodology and basing size distributions on three real distributions (Margolin's CP-67 distribution, the BYU distribution, and the U. of Maryland distribution.) For the generalized Fibonacci system, average fragmentation for the three workloads was close to what was predicted (22% predicted, 21% observed). For the plain Fibonacci system, the error was significant (29% predicted, 22% observed). For binary buddy the error was rather large (44% predicted, 30% observed).

Russell notes that the CP-67 data do not closely resemble a Zipf distribution, and for this distribution the fragmentation using conventional Fibonacci is in fact lower (at 15%) than his estimated lower bound (24%). Averaging just the results for the other two distributions brings the results closer to the predicted values on average, but for generalized Fibonacci they move further away. We believe that his estimation technique is unreliable, partly because we do not believe that distributions are generally exponential, and partly because of the randomness of request order that he assumes.

Wise, in an unpublished technical report [Wis78], described a double buddy system and its advantages over Fibonacci systems in terms of external fragmentation (producing free blocks of the same size as requested blocks). This report apparently went unnoticed until well after double buddy was reinvented by Page and Hagins [PH86].[118]

Reeves [Ree79, Ree80, Ree82, Ree83] used analytic techniques to determine the effect of a random fit allocator policy in the face of random workloads, using a "generating function" approach originated by Knuth [Knu73]. This work relies extremely heavily on randomness assumptions—usually in both the workload and the allocator—to enable the analyses of memories of significant size.

1980 to 1990.

> People at first were not so much concerned with what the story meant; what they wanted to know was where these lotteries were held, and whether they could go there and watch.
> —*Shirley Jackson*, "On the Morning of June 28, 1948, and 'The Lottery' "

Overview. The 1980–1990 period saw only modest development of new allocator techniques, and little new in the way of evaluation methodologies, at least in academic publications. Despite doubts cast by Margolin and Batson, most experimenters continued to use synthetic traces, often with smooth and well-behaved

[117] See also Bromley [Bro80].

[118] The first author of the present paper overlooked both and reinvented it yet again in 1992. It is next expected to appear in the year 2000.

distributions. This is probably due to the lack of a comprehensive survey addressing methodological concerns. (The present paper is an attempt to remedy that problem.) By this time, there were many papers on allocators, and Margolin's and Batson's were probably not among the most studied.[119] Most theoretical papers continued to make strong assumptions of randomness and independence, as well, with the exception of papers about worst-case performance.

Among the more interesting designs from this period are Standish and Tadman's exact lists scheme, Page and Hagins' double buddy system, Beck's age-match algorithm, and Hanson's obstack system.

Standish surveyed memory allocation research in a (short) chapter of a book on data structures [Sta80], describing segregated fits and introducing a segregated free lists method using exact lists. Citing Tadman's masters thesis [Tad78], he reported that an experimental evaluation showed this scheme to perform quite similarly to best fit—which is not surprising, because it *is* best fit, in policy terms—and that it was fast. (These experiments used the usual synthetic trace methodology, and Standish summarized some of Weinstock's results as well.)

Page [Pag84] analyzed a "cyclic placement" policy similar to next fit, both analytically and in randomized simulations. (Only uniformly distributed sizes and lifetimes were used.) The cyclic placement scheme generally resulted in significantly more fragmentation than first fit or best fit.

> "...over in the north village they're talking of giving up the lottery."
> —*Shirley Jackson*, "The Lottery"

Leverett and Hibbard [LH82] performed one of the all-too-rare studies evaluating memory allocators using real traces. Unfortunately, their workload consisted of five very small programs (e.g., towers of Hanoi, knight's tour) coded in Algol-68; none was more than 100 lines. It is unclear how well such textbook-style programs represent larger programs in general use.

[119] Margolin's paper was published in an IBM journal, while the main stream of allocator papers was published in *Communications of the ACM*. Batson and Brundage's paper was published in *CACM*, but its title may not have conveyed the significance of their data and conclusions.

Algol-68 did support general heap allocation, an improvement over Algol-60. The Algol-68 system used for experiments used reference counting to reclaim space automatically.[120] (Deferred) coalescing was performed only when memory is exhausted. The general allocator was first fit with a LIFO-ordered free list.

LIFO-ordered quick lists for different-sized blocks were used, as well as per-procedure lists for activation records,[121] and some lists for specific data types. Deferred coalescing greatly improved the speed of their allocator, and usually decreased overall memory usage.

Leverett and Hibbard also found that Knuth's roving pointer modification (i.e., next fit) was disappointing; search lengths did not decrease by much, and for some programs got longer.

Page [Pag82] evaluated Campbell's "optimal fit" method analytically and in randomized trace simulations. (Page's version of optimal fit was somewhat different from Campbell's, of necessity, since Campbell's was intertwined with a particular application program structure.) Page showed that Campbell's analysis erred in assuming randomness in first-fit-like placement policies, and that systematicities in placement matter considerably. In Page's analysis and simulations, Campbell's "optimal" fit was distinctly inferior to first fit and best fit in both search times and memory usage. (Only uniformly distributed sizes and lifetimes were used, however.)

Page also showed that (for uniformly distributed sizes and lifetimes), a first fit policy resulted in the same placement decisions as best fit most of the time, if given the same configuration of memory and the same request. He also showed that the free list for first fit tended toward being roughly sorted in size order. (See also similar but possibly weaker claims in [Sho75], discussed earlier.)

[120] A possibly misleading passage says that memory is freed "explicitly," but that is apparently referring to a level of abstraction below the reference counting mechanism. Another potentially confusing term, "garbage collection," is used to refer to deferred coalescing where coalescing is performed only when there is no sufficiently large block to satisfy a request. This is very different from the usual current usage of the term [Wil95], but it is not uncommon in early papers on allocators.

[121] Activation records were apparently allocated on the general heap; presumably this was used to support closures with indefinite extent (i.e., "block retention"), and/or "thunks" (hidden parameterless subroutines) for call-by-name parameter passing [Ing61].

Betteridge [Bet82] attempted to compute fragmentation probabilities for different allocators using first-order Markov modeling. (This book is apparently Betteridge's dissertation, completed in 1979.) The basic idea is to model all possible states of memory occupancy (i.e., all arrangements of allocated and free blocks), and the transition probabilities between those states. Given a fixed set of transition probabilities, it is possible to compute the likelihood of the system being in any particular state over the long run. This set of state probabilities can then be used to summarize the likelihood of different degrees of fragmentation.

Unfortunately, the number of possible states of memory is exponential in the size of memory, and Betteridge was only able to compute probabilities for memories of sizes up to twelve units. (These units may be words, or they may be interpreted as some larger grain size. However, earlier results suggest that small grain sizes are preferred.) He suggests several techniques to make it easier to use somewhat larger models, but had little success with the few he tried. (See also [Ben81, Ree82, McI82].) We are not optimistic that this approach is useful for realistic memory sizes, especially since memory sizes tend to increase rapidly over time.

To allow the use of a first-order Markov model, Betteridge assumed that object lifetimes were completely independent—not only must death times be random with respect to allocation order, but there could be *no* information in the request stream that might give an allocator any exploitable hint as to when objects might die. For this, Betteridge had to assume a random exponential lifetime function, i.e., a half-life function where any live object was exactly as likely to die as any other at a given time. (Refer to Section 2.2 for more on the significance of this assumption.) This is necessary to ensure that the frequencies of actual transitions would stabilize over the long run (i.e., the Markov model is *ergodic*—see Section 2.2), and allows the computation of the transition probabilities without running an actual simulation for an inconveniently infinite period of time. The system need not keep track of the sequences of transitions that result in particular states—actual sequences are abstracted away, and only the states where histories intersect are represented.

Even with these extremely strong assumptions of randomness, this problem is combinatorially explosive. (This is true even when various symmetries and rotations are exploited to combine (exactly) equivalent states [Ben81, McI82].)

We believe that the only way to make this kind of problem remotely tractable is with powerful abstractions over the possible states of memory. For the general memory allocation problem, this is simply not possible—for an arbitrary interesting allocator and real request streams, there is always the possibility of systematic and even chaotic interactions. The only way to make the real problem formalizable is to find a useful *qualitative* model that captures the likely range of program behaviors, each allocator's likely responses to classes of request streams, and (most importantly) allows reliable characterization of request streams and allocators in the relevant ways. We are very far away from this deep understanding at present.

Beck [Bec82] described the basic issue of fragmentation clearly, and designed two interesting classes of allocators, one idealized and one implementable. Beck pointed out that basic goal of an allocator is to reduce the number of *isolated* free blocks, and that the existence of isolated free blocks is due to neighboring blocks having *different death times.*

This motivated the design of an idealized offline allocator that looks ahead into the future to find out when objects will die; it attempts to place new objects near objects that will die at about the same time. This policy can't be used in practice, because allocators must generally make their decisions online, but it provides an idealized standard for comparison. This "release-match" algorithm is philosophically similar to Belady's well-known MIN (or OPT) algorithm for optimal demand-paging. (It is heuristic, however, rather than optimal.)

Beck also described an implementable "age match" algorithm intended to resemble release-match, using allocation time to heuristically estimate the deallocation (release) time.

For an exponential size distribution and uniform lifetime distribution, he found that the effectiveness of the age-match heuristic depended on the lifetime variance (i.e., the range of the uniform distribution). This is not surprising, because when lifetimes are similar, objects will tend to be deallocated in the order that they are allocated. As the variance in lifetimes increases, however, the accuracy of prediction is reduced.

Beck also experimented with hyper-exponential lifetime distributions. In this case, the age-match heuristic systematically failed, because in that case the age of an object is negatively correlated with the time un-

til it will die. This should not be surprising. (In this case it might work to *reverse* the order of estimated death times.)

Stephenson [Ste83] introduced the "Fast Fits" technique, using a Cartesian tree of free blocks ordered primarily by address and secondarily by block size. He evaluated the leftmost fit (address-ordered first fit) and better fit variants experimentally. Details of the experiment are not given, but the general result was that the space usage of the two policies was similar, with better fit appearing to have a time advantage. (A caveat is given, however, that this result appears to be workload-dependent, in that different distributions may give different results. This may be a response to the then-unpublished experiments in [BBDT84], but no details are given.)

Kaufman [Kau84] presented two buddy system allocators using deferred coalescing. The first, "tailored list" buddy systems, use a set of size-specific free lists whose contents are not usually coalesced.[122] This system attempts to keep the lengths of the free lists proportional to the expected usage of the corresponding sizes; it requires estimates of program behavior. The second scheme, "recombination delaying" buddy systems, adapts dynamically to the actual workload. In experiments using the usual synthetic trace methodology, Kaufman found that both systems worked quite well at reducing the time spent in memory management. These results are suspect, however, due to the load-smoothing effects of random traces, which flatter small caches of free blocks (Section 3.11).[123]

Bozman et al. [BBDT84] studied a wide variety of allocators, including sequential fits, deferred coalescing schemes, buddy systems, and Stephenson's Cartesian tree system. (Not all allocators were compared directly to each other, because some were tailored to an IBM operating system and others were not.) They used synthetic traces based on real lifetime distributions, primarily from two installations of the same IBM operating system, VM-SP. (Their main goal was to develop an efficient allocator for that system.) They also measured the performance of a resulting algorithm in actual use in the VM-SP system.

First, Bozman et al. compared first fit, next fit and best fit with the VM-SP algorithm. This algorithm, based on earlier research by Margolin et al., used deferred coalescing with a general pool managed by address-ordered first fit. In terms of fragmentation, VM-SP was best, followed by best fit, which was significantly better than first fit. This result is unclear, however, because they don't state which variety of first fit they were using (e.g., address-ordered or LIFO-ordered free lists). Next fit was considerably worse, using about 50% more memory than the VM-SP algorithm.

They then compared best-fit-first (taking the first of several equally good fits) with best-fit-last (taking the last), and found that best-fit-last was better. They also added a splitting threshold, which reduced the difference between best fit and first fit. (We are not sure whether these got better or worse in absolute terms.) Adding the splitting threshold also reversed the order of best-fit-first and best-fit-last.

Bozman et al. also tested a binary buddy and a modified Fibonacci buddy. They found that the memory usage of both was poor, but both were fast; the memory usage of the modified Fibonacci buddy was quite variable.

Testing Stephenson's Cartesian tree allocator, they found that the leftmost fit (address ordered first fit) policy worked better than the "better fit" policy; they latter suffered from "severe" external fragmentation for the test workload. They suggest that leftmost fit would make a good general allocator in a system with deferred coalescing.

After these initial experiments, Bozman et al. developed a fast deferred coalescing allocator. This allocator used 2 to 15 percent more memory than best fit, but was much faster. We note that the extra memory usage was likely caused at least in part by the policy of keeping "subpools" (free lists caching free blocks of particular sizes) long enough that the miss rate was half a percent or less. (That is, no more than one in two hundred allocations required the use of the general allocator.)

This allocator was deployed and evaluated in the same installations of the VM-SP operating system from which their test statistics had been gathered. The performance results were favorable, and close to what was predicted. From this Bozman et al. make the general claim—which is clearly far too strong—that

[122] This tailoring of list length should not be confused with the tailoring of size classes as mentioned in [PN77].

[123] The tailored list scheme worked better than the recombination delaying scheme, but this result is especially suspect; the tailored list scheme does not respond dynamically to the changing characteristics of the workload, but this weakness is not stressed by an artificial trace without significant phase behavior.

the statistical assumptions underlying the random-trace methodology are not a problem, and that the results are highly predictive. (We believe that this conclusion is difficult to support with what amount to two data points, especially since their validation was primarily relevant to variations on a single optimized design, not the wide variety of basic allocators they experimented with using synthetic traces.)

In a related paper, **Bozman** [Boz84] described a general "software lookaside buffer" technique for caching search results in data structures. One of his three applications (and empirical evaluations) was for deferred coalescing with best fit and address-ordered first fit allocators. In that application, the buffer is a FIFO queue storing the size and address of individual blocks that have been freed recently. It is searched linearly at allocation time.

For his evaluation, Bozman used the conventional synthetic trace methodology, using a real size distribution from a VM-SP system and exponentially distributed lifetimes; he reported considerable reductions in search lengths, in terms of combined FIFO buffer and general allocator searches. (It should be noted that both general allocators used were based on linear lists, and hence not very scalable to large heaps; since the FIFO buffer records individual free blocks, it too would not scale well. With better implementations of the general allocator, this would be less attractive. It also appears that the use of a randomized trace is likely to have a significant effect on the results (Section 3.11).

Coffman, Kadota, and Shepp [CKS85] have conjectured that address-ordered first fit approaches optimal as the size of memory increases. They make very strong assumptions of randomness and independence, including assuming that lifetimes are unrelated and exponentially distributed.

In support of this conjecture, they present results of simulations using pseudo-random synthetic traces, which are consistent with their conjecture. They claim that "we can draw strong engineering conclusions from the above experimental result."

Naturally, we are somewhat skeptical of this statement, because of the known non-randomness and non-independence observed in most real systems. Coffman, Kadota, and Shepp suggest that their result indicates that large archival storage systems should use first fit rather than more complex schemes, but we believe that this result is inapplicable there. (We suspect that there are significant regularities in file usage that are extremely unlikely to occur with random traces using smooth distributions, although the use of compression may smooth size distributions somewhat.)

We also note that for secondary and tertiary storage more generally, *contiguous* storage is not strictly required; freedom from this restriction allows schemes that are much more flexible and less vulnerable to fragmentation. (Many systems divide all files into blocks of one or two fixed sizes, and only preserve *logical* contiguity (e.g., [RO91, VC90, SKW92, CG91, AS95]). If access times are important, other considerations are likely to be much more significant, such as locality. (For rotating media and especially for tapes, placement has more important effects on speed than on space usage.)

Oldehoeft and Allan [OA85] experimented with variants of deferred coalescing, using a working-set or FIFO policy to dynamically determine which sizes would be kept on quick lists for for deferred coalescing. The system maintained a cache of free lists for recently-freed sizes. (Note that where Bozman had maintained a cache of individual free blocks, Oldehoeft and Allan maintained a cache of free lists for recently-freed *sizes*.) For the FIFO policy, this cache contains a fixed number of free lists. For the Working Set policy, a variable number of free lists are maintained, depending on how many sizes have been freed within a certain time window. In either policy, when a free list is evicted from the cache, the blocks on that list are returned to the general pool and coalesced if possible. Note that the number and size of uncoalesced free blocks is potentially quite variable in this scheme, but probably less so than in schemes with fixed-length quick lists.

One real trace was used, and two synthetic traces generated from real distributions. The real trace was from a Pascal heap (program type not stated) and the real distributions were Margolin's CP-67 data and Leverett and Hibbard's data for small Algol programs.

Oldehoeft and Allan reported results for FIFO and Working Set with comparable average cache sizes. The FIFO policy may defer the coalescing of blocks for a very variable time, depending on how many different sizes of object are freed. The Working Set policy to coalesce all blocks of sizes that haven't been freed within its time window. Neither policy bounds the volume of memory contained in the quick lists, although it would appear that Working Set is less likely to have excessive amounts of idle memory on quick lists.

The Working Set policy yielded higher hit rates—

i.e., more allocations were satisfied from the size-specific lists, avoiding use of the general allocator.

They also experimented with a totally synthetic workload using uniform random size and lifetime distributions. For that workload, Working Set and FIFO performed about equally, and poorly, as would be expected.

Effects on actual memory usage were not reported, so the effect of their deferred coalescing on overall memory usage is unknown.

Korn and Vo [KV85] evaluated a variety of UNIX memory allocators, both production implementations distributed with several UNIX systems, and new implementations and variants. Despite remarking on the high fragmentation observed for a certain usage pattern combined with a next fit allocator (the simple loop described in Section 3.5), they used the traditional synthetic trace methodology. (Vo's recent work uses real traces, as described later.) Only uniform size and lifetime distributions were used. They were interested in both time and space costs, and in scalability to large heaps.

Five of their allocators were variants of next fit.[124] The others included simple segregated storage (with powers of two size classes)[125] address-ordered first fit (using a self-adjusting "splay" tree [ST85]), segregated fits (using Fibonacci-spaced size classes), better fit (using Stephenson's Cartesian tree scheme), and two best fit algorithms (one using a balanced binary tree, and the other a splay tree).

It may be significant that Korn and Vo modified most of their allocators to include a "wilderness preservation heuristic," which treats the last block of the heap memory area specially; this is the point (called the "break") where the heap segment may be extended, using UNIX `sbrk()` system call, to obtain more virtual memory pages from the operating system. (See Section 3.5.)

To summarize their results, we will give approximate numbers obtained by visual inspection of their Figure 3. (These numbers should be considered very approximate, because the space wastage varied somewhat with mean object size and lifetimes.)

Space waste (expressed as an increase over the amount of live data, and in increasing order), was as follows. Best fit variants worked best, with space wastage of roughly 6 to 11 percent (in order of increasing waste, best fit (splay), best fit (balanced), better fit Cartesian). Segregated fits followed at about 16 percent. Address-ordered next fit wasted about 20 percent, and address-ordered first fit wasted about 24 percent. Standard next fit and a variant using adaptive search followed, both at about 26 percent. Two other variants of next fit followed at a considerable distance; one used a restricted search (42 percent) and the other treated small blocks specially (45 percent). Simple segregated storage (powers of two sizes) was worst at about 47 percent. (These numbers should be interpreted with some caution, however; besides the general problem of using synthetic workloads, there is variation among the allocators in per-block overheads.)

In terms of time costs, two implementations scaled very poorly, being fast for small mean lifetimes (and hence heap sizes), but very slow for large ones. The implementations of these algorithms both used linear lists of *all* blocks, allocated or free. These algorithms were a standard next fit and an address-ordered next fit.

Among the other algorithms, there were four clusters at different time performance levels. (We will name the algorithms within a cluster in approximately increasing cost order.) The first cluster contained only simple segregated storage, which was by far the fastest. The second cluster contained next fit with restricted search, next fit with special treatment of small blocks, segregated fits, and next fit with adaptive search. (This last appeared to scale the worst of this cluster, while segregated fits scaled best.) The third cluster contained best fit (splay), better fit (Cartesian), and address-ordered first fit (splay).

Gai and Mezzalama [GM85] presented a very simple deferred coalescing scheme, where only one size class is treated specially, and the standard C library allocator routines are used for backing storage. (The algorithms used in this library are not stated, and are not standardized.)

Their target application domain was concurrent simulations, where many variations of a design are tested in a single run. As the run progresses, faults are detected and faulty designs are deleted.[126] An

[124] Next fit is called "first fit" in their paper, as is common.

[125] This is allocator (implemented by Chris Kingsley and widely distributed with the BSD 4.2 UNIX system) is called a buddy system in their paper, but it is not; it does no coalescing at all.

[126] This is actually intended to test a test system; faulty designs are intentionally included in the set, and should be weeded out by the test system. If not, the test system must be improved.

interesting characteristic of this kind of system is that memory usage follows a backward (decreasing) ramp function after the initialization phase—aside from short-term variations due to short-lived objects, the general shape of the memory-use function is monotonically decreasing.

To test their allocator, they used a synthetic workload where memory usage rises sharply at the beginning and oscillates around a linearly descending ramp. The use of this synthetic trace technique is more somewhat more reasonable for this specialized allocator than for the general allocation problem; since there's essentially no external fragmentation,[127] there's little difference between a real trace and a synthetic one in that regard.

They reported that this quick list technique was quite fast, relative to the (unspecified) general allocator.

From our point of view, we find the experimental results less interesting than the explanation of the overall pattern of memory usage in this class of application, and what the attractiveness of this approach indicates about the state of heap management in the real world (refer to Section 1.1).

Page and Hagins [PH86] provided the first published double buddy system, and experimentally compared it to binary and weighted buddy systems. Using the standard simulation techniques, and only uniformly distributed sizes and lifetimes, they show that double buddies suffer from somewhat less fragmentation than binary and weighted buddies. They also present an analysis that explains this result.[128]

Brent [Bre89] presented a scalable algorithm for the address-ordered first fit policy, using a "heap," data structure—i.e., a partially-ordered tree, not to be confused with the sense of "heap" as a pool for dynamic storage allocation—embedded in an array. To keep the size of this heap array small, a two-level scheme is used. Memory is divided into equal-sized chunks, and the heap recorded the size of the largest free block in each chunk. Within a chunk, conventional linear searching is used. While this scheme appears to scale well, it has the drawback that the constant factors are apparently rather high. Other scalable indexing schemes may provide higher performance for address-ordered first fit.

> Although the villagers had forgotten the ritual and lost the original black box, they still remembered to use stones...
> "It isn't fair, it isn't right," Mrs. Hutchison screamed and then they were upon her.
> —*Shirley Jackson*, "The Lottery"

Coffman and Leighton, in a paper titled "A Provably Efficient Algorithm for Dynamic Storage Allocation" [CL89] describe an algorithm combining some characteristics of best fit and address-ordered first fit, and prove that its memory usage is asymptotically optimal as system size increases toward infinity.

To enable this proof, they make the usual assumptions of randomness and independence, including randomly ordered and exponentially distributed lifetimes. (See Section 2.2.) They also make the further assumption that the distribution of object sizes is known *a priori*, which is generally not the case in real systems.

Coffman and Leighton say that probabilistic results are less common than worst-case results, "but far more important," that their result has "strong consequences for practical storage allocation systems," and that algorithms designed to "create sufficiently large holes when none exist will not be necessary except in very special circumstances."

It should be no surprise that we feel compelled to take exception with such strongly-stated claims. In our view, the patterned time-varying nature of real request streams is the major problem in storage allocation, and in particular the time-varying shifts in the requested sizes. Assuming that request distributions are known and stable makes the problem mathematically tractable, but considerably less relevant.

Coffman and Leighton offer an asymptotic improvement in memory usage, but this amounts to no more than a small constant factor in practice, since real algorithms used in real systems apparently seldom

[127] Since memory usage is dominated by a single size, almost all requests can be satisfied by almost any free block;

[128] While we believe that double buddies are indeed effective, we disagree somewhat with their methodology and their analysis. Uniform random distributions do not exhibit the skewed and non-uniform size distributions often seen in real programs, or pronounced phase behavior. All of these factors may affect the performance of the double buddy system; a skew towards a particular size favors double buddies, where splitting always results in same-sized free blocks. Phase behavior may enhance this effect, but on the other hand may cause problems due to uneven usage of the two component (binary) buddy systems, causing external fragmentation.

waste more than a factor of two in space, and usually much less.[129]

While we believe that this result is of limited relevance to real systems, it does seem likely that for extremely large systems with many complex and independent tasks, there may be significant smoothing effects that tend in this direction. In that case, there may be very many effectively random holes, and thus a likely good fit for any particular request.

Unfortunately, we suspect that the result given is not directly relevant to any existing system, and for any sufficiently large and complex systems, other considerations are likely to be more important. For the foreseeable future, time-varying behavior is the essential policy consideration. If systems eventually become ver large (and heterogeneous), locality concerns are likely to be crucial. (Consider the effects on locality in a large system when objects are placed in effectively randomly-generated holes; the scattering of related data seems likely to be a problem.)

Hanson [Han90] presents a technique for allocating objects and deallocating them *en masse*. This is often more efficient and convenient than traversing data structures being deallocated, and freeing each object individually. A special kind of heap can be created on demand. In the GNU C compiler system, these are called "obstacks," short for "object stacks," and we will adopt that term here. Objects known to die at the end of a phase can be allocated on an obstack, and all freed at once when the phase is over. More generally, nested phases are supported, so that objects can be deallocated in batches whose extents are nested. Freeing an object simply frees that object and all objects allocated after it. (This is actually a very old idea, dating at least to Collins' "zone" system.[130] The fact that this idea has been independently developed by a variety of system implementors attests to the obvious and exploitable phase behavior evident in many programs.)

The obstack scheme has two advantages. First, it is often easier for the programmer to manage batches of objects than to code freeing routines that free each object individually. Second, the allocator implementation can be optimized for this usage style, reducing space and time costs for freeing objects. In Hanson's system, storage for a specially-managed heap is allocated as a linked list of large chunks, and objects can be allocated contiguously within a chunk; no header is required on each small object. The usual time cost for allocation is just the incrementing of a pointer into a chunk, plus a check to see if the chunk is full. The time cost for freeing in a large specially-managed heap is roughly proportional to the number of chunks freed, with fairly small constant factors, rather than the number of small objects freed.

Obstack allocation must be used very carefully, because it intertwines the management of data structures with the control structure of a program. It is easy to make mistakes where objects are allocated on the obstack, but the data objects they manage are allocated on the general heap. (E.g., a queue object may be allocated on an obstack, but allocate its queue nodes on the general heap.) When the controlling objects are freed, the controlled objects are not; this is especially likely to happen in large systems, where intercalling libraries do not obey the same storage management conventions.[131]

[129] We also note that their algorithm requires $\log_2 n$ time—where n is the number of free blocks—which tends toward infinity as n tends toward infinity. In practical terms, it becomes rather slow as systems become very large. However, more scalable (sublogarithmic) algorithms could presumably exploit the same statistical tendencies of very large systems, if real workloads resembled stochastic processes.

[130] Similar techniques have been used in Lisp systems (notably the Lisp Machine systems), and are known by a variety of names.

[131] The opposite kind of mistake is also easy to make, if the controlling objects' routines are coded on the assumption that the objects it controls will be freed automatically when it is freed, but the controlling object is actually allocated on the general heap rather than an obstack. In that case, a storage leak results. These kinds of errors (and many others) can usually be avoided if garbage collection [Wil95] is used to free objects automatically. Henry Baker reports that the heavy use of an obstack-like scheme used in MIT Lisp machines was a continuing source of bugs (Baker, personal communication 1995). David Moon reports that a similar facility in the Symbolics system often resulted in obscure bugs, and its use was discouraged after an efficient generational garbage collector [Moo84] was developed (Moon, personal communication 1995); generational techniques heuristically exploit the lifetime distributions of typical programs [LH83, Wil95]. For systems without garbage collection, however, the resulting problems may be no worse than those introduced by other explicit deallocation strategies, when used carefully and in well-documented ways.

4.2 Recent Studies Using Real Traces

> "Some places have already quit lotteries," Mrs. Adams said.
> "Nothing but trouble in *that*," Old Man Warner said stoutly.
> —*Shirley Jackson*, "The Lottery"

Zorn, Grunwald, et al. Zorn and Grunwald and their collaborators have performed a variety of experimental evaluations of allocators and garbage collectors with respect to space, time, and locality costs. This is the first major series of experiments using valid methodology, i.e., using real traces of program behavior for a variety of programs.

Our presentation here is sketchy and incomplete, for several reasons. Zorn and Grunwald are largely interested in time costs, while we are (here) more interested in placement policies' effect on fragmentation. They have often used complicated hybrid allocator algorithms, making their results difficult to interpret in terms of our basic policy consideration, and in general, they do not carefully separate out the effects of particular implementation details (such as per-object overheads and minimum block sizes) from "true" fragmentation. (Nonetheless, their work is far more useful than most prior experimental work.) Some of Zorn and Grunwald's papers—and much of their data and their test programs—are available via anonymous Internet FTP (from `cs.colorado.edu`) for further analysis and experimentation.

In [ZG92], **Zorn and Grunwald** present various allocation-related statistics on six allocation-intensive C programs, i.e., programs for which the speed of the allocator is important. (Not all of these use large amounts of memory, however.) They found that for each of these programs, the two most popular sizes accounted for at least half (and as much as 93%) of all allocations. In each, the top ten sizes accounted for at least 85% of all allocations.

Zorn and Grunwald [ZG94] attempted to find fairly conventional models of memory allocation that would allow the generation of synthetic traces useful for evaluating allocators. They used several models of varying degrees of sophistication, some of which modeled phase behavior and one of which modeled fine-grained patterns stochastically (using a first-order Markov model). To obtain the relevant statistics, they gathered real traces and analyzed them to quantify various properties, then constructed various drivers using pseudo-random numbers to generate request streams accordingly.

In general, the more refined attempts at modeling real behavior failed. (Our impression is that they did not necessarily expect to succeed—their earlier empirical work shows a strong disposition toward the use of real workloads.) They found that their most accurate predictor was a simple "mean value" model, which uses only the mean size and lifetime, and generates a request stream with uniformly distributed sizes and lifetimes. (Both vary from zero to twice the mean, uniformly.) Unfortunately, even their best model is not very accurate, exhibiting errors of around 20%. For a small set of allocators, this was sufficient to predict the rank ordering (in terms of fragmentation) in most cases, but with ordering errors when the allocators were within a few percent of each other.

From this Zorn and Grunwald conclude that the only reliable method currently available for studying allocators is trace-driven simulation with real traces. While this result has received too little attention, we believe that this was a watershed experiment, invalidating most of the prior experimental work in memory allocation.

Ironically, Zorn and Grunwald's results show that some of the most simplistic models—embodying clearly false assumptions of uniform size and lifetime distributions—generally produce more accurate results than more "realistic" models. It appears that some earlier results using unsound methods have obtained the right results by sheer luck—the "better" algorithms do in fact tend to work better for real programs behavior as well. (Randomization introduces biases that tend to cancel each other out for most policies tested in earlier work.) The errors produced are still large, however, often comparable to the total fragmentation for real programs, once various overheads are accounted for.

(Our own later experiments [WJNB95], described later, show that the random trace methodology can introduce serious and systematic errors for some allocators which are popular in practice but almost entirely absent in the experimental literature. This is ironic as well—earlier experimenters happened to choose a combination of policies and experimental methodology that gave some of the right answers. It is clear from our review of the literature that there was–and still is—no good model that predicts such a happy coincidence.)

Zorn, Grunwald, and Henderson [GZH93] measured the locality effects of several allocators: next fit, the G++ segregated fits allocator by Doug Lea, simple segregated storage using powers of two size classes (the Berkeley 4.2 BSD allocator by Chris Kingsley), and two simplified quick fit schemes (i.e., "Quick Fit" in the sense of [WW88], i.e., without coalescing for small objects).

One of simplified these quick fit allocators (written by Mike Haertel) uses first fit as the general allocator, and allocates small objects in powers-of-two sized blocks. (We are not sure which variant of first fit is used.) As an optimization, it stores information about the memory use within page-sized (4KB) chunks and can reclaim space for entirely empty pages, so that they can be reused for objects of other sizes. It can also use the pagewise information in an attempt to improve the locality of free list searches.

The other simplified quick fit allocator is uses the G++ segregated fits system as its general allocator, and uses quick lists for each size, rounded to the nearest word, up to 8 words (32 bytes).

Using Larus' QP tracing tool [BL92], Zorn et al. traced five C programs combined with their five allocators, and ran the traces through virtual memory and cache simulators.

They found that next fit had by far the worst locality, and attribute this to the roving pointer mechanism—as free list searches cycle through the free list, they may touch widely separated blocks only once per cycle. We suspect that there is more to it than this, however, and that the poor locality is also due to the effects of the free list *policy*; it may intersperse objects belonging to one phase among objects belonging to others as it roves through memory.

Because of the number of variables (use of quick lists, size ranges of quick lists, type of general allocator, etc.), we find the other results of this study difficult to summarize. It appears that the use of coarse size ranges degrades locality, as does excessive per-object overhead due to boundary tags. (The version of Lea's allocator they used had one-word footers as well as one-word headers; we have since removed the footers.) FIFO-managed segregated lists promote rapid reuse of memory, improving locality at the small granularities relevant to cache memories. Effects on larger-scale locality are less clear.

Barrett and Zorn [BZ93] present a very interesting scheme for avoiding fragmentation by heuristically segregating short-lived objects from other objects. Their "lifetime prediction" allocator uses offline profile information from "training" runs on sample data to predict which call sites will allocate short-lived objects. During normal (non-training) runs, the allocator examines the procedure call stack to distinguish between different patterns of procedure calls that result in allocations. Based on profile information, it predicts whether the lifetimes of objects created by that call pattern can be reliably predicted to be short. (This is essentially a refinement of a similar scheme used by Demers *et al.* for lifetime prediction in a garbage collector; that scheme [DWH+90] uses only the size and stack pointer, however, not the call chain.)

For five test applications, Barrett and Zorn found that examining the stack to a depth of four calls generally worked quite well, enabling discrimination between qualitatively different patterns that result in allocations from the same allocator call site.

Their predictor was able to correctly predict that 18% to 99% of all allocated bytes would be short-lived. (For other allocations, no prediction is made; the distinction is between "known short-lived" and "don't know.") While we are not sure whether this is the best way of exploiting regularities in real workloads,[132] it certainly shows that exploitable regularities exist, and that program behavior is not random in the manner assumed (implicitly or explicitly) by earlier researchers. (Barrett and Zorn found that using only the requested size was less predictive, but still provided useful information.)

Zorn and Grunwald [GZ93] have investigated the tailoring of allocators to particular programs, primarily to improve speed without undue space cost. One important technique is the use of inlining (incorporating the usual-case allocator code at the point of call, rather than requiring an out-of-line call to a subroutine). The judicious use of inlining, quick lists for the important size classes, and a general coalescing backing allocator appears to be able to provide excellent speed with reasonable memory costs.

Another useful empirical result is that when programs are run on different data sets, they typically allocate the same sizes in roughly similar proportions—the most important size classes in one run are likely to be the most important size classes in another, allowing offline tailoring of the algorithm using profile data.

[132] As noted in Section 2.4, we suspect that death time discrimination is easier than lifetime prediction.

Vo. In a forthcoming article, Vo reports on the design of a new allocator framework and empirical results comparing several allocators using real traces [Vo95]. (Because this is work in progress, we will not report the empirical results in detail.)

Vo's `vmalloc()` allocator is conceptually similar to Ross' zone system, allowing different "regions" of memory to be managed by different policies.[133] (Regions are subsets of the overall heap memory, and are not contiguous in general; to a first approximation, they are sets of pages.) A specific allocator can be chosen at link time by setting appropriate UNIX environment variables. This supports experimentation with different allocators to tune memory management to specific applications, or to different parts of the same application, which may allocate in zones that are managed differently. Various debugging facilities are also provided.

The default allocator provided by Vo's system is a deferred coalescing scheme using best fit for the general allocator. (The size ordering of blocks is maintained using a splay tree.) In comparisons with several other allocators, this allocator is shown to be consistently among the fastest and among the most space efficient, for several varied test applications.

Wilson, Johnstone, Neely, and Boles. In a forthcoming report [WJNB95], we will present results of a variety of memory allocation experiments using real traces from eight varied C and C++ programs, and more than twenty variants of six general allocator types (first fit, best fit, next fit, buddy systems, and simple segregated storage) [WJNB95]. We will briefly describe some of the major results of that study here.

To test the usual experimental assumptions, we used both real and synthetic traces, and tried to make the synthetic traces as realistic as possible in terms of size and lifetime distributions. We then compared results of simulations using real traces with those from randomly-ordered traces. (To generate the random traces, we simply "shuffled" the real traces, preserving the size and lifetime distributions much more accurately than most synthetic trace generation schemes do.) We found that there was a significant correlation between the results from real traces and those from shuffled traces, but there were major and systematic errors as well. In an initial test of eight varied allocators, the correlations accounted for only about a third of the observed variation in performance. This shows that the random ordering of synthetic traces *discards the majority of the information relevant to estimating real fragmentation.* Results from most of pre-1992 experiments are therefore highly questionable.

Using real traces, we measured fragmentation for our eight programs using our large set of allocators. We will report results for the twelve we consider most interesting here; for more complete and detailed information, see the forthcoming report [WJNB95]. These allocators are best fit (using FIFO-ordered free lists[134]), first fit (using LIFO-ordered, FIFO-ordered and address-ordered free lists), next fit (also using LIFO, FIFO, and address order), Lea's segregated fits allocator, binary and double buddy systems, simple segregated storage using powers-of-two size classes, and simple segregated storage using twice as many size classes (powers of two, and three times powers of two, as in the weighted buddy system).

We attempted to control as many implementation-specific costs as possible. In all cases, objects were aligned on double-word (eight-byte) boundaries, and the minimum block size was four words. Fragmentation costs will be reported as a percentage increase, relative to the baseline of the number of actual bytes of memory devoted to program objects at the point of maximum memory usage. All allocators had one-word headers, except for the simple segregated storage allocators, which had no headers.[135] (As explained earlier, we believe that in most systems, these will be the usual header sizes for well-implemented allocators of these types.)

We will summarize fragmentation costs for twelve allocators, in increasing order of space cost. We note that some of these numbers may change slightly before [WJNB95] appears, due to minor changes in our

[133] See also Delacour's [Del92] and Attardi's [AF94] sophisticated systems for low-level storage management in (mostly) garbage-collected systems using mixed languages and implementation strategies.

[134] No significant differences were found between results for variations of best fit using different free list orders. This is not too surprising, given that the best fit policy severely restricts the choice of free blocks.

[135] Rather than varying the actual implementations' header and footer schemes, we simulated different header sizes by compensating at allocation time and in our measurements. The sequential fits, segregated fits, and simple segregated storage allocators actually use two-word headers or one word headers and one word footers, but we reduced the request sizes by one word at allocation time to "recover" one of those words by counting it as available to hold a word of an object.

experiments. The nubers for next fit are also somewhat suspect—we are currently trying to determine whether they are affected by a failure to respect Korn and Vo's wilderness preservation heuristic.[136]

It should also be noted that our experimental methodology could introduce errors on the order of a percent or two. Worse, we found that the variance for some of these allocators was quite high, especially for some of the poorer algorithms. (We are also concerned that any sample of eight programs cannot be considered representative of all real programs, though we have done our best [WJNB95].) The rank ordering here should thus be considered very approximate, especially within clusters.

To our great surprise, we found that best fit, address-ordered first fit, and FIFO-ordered first fit all performed extremely well—and nearly identically well. All three of these allocators had only about 22% fragmentation, including losses due to header costs, rounding up for doubleword alignment, and rounding small block sizes up to four words.

They were followed by a cluster containing address-ordered next fit, segregated fits, and FIFO-ordered next fit at 28%, 31% and 32%. Then came a cluster consisting of LIFO-ordered first fit, double buddy, and LIFO-ordered next fit, and at 54%, 56%, and 59%. These were followed by a cluster consisting of simple segregated storage using closely-spaced size classes (73%) and binary buddy (74%). Simple segregated storage using powers-of-two sizes came last, at 85%.

For first fit and next fit, we note that the LIFO free list order performed far worse than the FIFO free list order or the address order. For many programmers (including us), LIFO ordering seems most natural; all other things being equal, it would also appear to be advantageous in terms of locality. Its fragmentation effects are severe, however, typically increasing fragmentation by a factor of two or three relative to either address-order or FIFO-order. We are not sure why this is; the main characteristic the latter two seem to have in common is deferred reuse. It may be that a deferred reuse strategy is more important than the details of the actual policy. If so, that suggests that a wide variety of policies may have excellent memory usage. This is encouraging, because it suggests that some of those policies may be amenable to very efficient and scalable implementations.

Double buddy worked as it was designed to—if we assume that it reduced internal fragmentation by the expected (approximate) 14%, it seems that the dual buddy scheme did not introduce significant external fragmentation—relative to binary buddies—as Fibonacci and weighted schemes are believed to do. Still, its performance was far worse than that of the best allocators.

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%. (This is comparable to the loss we expect just from the double word alignment and minimum block sizes.)

While the rankings of best fit and address-ordered first fit are similar to results obtained by random-trace methods, we found them quite surprising, due to the evident methodological problems of random-trace studies. We know of no good model to explain them.[137]

While the three excellent allocators fared well with both real and randomized traces, other allocators fared differently in the two sets of simulations. The segregated storage schemes did unrealistically well, relative to other allocators, when traces were randomized.

The results for randomized traces show clearly that size and lifetime distributions are not sufficient to predict allocator performance for real workloads. The ordering information interacts with the allocator's policies in ways that are often more important than the distributions alone. Some of these results were not unexpected, given our understanding on the methodology. For example, the unrealistically good performance of simple segregated fits schemes relative to the others was expected, because of the smoothing effect of random walks—synthetic traces tend not to introduce large amounts of external fragmentation, which is the Achilles' heel of non-splitting, non-coalescing policies.

Like Zorn and Grunwald, we will make the test pro-

[136] Most of the allocators appear fairly insensitive to this issue, and the others (our first fit and best fit) were designed to respect it by putting the end block at the far end of the free list from the search pointer.

[137] We have several just-so stories that could explain them, of course, but we haven't yet convinced ourselves that any of them are true.

grams we used available for others to use for replication of our results and for other experiments.[138]

5 Summary and Conclusions

> "[People refused to believe that the world was round] because it *looked like* the the world was flat."
> "What would it have looked like if it had looked like the world was round?"
> —*attributed to Ludwig Wittgenstein*

There is a very large space of possible allocator policies, and a large space of mechanisms that can support them. Only small parts of these spaces have been explored to date, and the empirical and analytical techniques used have usually produced results of dubious validity.

There has been a widespread failure to recognize anomalous data as undermining the dominant paradigm, and to push basic causal reasoning through—to recognize what data *could* be relevant, and what other theories might be consistent with the observed facts. We find this curious, and suspect it has two main causes.

One cause is simply the (short) history of the field, and expectations that computer science issues would be easily formalized, after many striking early successes. (Ullman [Ull95] eloquently describes this phenomenon.)

Another is doubtless the same kind of paradigm entrenchment that occurs in other, more mature sciences [Kuh70]. Once the received view has been used as a theoretical underpinning of enough seemingly successful experiments, and reiterated in textbooks without the caveats buried in the original research papers, it is very hard for people to see the alternatives.

The history of memory allocation research may serve as a cautionary tale for empirical computer science. Hartmanis has observed that computer science seems less prone to paradigm shifts than most fields [Har95]. We agree in part with this sentiment, but the successes of computer science can lead to a false sense of confidence. Computer scientists often have less to worry about in terms of the validity of "known" results, relative to other scientists, but in fact they often *worry less about it*, which can be a problem, too.

5.1 Models and Theories

There has been a considerable amount of theoretical work done in the area of memory allocation—if we use "theory" in the parlance of computer science, to mean a particular subdiscipline using particular kinds of logical and mathematical analyses. There has been very little theoretical work done, however, if we use the vernacular and central sense of "theory," i.e., what everyday working scientists do.

We simply have no theory of program behavior, much less a theory of how allocators exploit that behavior. (Batson made similar comments in 1976, in a slightly different context [Bat76], but after nearly two decades the situation is much the same.)

Aside from several useful studies of worst-case performance, most of the analytical work to date seems to be based on several assumptions that turn out to be incorrect, and the results cannot be expected to apply directly to the real problems of memory allocation.

Like much work in mathematics, however, theoretical results may yet prove to be enlightening. To make sense of these results and apply them properly will require considerable thought, and the development of a theory in the vernacular sense.

For example, the striking similarities in performance between best fit and address-ordered first fit for randomized workloads should be explained. How is it that such different policies are so comparable, for an essentially unpredictable sequence of requests? More importantly, how does this relate to real request sequences? The known dependencies of these algorithms on lifetime distributions should also be explained more clearly. Randomization of input order may eliminate certain important variables, and allow others to be explored more or less in isolation. On the other hand, interactions with real programs may be so systematically different that these phenomena have nothing important in common—for example, dependence on size distributions may be an effect that has little importance in the face of systematic interactions between placement policy and phase behavior.

Understanding real program behavior still remains the most important first step in formulating a theory of memory management. Without doing that, we

[138] Our anonymous FTP repository is on `ftp.cs.utexas.edu` in the directory `pub/garbage`. This repository also contains the BibTeX bibliography file used for this paper and [Wil95], several papers on persistence and memory hierarchies, and numerous papers on garbage collection by ourselves and others.

cannot hope to develop the science of memory management; we can only fumble around doing ad hoc *engineering*, in the too-often-used pejorative sense of the word. At this point, the needs of good science and of good engineering in this area are the same—a deeper *qualitative* understanding. We must try to discern what is relevant and characterize it; this is necessary before formal techniques can be applied usefully.

5.2 Strategies and Policies

Most policies used by current allocators are derived fairly straightforwardly from ideas that date from the 1960's, at least. Best fit and address-ordered first fit policies seem to work well in practice, but after several decades the reasons why are not much clearer than they were then. It is not clear which regularities in real request streams they exploit. (It is not even very clear how they exploit regularities in synthetic request streams, where the regularities are minimal and presumably much easier to characterize.) Because our current understanding of these issues is so weak, we will indulge in some speculation.

Given that there is no reason to think that these early policies were so well thought out that nothing could compete with them, it is worthwhile to wonder whether there is a large space of possible policies that work at least as well as these two. Recent results for FIFO-ordered sequential fits may suggest that close fits and address ordering are not crucial for good performance.

It may well be that the better allocators perform well because it's *very easy* to perform well. Program behavior may be so patterned and redundant (in certain relevant ways) that the important regularities in request streams are trivial to exploit. The known good policies may only be correlated to some more fundamental strategy—or combination of strategies—yet to be discovered.

Given the real and striking regularities in request streams due to common programming techniques, it seems likely that better algorithms could be designed if we only had a good *model* of program behavior, and a good understanding of how that interacts with allocation policies. Clustered deaths due to phase behavior, for example, suggest that contiguous allocation of consecutively-allocated blocks may tend to keep fragmentation low. (It probably has beneficial effects on locality as well.)

Segregation of different kinds of objects may avoid fragmentation due to differing death times of objects used for different purposes. (Again, this may increase locality as well—by keeping related objects clustered after more ephemeral objects have been deallocated.)

On the other hand, it is possible that the regularities exploited by good existing allocators are so strong and simple that we cannot improve memory usage by much—it's possible that all of our best current algorithms exploit them to the fullest, however accidentally. The other patterns in program behavior may be so subtle, or interact in such complex ways, that no strategy can do much better. Or it may turn out that once the regularities are understood, the task of exploiting them online is just too expensive. (That doesn't seem likely to us, though some intermediate situation seems plausible.)

If all else fails, relying best fit and first fit usually won't be a disaster, as long as the mechanisms used are scalable. (If one of them doesn't work well for your program, it's likely that the other will—or that some other simple policy will suffice.)

On the other hand, it is not clear that our best policies are robust enough to count on—so far, only a few experiments have been performed to asses the interactions between real program behavior and allocator policies. It is entirely possible that there is a non-negligible percentage of programs for which our "best" algorithms will fail miserably.

5.3 Mechanisms

Many current allocator policies are partly artifacts of primitive implementation techniques—they are mostly based on obvious ways of managing linear lists. Modern data structure techniques allow us to build much more sophisticated indexing schemes, either to improve performance or support better-designed policies.

Segregated fits and (other) indexing schemes can be used to implement policies known to work well in practice, and many others. More sophisticated indexing schemes will probably allow us to exploit whatever exploitable regularities we are clever enough to characterize, in a scalable way.

Deferred coalescing allows optimization of common patterns of short-term memory use, so that scalable mechanisms don't incur high overheads in practice. The techniques for deferred coalescing must be studied carefully, however, to ensure that this mechanism doesn't degrade memory usage unacceptably by changing placement policies and undermining strategies.

5.4 Experiments

New experimental methods must be developed for the testing of new theories. Trace-driven simulations of real program/allocator pairs will be quite important, of course—they are an indispensable reality check. These trace-driven simulations should include locality studies as well as conventional space and time measurements. Sound work of both sorts has barely begun; there is a lot to do.

If we are to proceed scientifically, however, just running experiments with a grab-bag of new allocators would may be doing things backwards. Program behavior should be studied in (relative) isolation, to identifying the fundamental regularities that are relevant to to various allocators and memory hierarchies. After that, it should be easier to design strategies and policies intelligently.

5.5 Data

Clearly, in order to formulate useful theories of memory management, more data are required. The current set of programs used for experimentation is not large enough or varied enough to be representative.

Some kinds of programs that are not represented are:

- *scientific computing* programs (especially those using sophisticated sparse matrix representations),
- *long-running system programs* such as operating system kernels, name servers, file servers, and graphics display servers,
- *business data analysis* programs such as spreadsheets, report generators, and so on,
- *graphical programs* such as desktop publishing systems, CAD interaction servers and interactive 3-D systems (e.g., virtual reality),
- *interactive programming environments* with source code management systems and interactive debugging facilities,
- *heavily object-oriented programs* using sophisticated kits and frameworks composed in a variety of ways,
- *automatically-generated programs* of a variety of types, created using specialized code-generation systems or compilers for very-high-level languages.

This partial list is just a beginning—there are many kinds of programs, written in a variety of styles, and test application suites should include as many of them as possible.

There are some difficulties in obtaining and using such programs that can't be overlooked. The first is that the most easily obtainable programs are often not the most representative—freely available code is often of a few types, such as script language interpreters, which do not represent the bulk of actual computer use, particularly memory use.

Those programs that are available are often difficult to analyze, for various reasons. Many used hand-optimized memory allocators, which must be removed to reveal the "true" memory usage—and this "true" memory usage itself may be skewed by the awkward programming styles used to avoid general heap allocation.

5.6 Challenges and Opportunities

> Computer Science and Engineering is a field that attracts a different kind of thinker... Such people are especially good at dealing with situations where different rules apply in different cases; they are individuals who can rapidly change levels of abstraction, simultaneously seeing things "in the large" and "in the small."
> —*Donald Knuth*, quoted in [Har95]

Memory management is a fundamental area of computer science, spanning several very different levels of abstraction—from the programmer's strategies for dealing with data, language-level features for expressing those concepts, language implementations for managing actual storage, and the varied hardware memories that real machines contain. Memory management is where the rubber meets the road—if we do the wrong thing at any level, the results will not be good. And if we don't make the levels work well *together*, we are in serious trouble. In many areas of computer science, problems can be decomposed into levels of abstraction, and different problems addressed at each level, in nearly complete isolation. Memory management requires this kind of thinking, but that is not enough—it also requires the ability to reason about phenomena that span multiple levels. This is not easy.

Unfortunately, the compartmentalization of computing disciplines has discouraged the development of a coherent memory management community. Memory management tends to be an orphan, sometimes

harbored by the programming language community, sometimes by the operating systems community—and usually ignored by the architecture community.

It seems obvious that memory management policies can have a profound impact on locality of reference, and therefore the overall performance of modern computers, but in the architecture community locality of reference is generally treated as a mysterious, incomprehensible substance. (Or maybe two or three substances, all fairly mysterious.) A program is pretty much a black box, however abraded and splintered, and locality comes out of the box if you're lucky. It is not generally recognized that different memory management policies can have an effect on memory hierarchies that is sometimes as significant as differences in programs' intrinsic behavior. Recent work in garbage collection shows this to be true ([WLM92, Wil95, GA95]), but few architects are aware of it, or aware that similar phenomena must occur (to at least some degree) in conventionally-managed memories as well [GZH93].

The challenge is to develop a theory that can span all of these levels. Such a theory will not come all at once, and we think it is unlikely to be primarily mathematical, at least not for a long time, because of the complex and ill-defined interactions between different phenomena at different levels of abstraction.

Computer science has historically been biased toward the paradigms of mathematics and physics—and often a rather naive view of the scientific process in those fields—rather than the "softer" natural sciences. We recommend a more naturalistic approach [Den95], which we believe is more appropriate for complex multilevel systems that are only partly hierarchically decomposable.

The fact that fact that we study mostly deterministic processes in formally-describable machines is sometimes irrelevant and misleading. The degrees of complexity and uncertainty involved in building real systems require that we examine real data, theorize carefully, and keep our eyes open.

Computer science is often a very "hard" science, which develops along the lines of the great developments in the physical sciences and mathematics the seventeenth, eighteenth and nineteenth centuries. It owes a great deal to the examples set by Newton and Descartes. But the nineteenth century also saw a very great theory that was tremendously important without being formalized at all—a theory that to this day can only be usefully formalized in special, restricted cases, but which is arguably the single most important scientific theory ever. Perhaps we should look to Darwin as an examplar, too.

Acknowledgements

We would like to thank Hans Boehm and especially Henry Baker for many enlightening discussions of memory management over the last few years, and for comments on earlier versions of this paper.

Thanks to Ivor Page, for comments that seem to connect important pieces of the puzzle more concretely than we expected, and to Ben Zorn, Dirk Grunwald and Dave Detlefs for making their test applicatons available.

Thanks also to Dave Barrett, Sheetal Kakkad, Doug Lea, Doug McIlroy, and Phong Vo for comments that have improved our understanding and presentation, and to Henry Baker and Janet Swisher for their help and extraordinary patience during the paper's preparation. (Of course, we bear sole responsibility for any opinions and errors.)

References

[Abr67] John Abramowich. Storage allocation in a certain iterative process. *Communications of the ACM*, 10(6):368–370, June 1967.

[AF94] G. Attardi and T. Flagella. A customizable memory management framework. In *Proceedings of the USENIX C++ Conference*, Cambridge, Massachussetts, 1994.

[AS95] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, 13(2):95–121, May 1995.

[Bae73] H. D. Baecker. Aspects of reference locality in list structures in virtual memory. *Software Practice and Experience*, 3(3):245–254, 1973.

[Bak93] Henry G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, April 1993.

[BAO85] B. M. Bigler, S. J. Allan, and R. R. Oldehoeft. Parallel dynamic storage allocation. In *1985 International Conference on Parallel Processing*, pages 272–275, 1985.

[Bat76] Alan Batson. Program behavior at the symbolic level. *IEEE Computer*, pages 21–26, November 1976.

[Bay77] C. Bays. A comparison of next-fit, first-fit and best-fit. *Communications of the ACM*, 20(3):191–192, March 1977.

[BB77] A. P. Batson and R. E. Brundage. Segment sizes and lifetimes in ALGOL 60 programs. *Communications of the ACM*, 20(1):36–44, January 1977.

[BBDT84] G. Bozman, W. Buco, T. P. Daly, and W. H. Tetzlaff. Analysis of free storage algorithms—revisited. *IBM Systems Journal*, 23(1):44–64, 1984.

[BC79] Daniel G. Bobrow and Douglas W. Clark. Compact encodings of list structure. *ACM Transactions on Programming Languages and Systems*, 1(2):266–286, October 1979.

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

[BCW85] B. S. Baker, E. G. Coffman, Jr., and D. E. Willard. Algorithms for resolving conflicts in dynamic storage allocation. *Journal of the ACM*, 32(2):327–343, April 1985.

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

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

[Ben81] V. E. Benes. Models and problems of dynamic storage allocation. In *Applied Probability and Computer Science—the Interface*. Institute of Management Science and Operations Research Society of America, January 1981.

[Bet73] Terry Betteridge. An analytical storage allocation model. *Acta Informatica*, 3:101–122, 1973.

[Bet82] Terry Betteridge. *An Algebraic Analysis of Storage Fragmentation*. UMI Research Press, Ann Arbor, Michigan, 1982.

[BJW70] A. P. Batson, S. M. Ju, and D. C. Wood. Measurements of segment size. *Communications of the ACM*, 13(3):155–159, March 1970.

[BL92] Ball and Larus. Optimal profiling and tracing of programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70. ACM Press, January 1992.

[Boz84] Gerald Bozman. The software lookaside buffer reduces search overhead with linked lists. *Communications of the ACM*, 27(3):222–227, March 1984.

[BR64] Daniel G. Bobrow and Bertram Raphael. A comparison of list-processing computer languages. *Communications of the ACM*, 7(4):231–240, April 1964.

[Bre89] R. Brent. Efficient implementation of the first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, July 1989.

[Bro80] A. G. Bromley. Memory fragmentation in buddy methods for dynamic storage allocation. *Acta Informatica*, 14(2):107–117, August 1980.

[Bur76] Warren Burton. A buddy system variation for disk storage allocation. *Communications of the ACM*, 19(7):416–417, July 1976.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[BZ93] David A. Barrett and Bejamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation* [PLD93], pages 187–196.

[BZ95] David A. Barrett and Benjamin G. Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation*, pages 301–314, La Jolla, California, June 1995. ACM Press.

[Cam71] J. A. Campbell. A note on an optimal-fit method for dynamic allocation of storage. *Computer Journal*, 14(1):7–9, February 1971.

[CG91] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level file system. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 200–209, Santa Clara, California, April 1991.

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

[CKS85] E. G. Coffman, Jr., T. T. Kadota, and L. A. Shepp. On the asymptotic optimality of first-fit storage allocation. *IEEE Transactions on Software Engineering*, SE-11(2):235–239, February 1985.

[CL89] E. G. Coffman, Jr. and F. T. Leighton. A provably efficient algorithm for dynamic storage allocation. *Journal of Computer and System Sciences*, 38(1):2–35, February 1989.

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

[Com64] W. T. Comfort. Multiword list items. *Communications of the ACM*, 7(6), June 1964.

[CT75] B. Cranston and R. Thomas. A simplified recombination scheme for the Fibonacci buddy system. *Communications of the ACM*, 18(6):331–332, July 1975.

[Dar59] Charles Darwin. *The Origin of Species*. 1859.

[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.

[DEB94] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically typed languages. Technical Report 400, Indiana University Computer Science Dept., March 1994.

[Del92] V. Delacour. Allocation regions and implementation contracts. In Bekkers and Cohen [BC92], pages 426–439.

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

[Den95] Daniel Dennett. *Darwin's Dangerous Idea*. 1995.

[Det92] David L. Detlefs. Garbage collection and runtime typing as a C++ library. In *USENIX C++ Conference*, Portland, Oregon, August 1992. USENIX Association.

[Dij69] Edsger W. Dijkstra. Notes on structured programming. In *Structured Programming*. Academic Press, 1969.

[Dou93] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, January 1993.

[DTM93] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation. Submitted for publication, August 1993.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, California, January 1990. ACM Press.

[EO88] C. S. Ellis and T. J. Olson. Algorithms for parallel memory allocation. *International Journal of Parallel Programming*, 17(4):303–345, 1988.

[Fer76] H. R. P. Ferguson. On a generalization of the Fibonacci numbers useful in memory allocation schema. *The Fibonacci Quarterly*, 14(3):233–243, October 1976.

[For88] R. Ford. Concurrent algorithms for real-time memory management. *IEEE Software*, pages 10–23, September 1988.

[FP74] J. S. Fenton and D. W. Payne. Dynamic storage allocations of arbitrary sized segments. In *Proc. IFIPS*, pages 344–348, 1974.

[FP91] Matthew Farrens and Arvin Park. Dynamic base register caching: A technique for reducing address bus width. In *18th Annual International Symposium on Computer Architecture*, pages 128–137, Toronto, Canada, May 1991. ACM Press.

[GA95] Marcelo J. R. Goncalves and Andrew W. Appel. Cache performance of fast-allocating programs. In *FPCA '95*, 1995.

[Gar94] Laurie Garrett. *The Coming Plague: Newly Emerging Diseases in a World out of Balance*. Farrar, Straus and Giroux, New York, 1994.

[Gel71] E. Gelenbe. The two-thirds rule for dynamic storage allocation under equilibrium. *Information Processing Letters*, 1(2):59–60, July 1971.

[GGU72] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. In *Fourth Annual ACM Symposium on the Theory of Computing*, 1972.

[GM85] S. Gai and M. Mezzalama. Dynamic storage allocation: Experiments using the C language. *Software Practice and Experience*, 15(7):693–704, July 1985.

[Gra] R. L. Graham. Unpublished technical report on worst-case analysis of memory allocation algorithms, Bell Labs.

[GW82] A. Gottlieb and J. Wilson. Parallelizing the usual buddy algorithm. Technical Report System Software Note 37, Courant Institute, New York University, 1982.

[GZ93] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. *Software Practice and Experience*, 23(8):851–869, August 1993.

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation* [PLD93], pages 177–186.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes.

Software Practice and Experience, 20(1), January 1990.

[Har95] Juris Hartmanis. Turing award lecture: On computational complexity and the nature of computer science. *Computing Surveys*, 27(1):7–16, March 1995.

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

[Hay93] Barry Hayes. *Key Objects in Garbage Collection*. PhD thesis, Standford University, March 1993.

[Hin75] J. A. Hinds. An algorithm for locating adjacent storage blocks in the buddy system. *Communications of the ACM*, 18(4):221–222, April 1975.

[Hir73] D. S. Hirschberg. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, October 1973.

[HS64] V. C. Harris and C. C. Styles. A generalization of the Fibonacci numbers. *The Fibonacci Quarterly*, 2(4):227–289, December 1964.

[HS89] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1629, December 1989.

[IGK71] S. Isoda, E. Goto, and I. Kimura. An efficient bit table technique for dynamic storage allocation of 2^n-word blocks. *Communications of the ACM*, 14(9):589–592, September 1971.

[IJ62] J. K. Iliffe and J. G. Jodeit. A dynamic storage allocation scheme. *Computer Journal*, 5(3):200–209, October 1962.

[Ing61] P. Z. Ingerman. Thunks. *Communications of the ACM*, 4(1):55–58, January 1961.

[Iye93] Arun K. Iyengar. Parallel dynamic storage allocation algorithms. In *Fifth IEEE Symposium on Parallel and Distributed Processing*, 1993.

[Joh72] G. D. Johnson. Simscript II.5 User's Manual, S/360-370 Version, Release 6, 1972.

[Joh91] Theodore Johnson. A concurrent fast fits memory manager. Technical Report 91-009, University of Florida, 1991.

[JS92] T. Johnson and D. Sasha. Parallel buddy memory management. *Parallel Processing Letters*, 2(4):391–398, 1992.

[Kau84] Arie Kaufman. Tailored-list and recombination-delaying buddy systems. *ACM Transactions on Programming Languages and Systems*, 6(4):118–125, 1984.

[KLS92] Phillip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache performance of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[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.

[Kri72] Saul A. Kripke. *Naming and Necessity*. Harvard University Press, 1972.

[Kro73] S. Krogdahl. A dynamic storage allocation problem. *Information Processing Letters*, 2:96–99, 1973.

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

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

[LH82] B. W. Leverett and P. G. Hibbard. An adaptive system for dynamic storage allocation. *Software Practice and Experience*, 12(6):543–556, June 1982.

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

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

[Mah61] R. J. Maher. Problems of storage allocation in a multiprocessor multiprogrammed system. *Communications of the ACM*, 4(10):421–422, October 1961.

[Mar82] David Marr. *Vision*. Freeman, New York, 1982.

[McC91] Ronald McClamrock. Marr's three levels: a reevaluation. *Minds and Machines*, 1:185–196, 1991.

[McC95] Ronald McClamrock. *Existential Cognition: Computational Minds in the World*. University of Chicago Press, 1995.

[McI82] M. D. McIlroy. The number of states of a dynamic storage allocation system. *Computer Journal*, 25(3):388–392, August 1982.

[MK88] Marshall Kirk McKusick and Michael J. Karels. Design of a general-purpose memory

allocator for the 4.3bsd UNIX kernel. In *Proceedings of the Summer 1988 USENIX Conference*, San Francisco, California, June 1988. USENIX Association.

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

[MPS71] B. H. Margolin, R. P. Parmelee, and M. Schatzoff. Analysis of free-storage algorithms. *IBM Systems Journal*, 10(4):283–304, 1971.

[MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX 1993 Winter Technical Conference*, San Diego, California, January 1993. USENIX Association.

[Nel91] Mark Nelson. *The Data Compression Book*. M & T Books, 1991.

[Nie77] N. R. Nielsen. Dynamic memory allocation in computer simulation. *Communications of the ACM*, 20(11):864–873, November 1977.

[OA85] R. R. Oldehoeft and S. J. Allan. Adaptive exact-fit storage management. *Communications of the ACM*, 28(5):506–511, May 1985.

[Pag82] Ivor P. Page. Optimal fit of arbitrary sized segments. *Computer Journal*, 25(1), January 1982.

[Pag84] Ivor P. Page. Analysis of a cyclic placement scheme. *Computer Journal*, 27(1):18–25, January 1984.

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

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

[PLD93] *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993. ACM Press.

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

[PS70] P.W. Purdom and S. M. Stigler. Statistical properties of the buddy system. *Journal of the ACM*, 17(4):683–697, October 1970.

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

[Put77] Hilary Putnam. Meaning and reference. In Stephen P. Schwartz, editor, *Naming, Necessity, and Natural Kinds*. Cornell University Press, Ithaca, New York, 1977.

[Qui77] W. V. Quine. Natural kinds. In Stephen P. Schwartz, editor, *Naming, Necessity, and Natural Kinds*. Cornell University Press, Ithaca, New York, 1977.

[Ran69] Brian Randell. A note on storage fragmentation and program segmentation. *Communications of the ACM*, 12(7):365–372, July 1969.

[Ree79] C. M. Reeves. Free store distribution under random-fit allocation. *Computer Journal*, 22(4):346–351, November 1979.

[Ree80] C. M. Reeves. Free store distribution under random-fit allocation: Part 2. *Computer Journal*, 23(4):298–306, November 1980.

[Ree82] C. M. Reeves. A lumped-state model of clustering in dynamic storage allocation. *Computer Journal*, 27(2):135–142, 1982.

[Ree83] C. M. Reeves. Free store distribution under random-fit allocation, part 3. *Computer Journal*, 26(1):25–35, February 1983.

[Rei94] Mark B. Reinhold. Cache performance of garbage-collected programs. In *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–217, Orlando, Florida, June 1994. ACM Press.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, California, October 1991. ACM Press. Published as *Operating Systems Review 25*(5).

[Rob71] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, July 1971.

[Rob74] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):491–499, July 1974.

[Rob77] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *Computer Journal*, 20(3):242–244, August 1977.

[Ros61] D. T. Ross. A generalized technique for symbol manipulation and numerical calculation. *Communications of the ACM*, 4(3):147–150, March 1961.

[Ros67] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, August 1967.

[Rus77] D. L. Russell. Internal fragmentation in a class of buddy systems. *SIAM J. Comput.*,

6(4):607–621, December 1977.

[Sam89] A. Dain Samples. Mache: No-loss trace compaction. In *ACM SIGMETRICS*, pages 89–97, May 1989.

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

[Sho75] J. E. Shore. On the external storage fragmentation produced by first-fit and best-fit allocation strategies. *Communications of the ACM*, 18(8):433–440, August 1975.

[Sho77] J. E. Shore. Anomalous behavior of the fifty-percent rule in dynamic memory allocation. *Communications of the ACM*, 20(11):558–562, November 1977.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In Antonio Albano and Ron Morrison, editors, *Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer-Verlag.

[SP74] K. K. Shen and J. L. Peterson. A weighted buddy method for dynamic storage allocation. *Communications of the ACM*, 17(10):558–562, October 1974.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3), 1985.

[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.

[Sto82] Harold S. Stone. Parallel memory allocation using the FETCH-AND-ADD instruction. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, November 1982.

[Tad78] M. Tadman. Fast-fit: A new hierarchical dynamic storage allocation technique. Master's thesis, UC Irvine, Computer Science Dept., 1978.

[Thi89] Dominique Thiebaut. The fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, pages 1012–1026, July 1989.

[Tot65] R. A. Totschek. An empirical investigation into the behavior of the SDC timesharing system. Technical Report SP2191, Systems Development Corporation, 1965.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In Norman Meyrowitz, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '88) Proceedings*, pages 1–17, San Diego, California, September 1988. ACM Press.

[Ull95] Jeffrey D. Ullman. The role of theory today. *Computing Surveys*, 27(1):43–44, March 1995.

[Ung86] David Ungar. *Design and Evaluation of a High-Performance Smalltalk System*. MIT Press, Cambridge, Massachusetts, 1986.

[VC90] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software Practice and Experience*, 20(3):225–242, March 1990.

[VMH+83] J. Voldman, B. Mandelbrot, L. W. Hoevel, J. Knight, and P. Rosenfeld. Fractal nature of software-cache interaction. *IBM Journal of Research and Development*, 27(2):164–170, March 1983.

[Vo95] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 1995. To appear.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 29(4):229–239, April 1980.

[Wal66] B. Wald. Utilization of a multiprocessor in command and control. *Proceedings of the IEEE*, 53(12):1885–1888, December 1966.

[WB95] Paul R. Wilson and V. B. Balayoghan. Compressed paging. In preparation, 1995.

[WDH89] Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, December 1989.

[Wei76] Charles B. Weinstock. *Dynamic Storage Allocation Techniques*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1976.

[Whi80] Jon L. White. Address/memory management for a gigantic Lisp environment, or, GC considered harmful. In *LISP Conference*, pages 119–127, Redwood Estates, California, August 1980.

[Wil90] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990. Also appears in *SIGPLAN Notices*

23(3):45–52, March 1991.

[Wil91] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, California, October 1991. IEEE Press.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [BC92], pages 1–42.

[Wil95] Paul R. Wilson. Garbage collection. *Computing Surveys*, 1995. Expanded version of [Wil92]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps`. In revision, to appear.

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

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, December 1993. Expanded version of workshop position paper submitted for publication.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Memory allocation policies reconsidered. Technical report, University of Texas at Austin Department of Computer Sciences, 1995.

[WJW+75] William A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *Design of an Optimizing Compiler*. American Elsevier, 1975.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation* [PLD91], pages 177–191. Published as *SIGPLAN Notices 26*(6), June 1992.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on LISP and Functional Programming*, pages 32–42, San Francisco, California, June 1992. ACM Press.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings*, pages 23–35, New Orleans, Louisiana, 1989. ACM Press.

[Wol65] Eric Wolman. A fixed optimum cell-size for records of various lengths. *Journal of the ACM*, 12(1):53–70, January 1965.

[WW88] Charles B. Weinstock and William A. Wulf. Quickfit: an efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–144, October 1988.

[Yua90] Taichi Yuasa. The design and implementation of Kyoto Common Lisp. *Journal of Information Processing*, 13(3), 1990.

[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.

[ZG94] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):107–131, 1994.

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

This article was processed using the LaTeX macro package with LLNCS style