

# Non-Compacting Memory Allocation and Real-Time Garbage Collection

Dissertation Proposal

Mark S. Johnstone  
Dept. of Computer Sciences  
University of Texas  
Austin, Texas 78712-1188  
markj@cs.utexas.edu

Advisor: Paul R. Wilson

January 30, 1996

## Abstract

*Garbage collection* is the automatic reclamation of computer storage [Knu73, Coh81, Wil92, Wil95]. While in many systems, programmers must explicitly reclaim heap memory at some point in their program by using a “free” or “dispose” statement, garbage collected systems free the programmer from this burden.

In spite of its obvious attractiveness for many applications, garbage collection for real-time programs is not popular. This is largely due to the perceived cost and disruptiveness of garbage collection in general, and of incremental garbage collection in particular.

Most existing “real-time” garbage collectors are not in fact usefully real-time, largely due to the use of a *read barrier* to trigger incremental copying of data structures being traversed by the running application. This may slow down running applications unpredictably, even though individual increments of garbage collection work are small and bounded.

We have developed a hard real-time garbage collector which uses a *write barrier* to only coordinate collection work with *modifications* of pointers in data structures, therefore making coordination costs cheaper and more predictable. We combine this write barrier approach with *implicit non-copying reclamation* (“fake copying”), which has most of the advantages of copying collection (notably avoiding both the sweep phase required by mark-sweep collectors, and the touching of garbage objects when reclaiming their space), without the disadvantage of having to actually copy the objects.

It has long been believed that fragmentation will be prohibitively high in a non-copying garbage collection algorithm. We address these fragmentation issues with our garbage collector, and compare the fragmentation produced by a number of traditional memory allocation algorithms. In doing this comparison we produce some surprising results demonstrating that methodologies used to study fragmentation, dating back as far as 1961, are fundamentally unsound and biased. We have conducted sound studies that suggest that fragmentation can be kept *very* low for most real programs using well-known, non-copying policies.

Finally, we have extended our garbage collector with generational techniques to attempt to provide excellent average-case efficiency while preserving good soft real-time responsiveness.

# 1 Introduction

Memory management is poorly understood. In this research, we attempt to clarify many of the issues involved in non-compacting memory management in general, and in non-copying garbage collection in particular, especially high-performance real-time garbage collection. Among the most important of these issues are time and space costs. Our results show that the space costs for non-compacting memory management are very important, and quite surprising. We show that non-copying techniques can be used to implement a real-time garbage collector with acceptable space and time bounds.

In spite of its obvious attractiveness for many applications, real-time garbage collection is not popular. This is largely due to the perceived cost and disruptiveness of garbage collection in general, and of incremental garbage collection in particular. Our work shows that garbage collection need be neither expensive nor disruptive.

Many people have attempted to reduce the disruptiveness of garbage collection by implementing “real-time” collection routines. However, most existing “real-time” garbage collectors are not in fact usefully real-time, largely due to the use of a *read barrier* to trigger incremental copying of data structures being traversed by the running application. (The read barrier consists of extra instructions accompanying each pointer use, which ensures that the running program will never see inconsistencies due to encountering partially-copied data structures.) This may slow down running applications unpredictably, even though individual increments of garbage collection work are small and bounded.

We have developed a model for real-time garbage collection that is applicable to both copying and non-copying collection. This model allows for the systematic study and evaluation of different garbage collection techniques. In addition, we have used this model to help implement a hard real-time garbage collector which uses a *write barrier* to only coordinate collection work with *modifications* of pointers in data structures, therefore making coordination costs cheaper and more predictable. We combine this write barrier approach with *implicit non-copying reclamation*, (“fake copying”) which has most of the advantages of copying collection (notably avoiding both the sweep phase required by mark-sweep collectors, and the touching of garbage objects when reclaiming their space), without the disadvantage of having to actually copy the objects.

We have also implemented generational techniques in our garbage collector in an attempt to provide excellent average case performance while maintaining reasonable real-time responsiveness. Some earlier implementations of generational real-time garbage collectors had the severe problem of the entire system unpredictably slowing to a crawl for long periods of time while a full garbage collection of all generations was performed. Our techniques allow each generation to be collected largely independently of the other generations, so that collection work can proceed evenly throughout the run of the program.

Because a non-copying collector cannot compact live data, fragmentation is a potential problem in such a system. We believe that the problem is not as severe as it appears at first glance, and that for most programs, usable worst-case space bounds can be ensured. For other programs, it may also be feasible to rely on an occasional compacting collection with some degradation of real-time guarantees and/or overall performance [Bro84]. For still other applications, fragmentation issues will force the choice of a copying collector. Our work evaluates many of the costs associated with both copying and non-copying collection and is applicable to the entire spectrum of collection techniques.

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

We are attempting to provide a sound methodology with which memory management algorithms can be studied and compared. In particular, we are interested in measuring actual costs for actual programs, and in characterizing the situations under which different algorithms would be attractive.

We have produced results which strongly suggest that for a large class of programs, *memory fragmentation may be a non-problem*.

## 1.1 Scope of this Dissertation

The overall goals of this dissertation are:

1. To explore the basic design issues in allocators and incremental garbage collectors, putting them on a sounder footing,
2. To clarify the performance issues in both copying and non-copying real-time garbage collection,
3. To demonstrate that time overheads of non-copying real-time garbage collection can be kept low, and
4. To demonstrate that memory fragmentation costs are usually acceptable, or can be made acceptable with a modest amount of programmer effort or compiler cooperation.

## 2 Motivation

Real-time programs have traditionally been implemented in low level languages, such as assembly or C, which require explicit management of the heap. While using these languages has made for easier computation of real-time bounds, the lack of garbage collection has caused modularity, abstraction, and design to suffer. Some real-time programmers are finding tremendous increases in productivity by moving to higher level languages, such as Smalltalk, for real-time development [NR95]. Unfortunately, in order to guarantee real-time responsiveness, these programmers often find it necessary to “defeat the garbage collector” by statically allocating all objects that will ever be live in their application. This can result in programs requiring dramatically more memory because each word of memory can serve one and only one purpose. We believe that this is unnecessary, and that by providing a garbage collector that can meet hard real-time deadlines, these higher level languages will become even more attractive for real-time program development.

An additional motivation of this research is to clarify the basic issues involved in non-compacting memory management. There have been many different algorithms proposed and studied for both explicit and automatic memory management. However, the methodology used in many of these studies has been questionable, and the authors have done little to break down the individual costs of their algorithms, or to clarify the essential issues. By failing to do these things, they are unable to determine if a seemingly poor algorithm has fundamental policy problems, or if the behavior they are seeing is the result of a poor implementation of that policy. In addition, by failing to separate the different costs and categorize algorithms in a basic framework, they have made it very difficult to compare different algorithms on the basis of suitability to the problem domain.

This research attempts to clarify the different issues in memory management, identify many of the important costs, and provide a sound framework that can be used to evaluate different algorithms. By laying this groundwork, we hope to provide the basic understanding necessary to develop better memory management algorithms.

## 3 Overview

Memory management is poorly understood. In this dissertation, we clarify the issues pertaining to memory management in general, and real-time automatic storage reclamation (garbage collection) in particular. In doing this, we explore the basic design issues in incremental garbage collectors, putting them on a sounder footing. In addition, we clarify the performance issues of both copying and non-copying real-time garbage collection, and demonstrate that time and space overheads of non-copying real-time garbage collection can be kept low. We also explore and clarify the basic design issues of allocators, revealing important new insights that have gone overlooked for almost thirty years. Finally, we demonstrate that memory fragmentation costs are usually acceptable, or can be made acceptable with a modest amount of programmer effort.

### 3.1 Issues Pertaining to Real-time Garbage Collection

Real-time garbage collection must be *incremental*; that is, it must be possible to perform small units of garbage collection work while an application is executing, rather than halting the application and performing

large amounts of work without interruption. Strict bounds on individual garbage collection pauses are often used as the criterion for real-time garbage collection, but for practical applications, the requirements are often even stricter. A second requirement for real-time applications that has been almost universally overlooked in the real-time garbage collection literature is that the application must be able to make significant *progress*. That is, for a garbage collector to be usefully real-time, not only must the pauses be short and bounded, they must also not occur too often. In other words, the garbage collector must be able to *guarantee* not only that every garbage collection pause is bounded, but that for any given increment of computation, a minimum amount of the CPU is always available for the running application. Finally, because of the critical nature of most real-time applications, it is important to guarantee space bounds. This issue is particularly complicated for garbage collected systems because the programmer no longer has direct control on when a block of memory becomes available for reuse.

While *hard* real-time applications (critical applications with strict deadlines) are very important and largely unaddressed by the garbage collection literature, *soft* real-time applications (less critical real-time applications such as multi-media) make up an even larger set of problems that could benefit greatly from a real-time garbage collector. The issues in hard real-time garbage collection are very different from those in soft real-time. Hard real-time applications need *guarantees* on the worst case time and space cost of any operation. Soft real-time applications, on the other hand, are often more interested in average case performance, even if it is at the risk of missing an occasional deadline, *as long as these deadlines aren't missed too often*. In this work, we develop a model for both hard *and* soft real-time garbage collection as well as provide an implementation of a garbage collector that is fully configurable for both types of applications; allowing us to measure the effectiveness of many different hard and soft real-time garbage collection strategies.

### 3.2 A Model for Real-time Garbage collection

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

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

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

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

### 3.3 Generational Garbage Collection Techniques

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

Because generational techniques rely on a heuristic—the guess that most objects will die young, and that older objects wont die soon—they are not strictly reliable, and may degrade collector performance in the

worst case. Thus, for some purely hard real-time systems, they are not attractive. For other hard real-time applications with well understood object lifetimes and periodic scheduling of tasks, or for general-purpose systems with mixed hard and soft deadlines, the normal-case efficiency gain is likely to be highly worthwhile and the worst case is likely to be manageable.

We propose, implement, and test a design for generational garbage collection that is more amenable to real-time applications than any other design that we know of. The key point of our design is to largely decouple the collection of each generation from the collection of the others. This allows collection of different generations to run at different speeds, and to be scheduled with minimal coordination.

### 3.4 Performance Issues: Copying and Non-copying Real-time Garbage Collection

In this work, we attempt to clarify the different performance issues in both copying and non-copying real-time garbage collection. It is impossible to pick a winning strategy for all real-time applications because different strategies lead to different performance tradeoffs, which are heavily dependent on the characteristics of the application. However, we attempt to provide guidelines that can be used based on the particular problem at hand.

The primary question that needs to be addressed is whether fragmentation is going to be a problem with the application. If it can be reasoned that fragmentation is not likely to be a problem, then a non-copying approach is likely to win out over a copying approach for three basic reasons:<sup>1</sup>

1. *Better locality is achieved:* non-copying algorithms will tend to have a smaller live memory footprint than copying algorithms because a copying algorithm must always maintain two copies of every object (the version being copied from, and the version being copied to), while non-copying algorithms use only a single copy of every object. In addition, a copying algorithm must reference every byte of every live object on every garbage collection cycle in order to copy the objects. A non-copying algorithm can keep its per object information in a separate area, and need never touch the actual objects during collection.
2. *Less coordination between the application and the collector is needed:* a non-copying algorithm need only worry about coordinating with the collector the changes made to the graph by the application—the collector makes no changes to the graph, and therefore no further coordination is needed. A copying algorithm, on the other hand, needs coordination in both directions—the application needs to coordinate its changes to the graph with the collector, and the collector needs to coordinate its changes to the graph with the application.
3. *Less compiler support is needed:* Since a non-copying routine does not modify the graph of reachable objects in any way, it is amenable to conservative pointer finding techniques.<sup>2</sup> However, a copying routine needs compiler support for exact pointer information<sup>3</sup> so that when an object is moved, all pointers to that object can be updated.

If memory fragmentation does prove to be a problem, then copying algorithms become more attractive than a non-copying algorithm. As we said above, copying routines need more coordination between the application and the collector. There are three basic approaches to providing this coordination:

1. *Read-barrier only techniques* [Bak78] prevent the application from ever seeing an inconsistent copy of any object by catching all references to all objects. This approach is attractive in situations where

---

<sup>1</sup>Note that non-copying collection need not incur the cost of the sweep phase of a mark-sweep collector as is commonly assumed. In Section 5.7.1 we explain a technique known as “fake copying” [Wan89] (also known as “implicit reclamation” [Bak91]) which avoids the cost of a sweep phase.

<sup>2</sup>Conservative pointer finding techniques are subject to erroneously keeping objects (and hence entire data structures) live by finding an integer that happens to look like a pointer into the heap. Thus it is difficult to make guarantees on the amount of memory that can be reclaimed at any given point in a program using these techniques. This makes them unattractive for hard real-time applications

<sup>3</sup>Technically, the user could provide this support as we have done with our smart pointer implementation. However, we believe that our assertion is true since this support must be provided by some mechanism, and that in a well designed language the programmer would be free of this burden.

assignments are very common since it has no write barrier overhead. Unfortunately, this approach also has systematic problems where the simple act of traversing a linked list can cause an application to miss all of its real-time deadlines.

2. *Read barrier/write barrier combinations* [Bro84] combine a read-barrier and a write barrier to alleviate the worst case situation that we described with a read-barrier only approach by requiring that all object references be made through a pointer indirection. In this approach, objects can be copied at regularly scheduled intervals at the cost of an extra pointer in the header of every object, and having all reads and writes to those objects undergo a level of indirection. While this in general doubles the cost of reading from and writing to all heap allocated objects, the costs become predictable so that hard real-time assertions can be made.
3. *Write-barrier only techniques* [NOPH92] provide coordination between the application and the garbage collector by maintaining two separate views of the heap: one for the garbage collector, and one for the application. The application sees the original versions of all objects until the very end of the garbage collection cycle. Thus, the garbage collector is free to copy the objects at regularly scheduled intervals. The garbage collector is made aware of changes to the graph due to the application by *replicating* all modifications to an object in both versions of the heap. This approach is most attractive in languages where side effects are rare, as the write barrier overhead can be considerable.

In this work, we explore many different real-time garbage collection designs and consider the performance tradeoffs of each part of these designs. In particular, we have implemented a non-copying implicit-reclamation collector which is fully configurable to a number of different write barrier approaches. We intend to measure these different costs to show that real-time garbage collection is practical for many applications.

### 3.5 Basic Design Issues in Allocators

This work is motivated, in part, by our perception that there is considerable confusion about the nature of memory allocators, and about the problem of memory allocation in general. Worse, this confusion is often unrecognized, and allocators are widely thought to be fairly well understood. In fact, we know little more about allocators than was known twenty years ago, which is not as much as might be expected. The literature on the subject is rather inconsistent and scattered, and considerable work appears to be done using approaches that are quite limited. 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.<sup>4</sup>

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.<sup>5</sup>

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

---

<sup>4</sup>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.

<sup>5</sup>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.

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 work. The paradigm of statistical mechanics has been used in theories of memory allocation, but we believe that it is the wrong paradigm, at least as it is usually applied. 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.

The traditional methodology of using random program behavior implicitly assumes that there is *no* ordering information in the request stream that could be exploited by the allocator—i.e., there is nothing in the sequencing of requests which the allocator 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.

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

- *Fragmentation is caused by isolated deaths.* A crucial issue is the creation of free areas whose neighboring areas are not free. This is a function of two things: *which objects are placed in adjacent areas* and *when those objects die*. Notice that if the allocator places objects together in memory, and they die “at the same time” (with no intervening allocations), no fragmentation results: the objects are live at the same time, using contiguous memory, and when they die they free contiguous memory. An allocator that can predict which objects will die at approximately the same time can exploit that information to reduce fragmentation, by placing those objects in contiguous memory.
- *Fragmentation is caused by time-varying behavior.* Fragmentation arises from *changes* in the way a program uses memory—for example, freeing small blocks and requesting large ones. Fragmentation is also caused by patterns in the changing behavior of a program, such as the freeing of large numbers of objects and the subsequent allocation of large numbers of objects of a different type. 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. The allocator’s job is to exploit these patterns, if possible, or at least not let the patterns undermine its strategy.

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

### 3.6 Developing a Sound Methodology

The traditional view has been that the program behavior responsible for fragmentation is determined only by their object size and lifetime distributions. Recent experimental results show that this is false ([ZG94, WJNB95]), 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.

To use the trace for a simulation, a driver routine reads request records out of a 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 does not 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.

### 3.7 Experimental Results

An interesting result that we have produced is that *small variations in policy can have large variations in fragmentation*. It is therefore critical that memory allocation researchers provide exact specifications of their policies in order to allow others to duplicate their work. We have developed a number of different features of memory allocation policies, each of which is relatively independent of the others, to help describe different policies. No doubt there are others, and we encourage other researchers to help identify them.

Another interesting result is that among the policies we have studied, the best-fit allocation policy works best. What is most surprising though is how well best-fit works. We have found that once minimum object size and hardware alignment constraints are factored out, when simulated with eight different memory intensive programs (see section C.2 for a description of these programs) the best-fit policy produces an average of less than *one percent* fragmentation. This strongly suggests that for a large class of programs, *memory fragmentation may be a non-issue*.

### 3.8 Memory Fragmentation Costs Can Be Kept Low

In this research, we intend to factor out the different memory allocation costs, and characterize the ways in which memory allocation algorithms are data-dependent so that programmers can reason about how these dependencies will affect their program's interaction with the memory allocator. In addition, we will separate out the average (expected) case memory fragmentation costs from the worst case costs for a variety of situations.

## 4 Background<sup>6</sup>

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

1. Distinguishing the live objects from the garbage in some way (*garbage detection*).
2. Reclaiming the garbage objects' storage so that the running program can use it again (*garbage reclamation*).

In practice, these two phases may be functionally or temporally interleaved, and the reclamation technique is strongly dependent on the garbage detection technique.

---

<sup>6</sup>This paper assumes some background in garbage collection techniques. For an excellent survey of basic garbage collection techniques the interested reader should see [Wil95].

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

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

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

There are two basic ways to reclaim garbage objects:

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

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

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

While at first these two methods of reclaiming garbage memory may seem fundamentally different, there is a way to combine them to receive many of the advantages of both [Wan89, Bak91]. This “fake copying” approach is fundamental to our real-time garbage collector implementation, and we will discuss it in detail in Section 5.5.

## 4.1 Real-time Garbage Collection

For truly real-time applications, fine-grained incremental garbage collection appears to be necessary. Garbage collection cannot be carried out as one atomic action while the program is halted. Small units of garbage collection must be interleaved with small units of program execution.

Real-time programs are usually characterized as being either *hard real-time* or *soft real-time*. Hard real-time programs are programs with very strict bounds of the running times of program operations. Examples of hard real-time programs are airplane fly-by-wire control software, missile guidance software, and medical equipment control software. The defining characteristic of hard real-time programs is that the consequences of missing a deadline are very great (the airplane crashes, the missile misses its target, or the patient dies). In addition, there are a number of programs which can benefit from a real-time collector, but do not have hard real-time requirements. We call these programs “soft real-time.”

Soft real-time programs are programs that should make a majority of their deadlines, but it is acceptable if an occasional deadline is missed, as long as the deadlines are not missed too frequently at a time-scale relative to the program. Examples of soft real-time programs are multimedia applications, graphical user interfaces, and non-critical control software. For these applications, it does not really matter if the occasional frame of video is missed or the mouse cursor occasionally skips a little, as long as this does not happen too often.

The difficulty with incremental tracing is that while the collector is tracing out the graph of reachable data structures, the graph may change—the running program may *mutate* the graph between invocations of the collector. For this reason, discussions of incremental collectors typically refer to the running program as

the *mutator* [DLM<sup>+</sup>78]. An incremental scheme must have some way of keeping track of the changes to the graph of reachable objects, perhaps re-computing parts of its traversal in the face of those changes.

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

In our work, we attempt to characterize how the choice of a write barrier affects the collector’s conservatism as well as its ability to meet its real-time deadlines.

## 4.2 Generational Garbage Collection

Given a realistic amount of memory, efficiency of simple garbage collection is limited by the fact that the system must traverse all live data during a collection cycle. In most programs in a variety of languages, *most objects live a very short time, while a small percentage of them live much longer* [LH83, Ung84, Sha88, Zor90, DeT90b, Hay91]. While figures vary from language to language and from program to program, usually between 80 and 98 percent of all newly-allocated heap objects die within a few million instructions, or before another megabyte has been allocated; the majority of objects die even more quickly, within tens of kilobytes of allocation.

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

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

For stop-and-collect garbage collection, generational garbage collection has an additional benefit in that most collections take only a short time—collecting just the youngest generation is much faster than a full garbage collection. This reduces the frequency of disruptive pauses, and for many programs without real-time deadlines, this is sufficient for acceptable interactive use. The majority of pauses are so brief (a fraction of a second) that they are unlikely to be noticed by users [Ung84]; the longer pauses for multi-generation collections can often be postponed until the system is not in use, or hidden within non-interactive compute-bound phases of program operation [WM89]. Generational techniques are often used as an acceptable substitute for more expensive incremental techniques, as well as to improve overall efficiency. However, they are *not* sufficient for *hard* real-time applications. In this dissertation we will explore some novel generational garbage collection algorithms in an attempt to provide the benefit of generational techniques for many *soft* real-time applications.

## 4.3 Fragmentation

An important issue with memory allocation routines (both manual and automatic) is *fragmentation*. Traditionally, fragmentation is classified as *external* or *internal* [RK68], and is combatted by splitting and coalescing free blocks.

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

Internal fragmentation arises when a 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 is called *internal* because the wasted memory is inside an allocated block, rather than being recorded as a free block in its own right.)

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

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

## 5 Real-Time Garbage Collection

We have implemented a hard real-time garbage collector using a technique called *non-copying implicit reclamation*, or “fake copying”. This technique gives our collector many of the advantages of a copying collector without some of the associated costs. This collector currently works with a locally developed Scheme compiler, Tower Technology Corporation’s Eiffel compiler, and C++.<sup>7</sup> We have found, through our survey of the real-time garbage collection literature, that there is a pervasive misconception as to what is required for a garbage collector to be called real-time. We will therefore begin with a discussion of our views on what is required of a garbage collector before it can be called real-time. We will then develop a theoretical framework for the discussion and comparison of incremental garbage collection techniques. After laying this groundwork, we will then describe our garbage collector implementation and show how it meets these criteria. Next, we will present measurements for the performance of our collector. Finally, we will discuss incremental generational garbage collection and its applicability to real-time programming.

### 5.1 Real-Time Collection: What it is and When it Is Not

Real-time garbage collection must be *incremental*; that is, it must be possible to perform small units of garbage collection work while an application is executing, rather than halting the application and performing large amounts of work without interruption. Strict bounds on individual garbage collection pauses are often used as the criterion for real-time garbage collection, but for practical applications, the requirements are often even stricter. A second requirement for real-time applications that has been almost universally overlooked in the real-time garbage collection literature is that the application must be able to make significant *progress*. That is, for a garbage collector to be usefully real-time, not only must the pauses be short and bounded, they must also not occur too often. In other words, the garbage collector must be able to guarantee not only that every garbage collection pause is bounded, but that for any given increment of computation, a minimum amount of the CPU is always available for the running application. Finally, because of the critical nature of most real-time applications, it is important to guarantee space bounds. This issue is particularly complicated for garbage collected systems because the programmer no longer has direct control on when a block of memory becomes available for reuse.

#### 5.1.1 Why Baker’s Read Barrier is Not Usefully Real Time

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

---

<sup>7</sup>We use the smart pointer idiom [Str92] to provide the necessary support for garbage collection.

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

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

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

Nilsen’s proposed solution to this problem is to build special hardware that guarantees that the worst-case delay for any individual program operation is small relative to that operation’s normal execution time [NS90]. Unfortunately, he gives no indication of what that worst-case delay is for his hardware except to admit that the worst-case for a pointer dereference is 2 microseconds.<sup>8</sup> On a 100 MIPS machine, this would be a slowdown of 200 times! In addition, his scheme requires that the cache be flushed at the end of every garbage collection cycle, causing a further unpredictable loss of performance. The basic problem with his approach is that he is trying to use special hardware to speed up Baker’s incremental copying algorithm, and that algorithm is fundamentally unsuitable for real-time garbage collection.

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

Others have proposed copying based algorithms that rely on a combination of a read barrier and a write barrier [Bro84], or a write barrier only [NOPH92] to coordinate the collector’s view of the graph with that of the application. Because these algorithms rely on a write barrier, the issues that we explore with our garbage collection model and measure with our implementation are applicable to these collectors as well as our own non-copying collector.

## 5.2 Coherence and Conservatism

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

---

<sup>8</sup>Nilsen gives no indication of the parameters used in timing this worst-case, so it is impossible to evaluate this pause.

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

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

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

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

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

The kind of relaxed consistency used, and the corresponding coherence features of the collection scheme, are closely intertwined with the notion of conservatism. In general, the more we relax the consistency between the mutator’s and the collector’s views of the reachability graph, the more conservative our collection becomes, and the more floating garbage we must accept. On the positive side, the more relaxed our notion of consistency, the more flexibility we have in the details of the traversal algorithm. (In parallel and distributed garbage collection, a relaxed consistency model also allows more parallelism and/or less synchronization, but that is beyond the scope of this dissertation.)

### 5.3 Tricolor Marking

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

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

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

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

and the offspring are colored gray.

In summary, the significance of these three colors is:

- White objects are those that have not yet been reached by the collector’s tracing traversal. If at the end of collection an object is still marked white, then it is known to be garbage.
- Gray objects are those currently under consideration by the collector, because they are known to be reachable, but it is not yet known what other objects are reachable from them. In implementation terms, this just means that the objects are in the stack (or queue) that controls the collector’s traversal of reachable data.
- Black objects are those that the collector is through with—they have already been examined and their role in the reachability graph is known. In terms of implementation, this means that they have been removed from the traversal stack (or queue).

As shown in Figure 1, the traversal proceeds in a wavefront of gray objects, which separates the white (unreached) objects from the black objects that have been passed by the wave—that is, there are no pointers directly from black objects to white ones. This abstracts away from the particulars of the traversal algorithm—it may be depth-first, breadth-first, or just about any kind of exhaustive traversal. It is only important that a well-defined gray fringe be identifiable, and that the mutator preserve the invariant that no black object hold a pointer directly to a white object.

The importance of this invariant<sup>9</sup> is that the collector must be able to assume that it is “finished with” black objects, and can continue to traverse gray objects and move the wavefront forward. If the mutator creates a pointer from a black object to a white one, it must somehow notify the collector that this assumption has been violated. This ensures that the collector’s bookkeeping is brought up to date.

### 5.3.1 The Tricolor Invariants

The main problem of incremental collection is to ensure that the collector’s notion of the reachability graph does not get out of synchronization with the actual reachability graph, due to changes made by the running program (see Figure 2). If the program creates a pointer from a black object to a white object, and nothing special is done, the pointer will not be found by the collector. (Recall that the garbage collector has already examined black objects, and will not look at them again.) If all other paths to the white object are broken before being reached by the collector, the white object will be reclaimed. Since the application can still reach the white object through the black object’s pointer, this creates a dangling pointer.

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

1. *The **strong** tricolor invariant*: For all black objects in the graph which have a path to a white object, *all* paths from that black object to that white object will contain at least one gray object [DLM<sup>+</sup>78].
2. *The **weak** tricolor invariant*: For all black objects in the graph which have a path to a white object, *at least one* path from that black object to that white object will contain at least one gray object [Yua90].

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

---

<sup>9</sup>We call this the *strong* tricolor invariant (see Section 5.3.1). Other invariants are also possible.

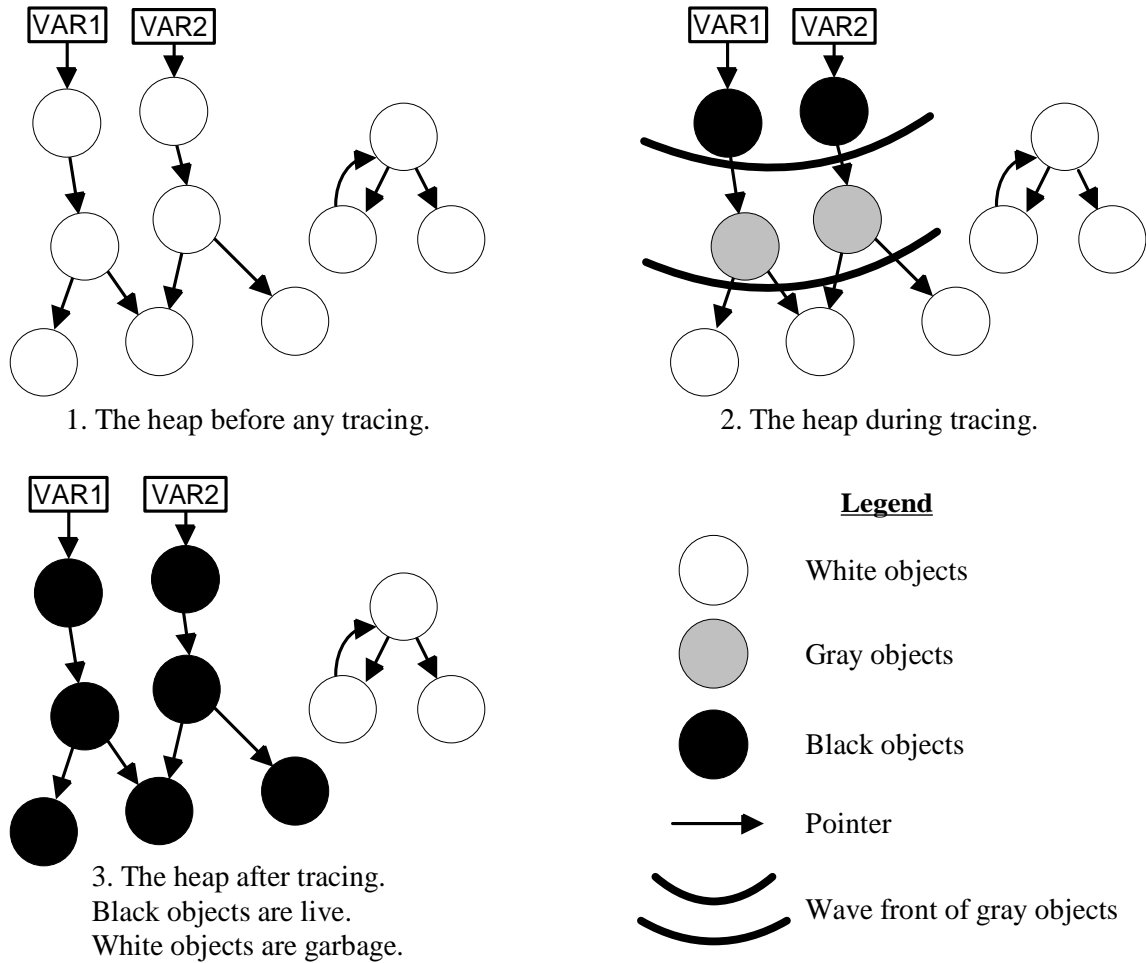


Figure 1: Example of tricolor marking

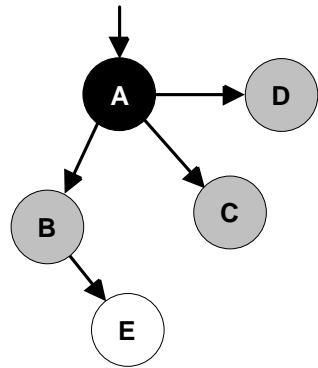
**Proof Sketch for the Weak Tricolor Invariant.** To see that preserving the weak tricolor invariant will ensure that all live objects are eventually marked, consider that the weak tricolor invariant states that all live white objects have at least one path to them from a gray object and that path does not contain a black object. Garbage collection work will only shorten such paths. The only way these paths can get longer is if the application changes the graph by adding a new white live object. In this case, the ratio of the number of live white objects discovered by the collector to the number of new white objects added to the graph by the application must be greater than one,<sup>10</sup> so that the length of these paths can only decrease, and eventually the garbage collector will find all live white objects and terminate.

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

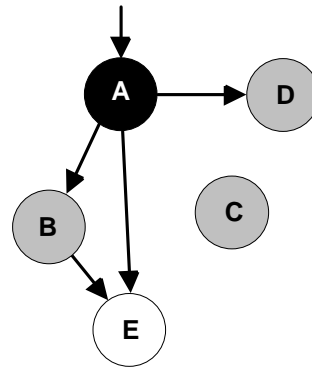
Still another optimization allowed by the strong tricolor invariant is that the collector can use any available oracle which can tell it if a white object is already garbage. For example, in combining garbage collection of some objects with explicit (programmer invoked) deallocation of others, the programmer may notify the collector that an object will never be used again beyond a particular point in the program's execution.<sup>11</sup> If

<sup>10</sup>The user is required to select this ratio based on the requirements of the application. See Section 5.9.

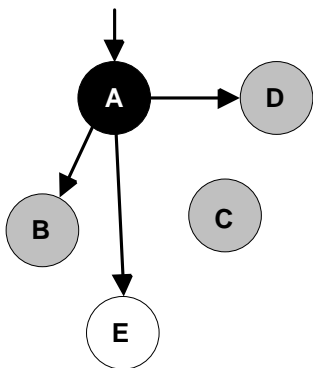
<sup>11</sup>Another example is combining tracing collection with reference counting, as is common in distributed garbage collection.



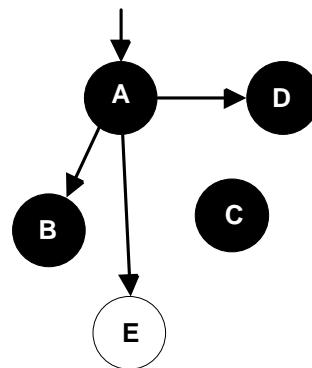
1. Before pointer manipulation.



2. Object A is modified to point to object E instead of object C.



3. The pointer from object B to object E is removed.



4. At the end of collection, object E will be (incorrectly) reclaimed. Object C is floating garbage.

Figure 2: Example of violating the tricolor invariant

the object is white or gray, the collector can use that information to short-circuit its traversal at that object. Assuming the oracle is correct, this will not erroneously reclaim any reachable objects—if there are other paths to other white objects, there will be gray objects on those other paths, and the collector will find them. This kind of optimization is also the essence of hierarchical garbage collection, which we will discuss in Section 6. Note that this optimization is not possible with the weak tricolor invariant. If an oracle tells you that an object is garbage you can only reclaim its space—you cannot short-circuit the traversal at that object because it might hold the only path *known to the garbage collector* to a set of live objects that will otherwise be erroneously reclaimed.

In the worst case, the collector will traverse all objects just before they become garbage, and the above two optimizations will have no effect on the amount of garbage reclaimed by the collector. Even though they provide no benefit for a hard real-time collector, we will see in Section 5.10 that these optimizations can be very useful for a soft real-time collector.

However, an important optimization for hard real-time collection *is* possible with the weak tricolor invariant: a pointer that has already been traced can be overwritten without violating the invariant. This is because once a pointer is traced, it is a pointer from a black object to a black or gray object, and hence cannot be a pointer on a path from a gray object to a white object with no intervening black objects. Also, any value written over this pointer must either be a copy of a pointer that exists elsewhere in the graph, or a pointer value that points to a new object. As long as all new objects are allocated black (see Section 5.3.2), and we optimistically trace all root objects at the beginning of collection, we can use this observation

to avoid having a write barrier (extra instructions executed at every pointer store) to protect against the mutation of root objects.

In summary, an error in collection like that shown in Figure 2 requires that two things happen:

1. The mutator creates a pointer from a black object to a white object.
2. The mutator breaks the old path to the white object.

Maintaining the strong tricolor invariant requires that special action be taken to ensure the first case never happens. Preserving the weak tricolor invariant requires that special action be taken to ensure the second case never happens because this might be the only path *known to the garbage collector* to that white object. As we will see, the choice of which invariant to maintain dictates the conservatism and coherence properties of the two algorithms.

### 5.3.2 Allocation Color

When using a tricolor marking technique such as we do, newly allocated objects must be colored one of three colors: white, gray, or black. If a newly allocated object is colored white, then no write barrier is needed on initializing writes to this object to maintain either the strong or the weak tricolor invariants. In addition, if the strong tricolor invariant is being preserved, and this object becomes garbage before being traced, then it can be reclaimed at the end of this garbage collection cycle. However, since the set of live white objects is no longer monotonically decreasing, the garbage collector must trace at a rate *greater than the rate of allocation* in order to ensure that the collection will eventually terminate.

If newly allocated objects are colored gray, then again neither the strong nor weak tricolor invariants can be violated by initializing writes to these objects. In addition, since the number of live white objects does not increase, the collector can trace at a rate much slower than when allocating white. However, since these objects are already shaded gray, if a newly allocated object becomes garbage before the end of this collection cycle, it cannot be reclaimed before the end of the *next* garbage collection cycle (for example, object C in Figure 2). In addition, the newly allocated objects must still be traced by the collector to look for pointers to untraced objects.

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

## 5.4 Incremental Tracing Algorithms

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

Snapshot-at-beginning algorithms rely on the weak tricolor invariant to ensure that all live objects are eventually marked. The weak tricolor invariant is preserved by using a write barrier to detect any attempts to overwrite any pointers in the graph. The overwritten pointers are saved until the collector can process them. For example, in Figure 2 step 3, before the pointer from object B to object E is broken, it would be recorded, and object E would be traversed when this pointer is reexamined. This has the effect of generating a *snapshot* of the graph at the beginning of the garbage collection cycle.

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

The other strategy for collection, *incremental update*, focuses on the writing of new pointers in objects that the collector has already reached and examined. When such a pointer is created, the collector is notified

so that it can either trace the pointed-to object immediately, or re-examine the location in which the pointer was stored again later to find any “hidden” objects [Ste75, DLM+78, BDS91]. For example, in Figure 2 step 2, when the pointer from object A to object E is created, a pointer *to this pointer* is recorded, and object E would be traversed when this pointer is reexamined. That is, the collector’s view of reachable data structures is incrementally updated in the face of changes to those data structures by the running program.

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

## 5.5 Non-Copying, Incremental Read Barrier Techniques

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

Baker’s incremental non-copying algorithm (called the *treadmill* algorithm) [Bak91] uses doubly-linked lists (and per-object color fields) to implement the garbage collection sets, rather than separate memory areas. These lists are linked into a cyclic structure, as shown in Figure 3. This cyclic structure is divided into four sections: the *new-set*, the *free-set*, the *from-set* and the *to-set*.

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

The *from-set* holds object that were allocated before garbage collection began, and which are currently subject to garbage collection. In terms of tri-color marking, these objects are white. As the collector and mutator traverse data structures, objects are moved from the from-set to the *to-set* and colored gray by setting a bit in the objects’ header. The *to-set* is initially empty, but grows as objects are removed from (unlinked from) the from-set and moved to (linked into) the *to-set* during collection.

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

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

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

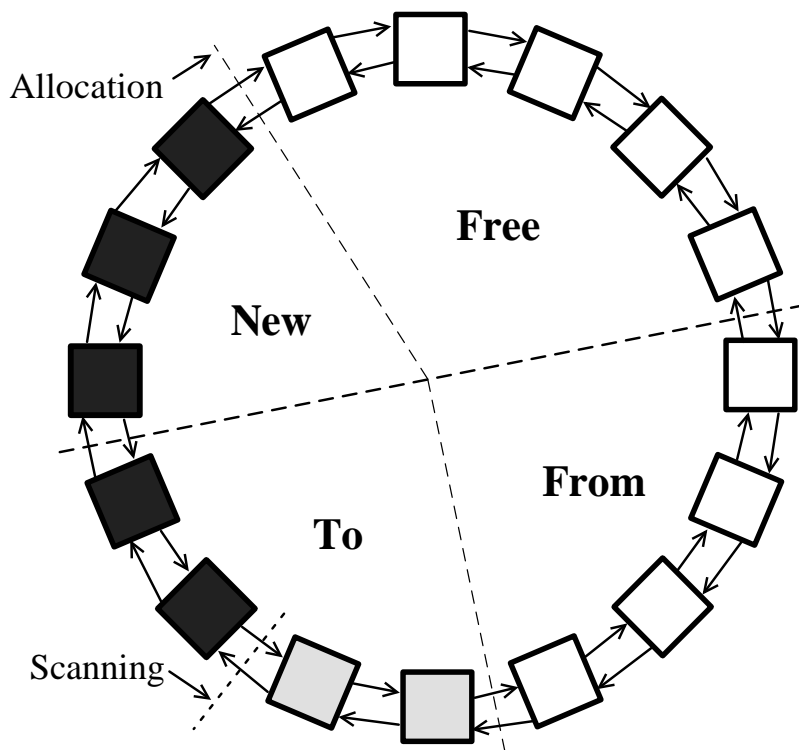


Figure 3: Treadmill collector during collection.

to the size of the object as it was in the copying version.

## 5.6 Non-Copying, Incremental Write Barrier Techniques

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

Write barrier algorithms coordinate the collector's view of data structures with the running application's view. Whenever the application modifies a data structure by changing a pointer, the collector must be protected from being confused and "losing" objects. This can happen if an object is hidden from the collector by storing a pointer to the object in another object that the collector has already examined, and then breaking all other paths to the first object (see Figure 2).

## 5.7 Our Testbed Implementation

To test the relative effectiveness of the various garbage collection strategies presented above, we have implemented a fully configurable, non-copying, implicit reclamation garbage collector. In what follows, we will describe one possible configuration of this collector, and then present some performance figures for that implementation. It should be noted that this is only one of many possible configurations, and, as part of the proposed work for this dissertation, we will be experimenting with other configurations. It should also be noted that the results that we will gather will be valid for any write barrier collection strategy, not just

for non-copying implementations.

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

- *No write barrier on initializing writes*: Initializing writes in newly-allocated objects need incur no write barrier overhead because such stores overwrite no existing pointers, and therefore cannot violate the weak tricolor invariant.
- *No write barrier on pointer stores in root variables*: Because we trace all root objects at the beginning of collection, pointer stores in local or global variables need incur no write barrier overhead; such writes cannot violate the weak tricolor invariant.<sup>12</sup>
- *Slower rate of tracing is required*: The garbage collector can trace at a rate that is much slower than if we allocate white, and therefore incur much less runtime overhead. In fact, with black allocation the tracing rate can be made arbitrarily slow as the amount of memory is increased. With white allocation, the tracing rate can never be slower than the rate of allocation.
- *New objects need not be considered by the collector*: Newly allocated objects potentially have as much as one full garbage collection cycle to die before they are traced by the collector. If they die before the end of the cycle during which they are allocated, then they are never considered by the collector. Since a majority of objects die very quickly after being allocated, this is likely to significantly reduce the amount of garbage collection work that needs to be performed. We expand on this idea considerably in Section 6.

### 5.7.1 Non-copying Implicit Reclamation

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

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

Figure 4 shows an example of one of these doubly-linked lists. The white objects are all of the objects between the free pointer (inclusive) and black pointer (exclusive); the black objects are those objects between the black pointer (inclusive) and the scan pointer (exclusive); and the gray objects are those from the scan pointer on to the right.

Figure 5 shows how an object is grayed. It is unlinked from the white set, linked into the gray set, and a bit in its header is set to indicate that it is now gray. If it is put on the right end of this list (as shown in the example) then the garbage collector’s traversal is breadth first. If it is put on the left end, then the traversal is depth-first.

Figure 6 shows how an object is blackened. First the object under the scan pointer is scanned for pointers to white objects. If any white objects are found, they are put into the gray set. Finally, the scan pointer is moved one object to the right. After all reachable data have been moved from the white set to the black set, the remaining objects in the white set are known to be garbage and that list can simply be appended to the free list in small constant time.

This pseudo-copying is cheaper than real-copying for most objects, and also avoids the need to keep the relocation of objects from confusing the running application.<sup>13</sup>

Our collector generalizes this scheme by combining it with a simple *segregated storage* scheme for the management of different-sized chunks of memory. In such a scheme, separate sets of lists are used to manage different-sized objects, with objects of similar sizes grouped into “size classes.”

---

<sup>12</sup>See Section 5.3.1 for a more detailed discussion as to why this is true.

<sup>13</sup>The main motivation for Baker’s read barrier is really to keep the running application from seeing temporary inconsistencies in data structures while they are being copied by the collector—that is, the read barrier protects the application program from changes made by the collector, as well as keeping the collector from being confused by the application’s changes.

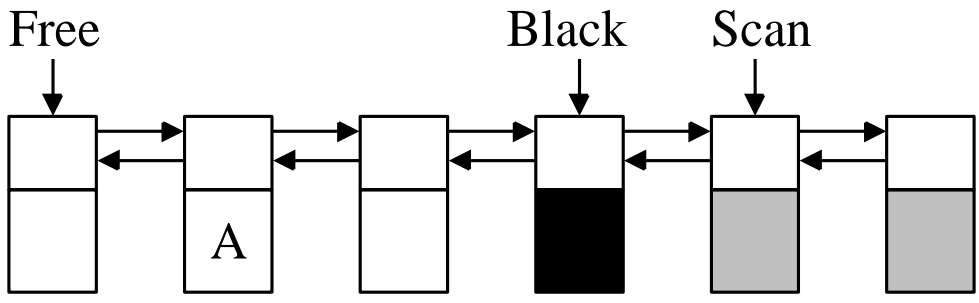


Figure 4: The initial state of the heap

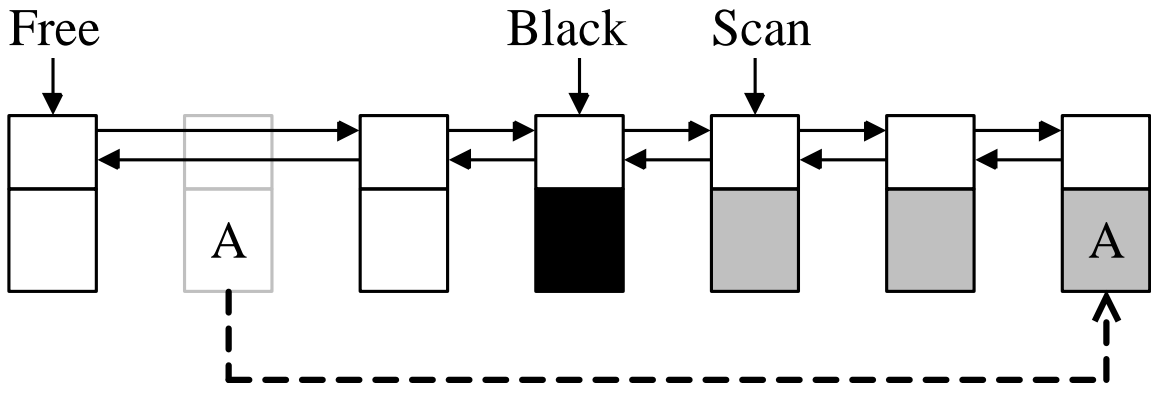


Figure 5: Graying a white object

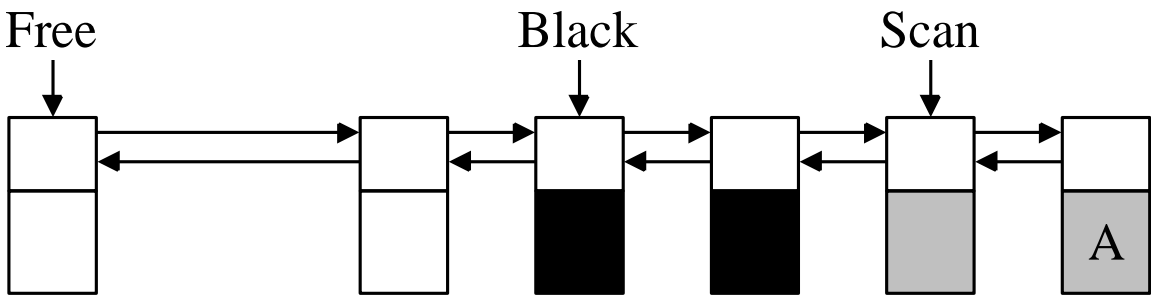


Figure 6: Blackening a gray object

Currently, our size classes are powers of two. When allocating an object, we round its size up to the next power of two and allocate it in that size class. Like any non-compacting storage scheme, our segregated storage scheme is vulnerable to fragmentation. We expect the fragmentation cost to be low for most programs, and we decrease the worst-case bound (at a cost in average-case fragmentation) by using relatively coarse gradations of sizes. This keeps memory from ever being divided too finely and hence becoming unusable for different-sized objects. The current scheme does no coalescing of adjacent free blocks into larger free blocks.

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

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

We do this recoloring in real-time by deferring it until the garbage blocks are reallocated. I.e., we append the garbage objects to the list of free objects, so that the free-set contains objects of multiple colors, and lazily reset the headers at allocation time.<sup>14</sup> Since neither the application nor the collector examines the objects in the free set until they are reallocated, this works just as well, and spreads the cost over the allocations, rather than incurring it all at once.

## 5.8 Real-time Timing Requirements

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

In the following subsections, we will only be describing the hard real-time mode of our collector. In Section 5.10 we will look at these same details with respect to the soft real-time mode of our collector.

### 5.8.1 Allocating Memory

As we said earlier, the current implementation of our garbage collector uses a segregated storage system for its lists of free blocks.<sup>15</sup> When a request for a free block is processed, the smallest power of two size which is larger than the requested size is computed. This computation is done in very small constant time by a simple table lookup.<sup>16</sup>

If bounds on the memory usage of the running application can be computed in advance (see Section 5.9 for a detailed discussion on this) then memory can be pre-allocated for each size class. In this case, the remainder of the allocation work reduces to a check to see if a garbage collection increment must be performed, and if not, the size of this request is recorded, the free pointer is incremented in the doubly linked list that

---

<sup>14</sup>Conceptually, we recolor all of the freed chunks of memory at the instant a collection is complete, but the chunks' color fields are only reset when they are reused to allocate new objects.

<sup>15</sup>We will discuss how to implement other storage management schemes in real-time in section 7. The choice of a simple segregated storage scheme was made for simplicity of implementation, and for adaptability to C++. Nevertheless, we feel it necessary to describe the timing requirements of the segregated storage algorithm here, so that the reader can feel comfortable that at least one algorithm exists which can meet our hard real-time goals as set forth in section 5.1.

<sup>16</sup>For object sizes less than 255 words, we use a simple table lookup. For larger object sizes, we use a conditional, a bit shift, and a table lookup. For the largest object sizes, we have four conditionals, one bit shift, and a table lookup. If the size is known at compilation time, the conditionals and bit shift can be optimized away. With very aggressive optimization, the table lookup can also be eliminated.

represents the appropriate size class, and a copy of the free pointer (which is a pointer to the newly allocated block) is returned.

If bounds on the memory usage of the running application cannot be computed in advance, then the additional case of needing to request more memory from the operating system is added to the work needed to be performed to allocate a block. If, at any time, there are no free blocks in the appropriate size class, then another free block is created out of a raw page of memory which is allocated by the operating system.<sup>17</sup> For size classes of less than 4K bytes, a single 4K page of memory is used for each size class to satisfy the need for additional objects. Objects are carved out of this page one at a time and returned to the application. This operation involves a conditional and the adjustment of a few pointers. If this page was exhausted by the previous new object request, then another page is requested from the operating system. For size classes greater than 4K bytes, the memory for each new object is directly requested from the operating system and these pages are recorded as being large object pages.

### 5.8.2 The Write Barrier

Our write barrier is very simple. It only needs to check whether the object on the right hand side of the assignment (R-value) is already shaded (colored black or gray). If not, then the R-value object is immediately grayed. On most modern architectures, this can be performed in a small number of instructions.<sup>18</sup> On a Pentium using the GNU g++ compiler version 2.7.0, our write barrier's conditional is just 11 instructions. If this conditional determines that the R-value needs to be shaded, then an additional 48 instructions are required to gray the object. Note that graying the old R-value object is an operation that *must* eventually be done; if these objects were not found by the write barrier, then they would eventually be found by the collector's marking traversal, so over the course of the computation, no additional work is performed by shading these objects eagerly.

### 5.8.3 Performing an increment of garbage collection work

During the first increment of a garbage collection cycle, the root set is traced *atomically* and all objects pointed to by root variables are grayed. This gives us a cost directly proportional to the number of root variables. As we described in Section 5.7 this allows us to forgo the need for a write barrier on assignments into root variables.

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

If at any point, there are no more grays to blacken, then collection completes. At this point, the garbage collector has a choice of whether to reclaim garbage objects immediately, or to continue allocating with the garbage collector turned off. If the collector is coloring newly allocated objects white, then the garbage objects are immediately reclaimed, and collection starts again. If the collector is allocating black (recommended for hard real-time applications) and there still exist available free objects then garbage collection stops until some object size is unavailable. At that time the garbage objects are reclaimed and garbage collection begins again.

The rate and duration of pauses in the running application due to garbage collection work is directly related to the number of bytes allocated by the application. The user chooses how many bytes can be allocated between interruptions by the collector, and how many bytes should be reclaimed during each interruption. These two parameters determine the length and frequency of garbage collection pauses. Figure 7 is a histogram of garbage collection pauses for the Lindsay program being traced at a rate of blackening 4K

---

<sup>17</sup>Here, by page, we mean a logical page of memory and not a physical page as defined by the virtual memory system of the host machine. In fact, for a hard real-time application, virtual memory (or at least paging) will not be used at all.

<sup>18</sup>We first load and mask to get the color of the R-value object from its header. We then load the current shade value (recall that the bit pattern used to indicate that an object is shaded (gray or black) is not changed in each *object* from collection cycle to collection cycle, but rather the *meaning* of the bit pattern changes) and compare that to the color of the R-value object. Finally, we conditionally gray the R-value object if it is not already the current shade value. Graying an object involves setting a bit in the header of the object, and six pointer modifications.

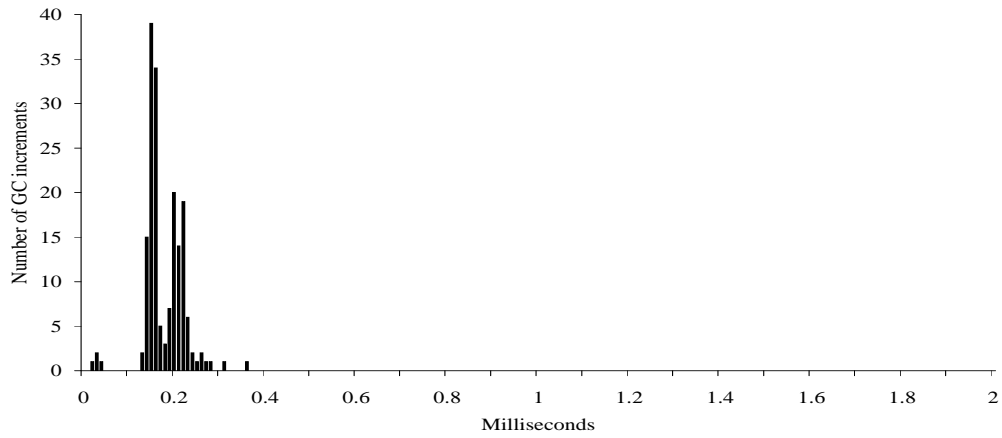


Figure 7: Histogram of the garbage collection increment costs for the Lindsay program (Throttle 0.5)

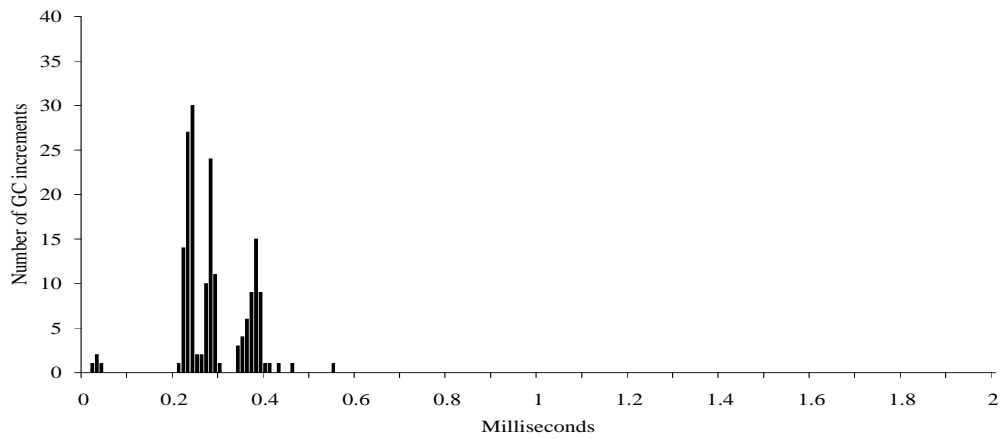


Figure 8: Histogram of the garbage collection increment costs for the Lindsay program (Throttle 1.0)

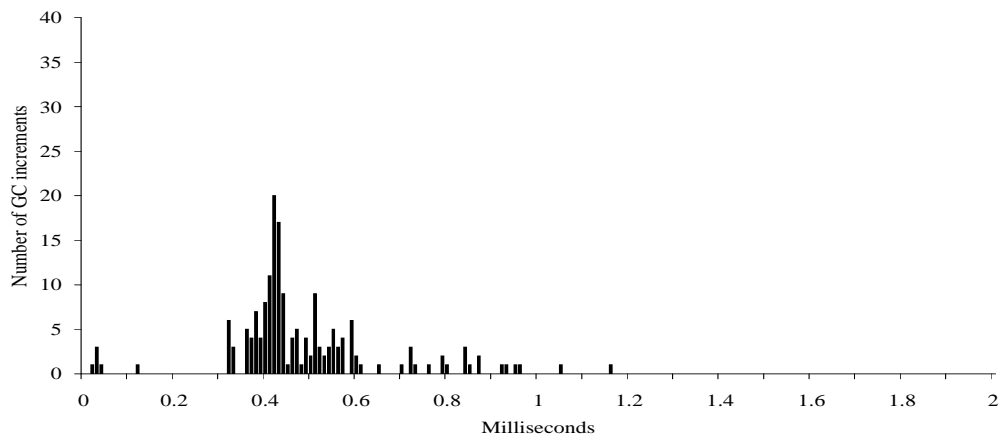


Figure 9: Histogram of the garbage collection increment costs for the Lindsay program (Throttle 2.0)

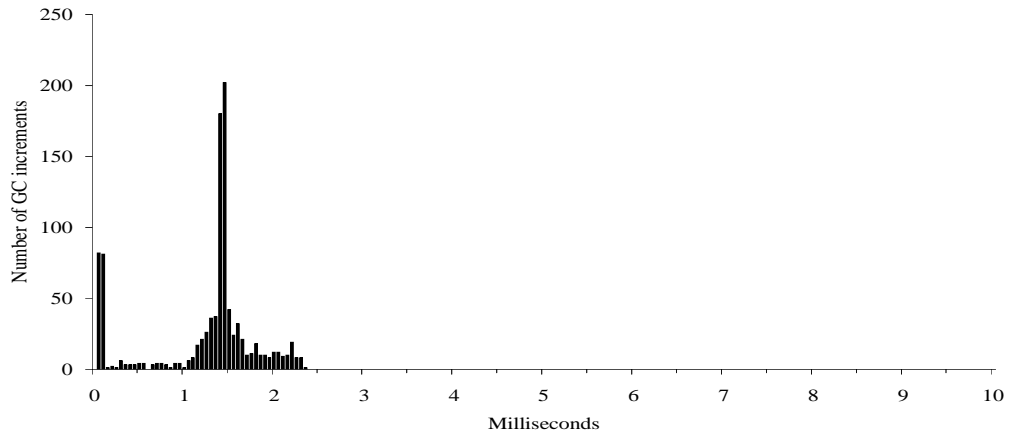


Figure 10: Histogram of the garbage collection increment costs for the Grobner program (Throttle 0.5)

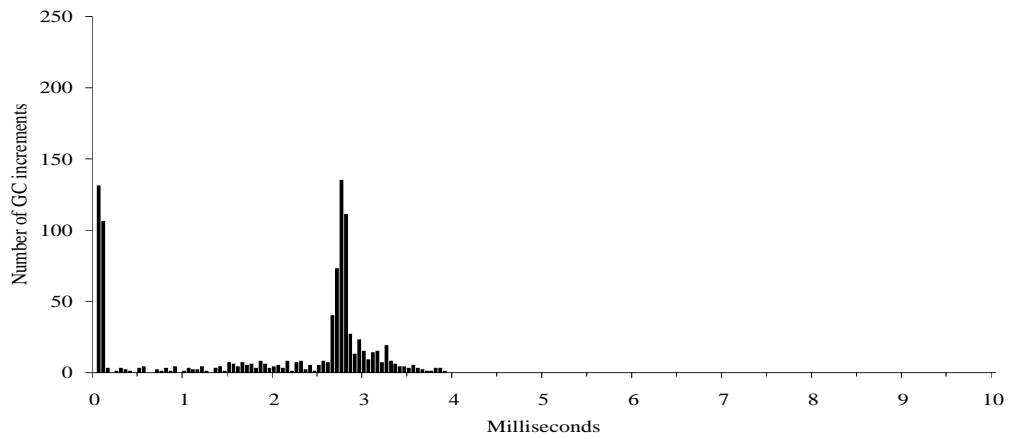


Figure 11: Histogram of the garbage collection increment costs for the Grobner program (Throttle 1.0)

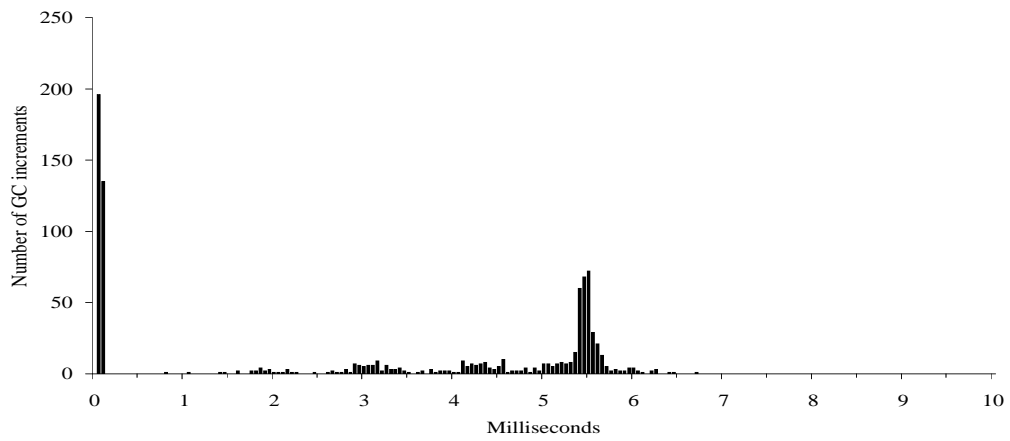


Figure 12: Histogram of the garbage collection increment costs for the Grobner program (Throttle 2.0)

bytes for every 8K bytes allocated. The x-axis shows the length of garbage collection pauses (in milliseconds) on a Pentium running Linux.<sup>19</sup> The minimum, average, and maximum pauses for this program were: 0.01, 0.17, and 0.35 milliseconds respectively. Figure 8 is a histogram of the garbage collection pauses for the same program tracing at a rate twice as fast (8K bytes for every 8K bytes allocated). The minimum, average, and maximum pauses for this program were: 0.01, 0.28, and 0.55 milliseconds respectively. Finally, Figure 9 is a histogram of the garbage collection pauses for the same program tracing 16K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were: 0.01, 0.48, and 1.15 milliseconds respectively.

Figure 10 is a histogram of garbage collection pauses for the Grobner program tracing 4K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were: 0.03, 1.21, and 2.30 milliseconds respectively. Figure 11 is a histogram of garbage collection pauses for the Grobner program tracing 8K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were: 0.03, 1.94, and 3.89 milliseconds respectively. Finally, Figure 12 is a histogram of garbage collection pauses for the Grobner program tracing 16K bytes for every 8K bytes allocated. The minimum, average, and maximum pauses for this program were: 0.03, 3.01, and 6.69 milliseconds respectively.

#### 5.8.4 More Accurate Timing Results

The above timing results were for C++ programs, and include many costs not directly related to the garbage collector. The most significant of these costs could be removed if the compiler supplied better support for garbage collection. In particular, the garbage collection increment costs include the time required to derive the start of objects. Languages like Scheme and Eiffel provide this support. As part of the proposed work for this dissertation, we will measure and isolate the actual costs for different real-time garbage collection strategies using our testbed collector combined with the RScheme compiler developed here at UT.

### 5.9 Memory Bounds

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

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

There are many different issues involved in calculating the space bounds for a non-copying algorithm, and all of the answers are not immediately clear. As part of the proposed work for this dissertation, we will be working out a number of different cases for memory bounds for different variations on real-time garbage collection. The current configuration of our testbed garbage collector uses the simple segregated storage mechanism. Below, we show how to compute the worst case memory bounds for this policy.

To calculate the worst case memory usage for a garbage collector using the simple segregated storage mechanism, the user must first know the maximum amount of live data for each size class. Given this information, the worst case memory usage for the program can be computed by determining the worst case memory usage for each size class and then summing these values.

The worst case memory usage for a single size class is simply the maximum amount of memory that can be live in that size class at any given time, plus twice the amount of memory that is allocated per full garbage collection cycle.<sup>20</sup> To see this, consider the case where the maximum number of objects for that size class are live, and we have just completed a garbage collection cycle. In the worst case, one new object will be allocated and immediately shaded, and then immediately freed making it garbage. This process repeats

---

<sup>19</sup>Timings are in milliseconds on a 90 MHz. Pentium running Linux 1.2.13, compiled with g++ 2.7.0 and full optimization. The cycle counts were recorded using the `rdtsc` opcode. This opcode returns the value of a 64-bit register that is incremented every instruction cycle. We are still experimenting with our timing routines, and these numbers should be taken as preliminary.

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

for the entire garbage collection cycle, and no other objects of any other size are allocated. In this case, none of the newly allocated objects will be reclaimable until the end of the next full garbage collection. We will need enough memory for the maximum live objects, and for every object that was allocated during *this and the next* full garbage collection. At the end of the next full garbage collection, the objects that become free during this garbage collection will be reclaimed and can be reused if this process continues.

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

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

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

### 5.9.1 Memory Computations for Eight Real C and C++ Programs

A disadvantage of simple segregated storage is that it can possibly suffer from severe memory fragmentation. We believe that in practice, for real programs, this will not be much of a problem. To get an idea of the amount of memory that would be needed to use simple segregated storage with real programs, we performed the above analysis on eight memory intensive C and C++ programs. These programs are the same programs that we used for our allocator studies, and are described in detail in Section C.2.

The first part of our analysis involves discovering the maximum number of objects of each size class that could be alive at the same time. Since we do not have deep understanding of the algorithms used by all eight of these programs, we substitute *measured* values. Although these values are by no means the maximum possible values, there are at least representative for a real workload, and it is reasonable to assume that these numbers will provide a flavor for what the actual worst case memory requirements will be. However, this method is not suggested as a substitute for real worst case analysis. Tables 1 and 2 show the maximum memory usage of our eight programs for each size class in the program *for the particular input set described in Appendix C*.

By looking at Tables 1 and 2 and rounding up some objects to a larger size class to reduce the number of different size classes, we can come up with an upper bound on the amount of memory that would be needed to run these eight programs with a hard real-time collector, *for this input set*.

Table 3 shows the amount of memory that would be required to run this program in the cases of tracing 1/2, 1, and 2 bytes for every byte allocated.<sup>23</sup> “Max Live” is the total maximum number of live bytes for any point in the program run; “Bytes Allocated” is the total number of bytes allocated in the run of the program; and “Mem Required .5”, “Mem Required 1”, and “Mem Required 2” is an upper bound on the amount of memory required to run the programs for a throttle setting of .5, 1, and 2 respectively.

Note that in some cases, the memory usage is startlingly poor. However, for the given input sets, these are strictly upper bounds computed without any eye towards optimizing the memory usage, and actual

---

<sup>21</sup> While at first glance, this would seem to only make things worse, what we are actually doing is trading internal fragmentation for external fragmentation—something that many common allocator algorithms do all of the time.

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

<sup>23</sup> We refer to these values as “throttle settings”. A throttle setting of 2 means that 2 bytes are traced for every byte allocated. The term “throttle setting” is meant to suggest the throttle on an engine. The higher the throttle setting, the faster things go.

Object Size (bytes)	16	32	64	128	256	512	1K	2K	4K
LRUsim	41	30	39010	0	9	0	1	0	1
cc1	14665	57204	11846	3420	172	18	73	18	18
espresso	10	98	4325	18	11	13	5	8	6
ghostscript	0	552	10472	2644	1317	318	36	4	33
grobner	3664	7185	483	36	80	19	4	1	0
lindsay	0	1	292	0	0	0	0	2	0
p2c	1383	5595	5336	1517	1	1	10	1	0
perl	377	248	1284	28	4	5	3	2	1

Table 1: Maximum number of live objects per size class

Object Size (bytes)	8K	16K	32K	64K	128K	256K	512K	1M	2M
LRUsim	1	1	0	0	0	0	0	0	0
cc1	18	18	18	4	1	4	4	0	0
espresso	8	7	2	4	0	0	0	0	0
ghostscript	2	2	4	0	0	0	0	0	0
grobner	0	1	0	0	0	0	0	0	0
lindsay	0	0	0	0	0	0	1	0	1
p2c	0	2	0	0	0	0	0	0	0
perl	1	1	0	0	0	0	0	0	0

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

memory use would most likely be much lower. If more effort is put into understanding the characteristics of the programs, particularly paying attention to objects which can be split into two or more smaller objects to eliminate a size class, and to phase behaviors which can cause use of one size class of objects to overlap the use of another, the amount of memory required can be made considerably smaller.

As part of the proposed work for this dissertation, we plan to use some of the tools described in Section 7 to simulate the garbage collector’s allocation behavior and measure what the actual memory usage of these eight programs would be if they were garbage collected. We hope that this will give some insight into the worst case memory computations for simple segregated storage, and eventually into developing methods for computing worst case memory usage for many different garbage collection policies.

## 5.10 Soft Real-time Programs

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

There are two different aspects of soft real-time that we will study:

1. *Soft time.* A soft time collector should be configured with an eye towards improving overhead, at the expense of very occasionally missing hard real-time bounds.
2. *Soft space.* A soft space collector should be configured with an eye towards improving memory usage, at the expense of very occasionally exceeding resource bounds.<sup>24</sup>

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

<sup>24</sup>Clearly, a program that exceeds its hard resource bounds is simply incorrect. However, a program that exceeds its soft resource bounds may be able to resort to paging to continue execution. It is this second since that we are interested in.

	Max Live	Bytes Allocated	Mem Required .5	Mem Required 1	Mem Required 2
LRUsim	1,413,565	1,430,400	14,006,240	8,352,628	5,524,858
cc1	2,376,053	18,403,819	60,022,800	32,611,098	22,622,153
espresso	269,647	106,892,887	5,067,060	3,449,050	2,309,771
ghostscript	1,136,887	50,169,140	21,736,560	12,641,464	8,349,660
grobner	148,537	4,081,352	2,527,404	1,636,182	1,190,603
lindsay	2,097,980	7,555,396	11,031,984	6,836,088	4,738,044
p2c	402,432	4,752,046	5,219,072	3,609,472	2,418,944
perl	71,327	33,844,674	1,083,508	655,546	441,565

Table 3: Memory needed to run real C and C++ programs

As part of the proposed work for this dissertation, we plan to measure the following variations in our garbage collector to test their suitability for use with a soft time or soft space application:

- Allocate new objects white.
- Use an incremental update write barrier
- Use a dynamically adjustable rate of garbage collection.

### 5.11 Adjusting the Rate of Garbage Collection

There are three related parameters to the garbage collector that need to be set by the user: the amount of memory to allocate per full garbage collection, the amount to allocate per garbage collection increment, and the amount of memory that should be traced per increment. We call this last parameter the garbage collector’s “throttle setting” and define it as the ratio of bytes traced to bytes allocated. For hard real-time applications, the user is required to determine the maximum number of live bytes that can occur at any point in the program. If the user chooses a throttle setting of .5, then two times max live bytes will be allocated per full garbage collection cycle. The user is free to set the number of bytes allocated per increment of collection as needed in order to meet the real-time bounds.<sup>25</sup>

### 5.12 Interface to C++

Currently, our collector uses a *smart pointer* interface to collect C++ [Str92]. In this interface, garbage collected objects have an associated pointer type defined in a library as a parameterized class, and client code must use these pointers rather than raw C++ pointers. (Parameterization and operator overloading make this relatively easy, although smart pointers cannot be used quite as flexibly as raw pointers [Ede92].) The main difference between our parameterized pointers and normal pointers is that pointer assignments execute an additional few lines of code, which constitute the write barrier.

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

Reading this header information from an object requires that the garbage collector always have access to the start of the object. In C++, however, there are many common cases where a pointer will point to the

<sup>25</sup> Actually, the computation is slightly more complicated if the user is using a snapshot-at-beginning write barrier. In this case, since the root set is atomically scanned at the beginning of collection, and because this is the only work done during this increment, one extra increment of garbage collection work is needed to complete collection cycle. So, if the user chooses a throttle setting of .5, then two times maximum number of live bytes, *plus* the number of bytes to allocate per increment is allocated per full garbage collection cycle.

middle of an object.<sup>26</sup> Because we use a segregated storage scheme, with all objects aligned on known word boundaries, it is relatively easy to derive the start of objects from these interior pointers.

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

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

## 6 Generational Collection

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

Because generational techniques rely on a heuristic—the guess that most objects will die young, and that older objects won't die soon—they are not strictly reliable, and may degrade collector performance in the worst case. For purely hard real-time systems, therefore, they may not be attractive. For general-purpose systems with mixed hard and soft deadlines, or for hard real-time systems with very regular periodic tasks, however, the normal-case efficiency gain is likely to be highly worthwhile.

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

### 6.1 Discussion

Generational collection can be combined with real-time techniques, but in the general case, the marriage is not a particularly happy one [WJ93]. Typically, generational techniques improve expected performance at the expense of worst-case performance, while real-time garbage collection is oriented toward providing absolute worst case guarantees. If the generational heuristic fails and most data are long-lived, garbage collecting the young generation(s) will be a waste of effort, because no space will be reclaimed. In that case, the full-scale garbage collection must proceed just as fast as if the collector were a simple, non-generational, incremental scheme.

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

When it is desirable to combine generational and incremental techniques, the details of the generational scheme may be important to enabling proper incremental performance. For example, the (Symbolics, LMI, and TI) Lisp machines' collectors are the best known “real-time” generational systems, but the interactions between their generational and incremental features turn out to have a major effect on their worst case performance. Rather than garbage collecting older generations slowly over the course of several collections

---

<sup>26</sup>Pointers may point to an element of an array or a substructure of a record. In addition, because of the usual C++ implementation strategy for multiple inheritance, a pointer may also point to a subcomponent of an object whose layout is a concatenation of the layouts of classes from which it is derived.

of younger generations, only one garbage collection is ongoing at any time, and that collection collects only the youngest generation, or the youngest two, or the youngest three, etc. That is, when an older generation is collected, it and *all younger generations* are effectively regarded as a single generation, and garbage collected together. This makes it impossible to benefit from younger generations' generational effect *while* garbage collecting older generations; in the case of a full garbage collection, it effectively degenerates into a simple non-generational incremental copying scheme. This will cause systematic performance losses during large scale garbage collections.

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

## 6.2 How to make a Generational Collector Real-Time

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

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

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

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

## 6.3 Object Advancement

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

---

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

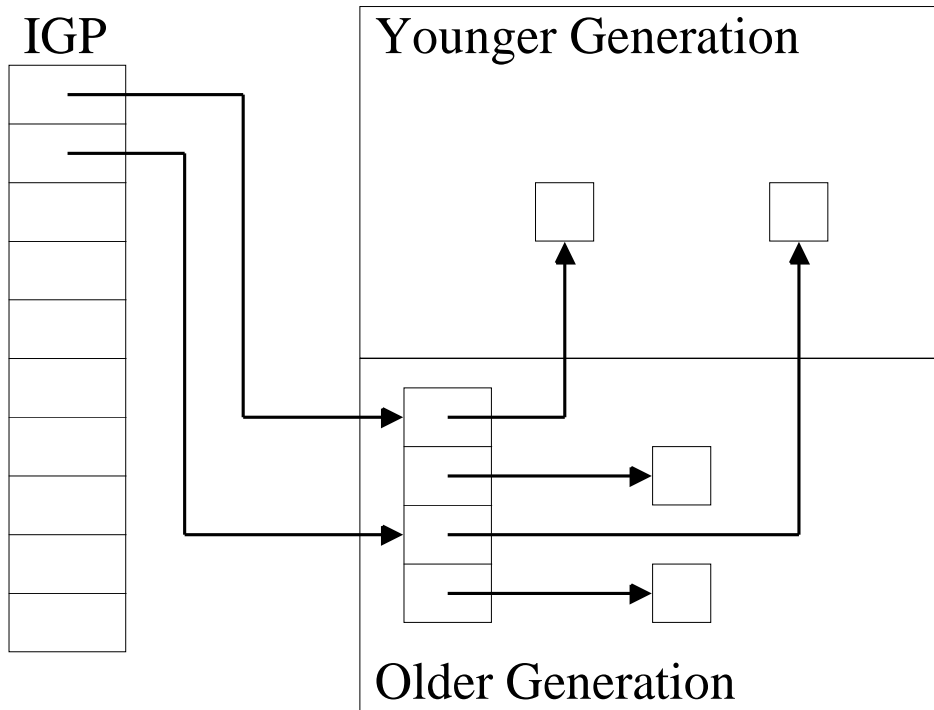


Figure 13: Example of the intergenerational pointer list

collection cycle in the youngest generation before being advanced.<sup>28</sup> This gives us some of the benefit of the counter without the associated cost. Once an object reaches the oldest generation, it simply remains there until it dies or the program terminates.

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

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

As part of the proposed work for this dissertation, we want to measure the actual costs of promoting objects black vs. white, and the relative effect of having an advancement counter on the objects.

## 6.4 Managing Intergenerational Pointers

The InterGenerational Pointer (IGP) list is implemented as a list of *pointers to the pointers* in old objects that point to young objects (see Figure 13). This is an important optimization. If a memory location in an object in the older generation is assigned many pointers to different younger objects, this extra level of indirection allows us to only trace the last object that is referenced by this memory location, and not all of the earlier objects.

If we were to naively record all such pointers to pointers from older objects to younger objects in the corresponding IGP list, this list would monotonically increase, making it virtually impossible to follow all of these pointers in any reasonable amount of time. We alleviate this problem by relying on the observation that objects can only spend a limited number of garbage collection cycles in the younger generation before they either die or are promoted. So, we actually manage a series of IGP lists between each older and younger generation. Each of these lists is associated with the garbage collection cycle of the younger generation in

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

which the IGP was created. So, if an object can remain in the younger generation for, say, at most three complete garbage collection cycles, then there will be three IGP lists. At the end of each cycle, the oldest IGP list is thrown away, and the second oldest IGP list becomes the oldest IGP list, and so on. The final complication is that the IGP list grows with the number of pointer assignments. This list can be kept to a bounded size by keying the rate of garbage collection to the rate of growth of this list, much like we do with the stored into list for the incremental update version of our write barrier.

## 6.5 Generational Timing Results

As a part of this proposed research, we intend to measure the performance of our generational collector to show that it is efficient relative to the running application, and that it provides acceptable soft real-time bounds on collector operations.

## 7 Memory Allocation Studies

As previously described, our garbage collector currently uses a simple segregated storage scheme to manage its heap. We made this design decision to support derived (interior) pointers in languages like C and C++ that do not have compiler support for garbage collection. For languages with better support for garbage collection such as Scheme or Eiffel there is no principled reason why we would necessarily have to manage the heap with simple segregated storage. Any *policy* will do as long as the *mechanism* used to implement it ensures that memory can be allocated in *bounded* time. In fact, at first glance, one might think that simple segregated storage will suffer considerably from both internal and external fragmentation.

We became interested in whether fragmentation is a real problem for simple segregated storage. In surveying the allocation literature we discovered that virtually all past work in this field suffered from one common flaw: almost no one measured how well different allocators performed for *actual* programs.<sup>29</sup>

The overwhelming majority of memory allocation studies to date have been based on a methodology developed in the 1960's [Col61], which uses synthetic traces intended to model "typical" program behavior. This methodology has the advantage that it is easy to implement, and allows experiments to avoid quirky behavior specific to a few programs. Often the people doing these studies went to great lengths to ensure that their traces had similar statistical properties to real programs. However, none of these studies showed the validity of using a randomly generated trace (no matter how well they statistically model the original program trace) to predict performance on real programs. As we show in Section 7.1.2 what all of this previous work ignores is that a randomly generated trace is *not* valid for predicting how well a particular allocator will perform on a real program.

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

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

The third reason is that, in a randomly generated trace, the relative proportions of different sized objects are relatively stable, whereas in a real trace, the relative proportions will change dramatically based on the program's phase.

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

Simply stated, an allocator's job is to *predict* the sizes and death times of objects that will be requested in the future. Using information from past memory requests, it chooses which free block of memory to allocate for the current request such that future requests will be able to be satisfied with a minimal amount

---

<sup>29</sup>The research done by Zorn[DDZ93, ZG92, Vo95] was the only work we found that used actual programs in their studies. They have made these programs, many of which we used, available by anonymous ftp. We intend to do the same with the additional programs we used.

Allocator	real waste	real est. frag	shuffled waste	shuffled est. frag
BestFit	33.41	14.34	36.90	20.23
SegrFits	42.41	26.08	61.63	46.36
DblBuddy	56.30	48.63	57.57	53.73
FirstFit	66.79	49.33	122.79	95.31
NextFit	72.23	52.41	144.25	115.48
BinBuddy	73.87	62.13	79.34	69.28
SegrStor2:3	72.76	72.76	63.43	63.47
SegrStorBin	84.82	84.63	72.23	72.03
Average	62.83	51.29	79.77	66.99

Table 4: Average Percentage Increase in Memory Use by Allocators

of fragmentation. But it is the very information that a well written allocator will use to predict these requests that is thrown out by a randomly generated trace.

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

We therefore decided to perform some simulation studies on various implementations of `malloc()` using memory allocation traces from real programs. Using a large set of tools that we built, we measured how well synthetic traces approximate real program traces, as well as how well these malloc algorithms performed on our real traces.

In the rest of this section, we will present some preliminary results from our memory allocation studies. In order to better understand these results, the reader not familiar with memory allocation policies may want to read our brief survey in Appendix A. The allocators used for these studies are described in detail in Appendix B, and the test programs we use are described in Appendix C.

## 7.1 Total Memory Usage

In Table 4 we present the amount of memory wasted by each of our allocators, as a percentage of the amount of live data at peak memory usage. We present figures both for overall waste, and estimates for waste due to fragmentation, where we have compensated for header and footer overheads.<sup>30</sup> The results are averaged across all eight traces, with separate averages for real and shuffled traces; they are rank ordered by their average fragmentation for real traces. We present the data in fragmentation order, because we believe it is most important to understand the interactions of policies with program behavior; per-object overheads are also interesting, but the best implementations of most of these policies will have similar overheads.

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

### 7.1.1 Best-fit is Best

The most striking result here is that among these basic allocators, the best-fit policy works best by far. The simple best fit allocator has only about 14% fragmentation, and the Lea g++ segregated fits approximation of best fit (SegrFits) has about 26%. Double Buddy is a distant third at about 49%.

Since our allocators round up to 8-byte aligned boundaries, these results are somewhat conservative. We would expect rounding to 8-byte boundaries to cost about 9 percent, given that the average object size of these programs is about 11 words. This suggests that best fit has a true fragmentation cost of only

<sup>30</sup>This compensation is done by having the simulator ask the `malloc()` routine for fewer bytes than the actual object size in the trace: fewer by the number of bytes in the header

3(SegrFits) allocator has only about 15therefore have several times less fragmentation than their nearest competitor.

This contrasts with traditional simulation results, where best fit usually performs well but is sometimes outperformed by next fit (e.g., in Knuth’s small but influential study [Knu73]). In terms of practical application, we believe this is one of our most significant findings. Since segregated fits implements an approximation of best fit fairly efficiently, it shows that a reasonable approximation of a best fit policy is both desirable and achievable.

### 7.1.2 Randomized Traces

The results for randomized traces differ in important ways from the results for real traces. Most importantly, the rank ordering of allocators is wrong in several places, and the average fragmentation for most allocators is also too high — by about 15%. (This is comparable to the total fragmentation for our best allocator.)

The rank ordering is crucial, since the major purpose of simulation studies is usually to “pick a winner.” The expected amount of memory lost to fragmentation is also important, however, because expected fragmentation is an important factor in deciding whether to use heap allocation or use static allocation, or to use some more restrictive allocation scheme (such as GNU “obstacks”) which exploit stereotyped usage patterns.

The correlation between real and shuffled results for total memory waste is about .50; for estimated fragmentation it is about .54.<sup>31</sup>

These correlations show that the results for shuffled runs are suggestive of the results for real programs. On the standard interpretation of correlation, however, the amount of variation in one set accounted for by regularities in the other set is proportional to the *square* of the correlation. This means that only 26% of the variation in wasted memory in the actual traces is accounted for by variation in the shuffled traces. Similarly, only 30% of the variation in estimated fragmentation is accounted for by variation in the shuffled traces. That is, for fragmentation, *the results of randomized experiments fail to account for more than two thirds of the observed variation due to real program behavior*. For both raw memory waste and fragmentation, simply randomizing the order of object creation (and keeping the size and lifetimes fixed) discards a large majority of the important information in a trace.

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

Randomization not only changes the rank ordering, but it changes some allocators’ rank dramatically—first fit and next fit move down 3 out of 8 places, and the simple segregated storage allocators move up 2 and 3 positions.

(Interestingly, the two best allocators keep their positions at the head of the pack, suggesting that the advantage of the best fit policy is unexpectedly robust across both patterned and randomized inputs.)

The low correlation between real and shuffled results shows that the random methodology is inaccurate for allocators’ average performance across our eight test traces. Examination of the results for individual allocators and individual traces shows extreme errors in some cases. For example, for one allocator, the fragmentation results for GCC are *thirteen times as high* for the shuffled trace (about 109%) as for real traces (about 8%).

## 7.2 Additional Studies

In this research, we intend to factor out the different memory allocation costs, and characterize the ways in which memory allocation algorithms are data-dependent, so that programmers can reason about how these dependencies will affect their program’s interaction with the memory allocator. In addition, we will

---

<sup>31</sup>We use population correlation, rather than straight correlation, because we are sampling the space of possible programs and program executions.

separate out the average (expected) case memory fragmentation costs from the worst case costs for a variety of situations.

### 7.3 Conclusions

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

### 7.4 How these results relate to garbage collection

It is clear from our studies that simple segregated store performs disappointingly poorly. On average, for our eight sample programs, segregated store wastes about 46% of its memory. Clearly, for languages with better support for garbage collection<sup>32</sup> an allocator such as best-fit would allow the program to run in far less memory. It is also important to note that our studies show that the best-fit *policy* performs well independent of the mechanism used to implement it. There are a number of ways to implement a best-fit-like policy that will likely have much lower memory wastage than segregated storage, while still allowing for real-time performance. We will be considering these ways for use in our collector. Finally, it should be noted that traditional copying collectors waste 50% of their memory in order to guarantee that there will be enough room to copy all live objects at every garbage collection. Looked at in this light, a simple segregated storage collector does not perform all that badly.<sup>33</sup> Still, we believe that real-time garbage collectors can be made to use much less memory while maintaining their real-time requirements.

## 8 Current Status

Our real-time garbage collector is fully implemented and has been tested using our C++ smart pointer interface. In addition, it has been integrated and tested with our Scheme system, and the Tower Eiffel compiler. Preliminary results are promising. Our initial, untuned implementation is relatively efficient. The results of running programs with our C++ smart pointer interface (where direct comparison with standard new/delete C++ code is possible) show a slow down from 10 to 90%.<sup>34</sup>

Our real-time garbage collector has also been extended with the generational techniques discussed in Section 6, and tested with our Scheme system, and the Tower Eiffel compiler.

Finally, we have implemented a large number of testing tools to analyze memory fragmentation for the eight programs described in Appendix C using the allocators described in Appendix B. To date, we have run over 900 different simulations to study fragmentation issues.

---

<sup>32</sup>Recall that we use a simple segregated store policy in our garbage collector to support derived pointers in C++.

<sup>33</sup>The 46% wastage number for segregated store does *not* include the header overhead that we incur with our implementation of non-copying implicit reclamation. If we factor in these extra 8 bytes of header, we find that our collector will waste about 51% of its memory. Statistically, this is indistinguishable from the wastage of a copying collector.

<sup>34</sup>Note that this overhead is measured in a high performance C++ system. Performance would appear better in a relatively slower system, such as most Lisp or Smalltalk implementations.

## A Survey of Memory Allocation Policies

Before we can discuss our experiments, we need to give a brief overview of allocator terminology.<sup>35</sup> The basic kinds of allocators we discuss are:

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

### A.1 Segregated Free Lists

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

One common variation is to use *size classes* to group similar sizes together, 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 size classes that are a power of two apart (e.g., 4 words, 8 words, 16 words, and so on) and round the requested size up to the nearest size class.

#### A.1.1 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.

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.

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 wait until the next allocation of the same size, and can be reallocated without splitting.

The disadvantage of this scheme is that it is subject to potentially severe external fragmentation as no attempt is made to split or coalesce blocks to satisfy requests for other sizes. The worst case is a program that allocates many objects of one size class and frees them, then does the same for many other size classes. In that case, separate storage is required for the maximum volume of objects of all sizes, and none can be reused for the others.<sup>36</sup>

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.)

A crude but possibly effective form of coalescing for simple segregated storage is to maintain a count of live objects for each page, and notice when a page is entirely 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.

---

<sup>35</sup>For a much more extensive discussion on these issues, see [WJNB95]

<sup>36</sup>A simple enhancement can support splitting: when a free list for a requested size is empty, a suitable larger block may be found by searching free lists for larger sizes. In most circumstances, however, this will require a header word block to record the sizes of allocated blocks so that they can be freed (i.e., put on the proper free list). This per-block size recording is required because splitting results in a mix of block sizes within a page.

### A.1.2 Segregated Fit.

This variant uses an array of free lists, with each array element 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 fit search, and many significant variations are possible (see below). Typically a first-fit or next-fit policy is used. It is often pointed out that the use of multiple free lists makes the implementation faster than searching a single free list. What is often *not* appreciated is that this also affects the *policy* in a very important way: the use of segregated lists excludes blocks of very different sizes, meaning *good* fits are usually found. The policy is therefore a *good fit* or even a *best fit* policy, despite the fact that it's usually described as a variation on first fit.

## A.2 Buddy Systems

Buddy systems [Kno65, PN77] are a variant of segregated lists that support 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, and how they may be coalesced into larger free areas. A free area may only be merged with its *buddy*, which is the corresponding block at the same level in this hierarchical division. The resulting free block is therefore always one of the free areas at the next higher level in the memory-division hierarchy. At any level, the first block of a buddy pair may only be merged with the following block of the same size; conversely, the second block of a buddy pair may only be merged with the first, which precedes it in memory. This constraint on coalescing ensures that the resulting merged free area will always be aligned on one of the boundaries of the hierarchical splitting of memory.

The purpose of the buddy allocation constraint is to ensure that when a block is freed, its (unique) buddy can always be found by a simple address computation, and its buddy will always be 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 buddy's header — it will never find the middle of an allocated object.

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

Several significant variations on buddy systems have been devised:

### A.2.1 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.

A major problem with binary buddies is that internal fragmentation is usually relatively high—the expected case is about 28%, because any object size must be rounded up to the nearest power of two (minus a word for the header, if a bit cannot be stolen from the block given to the language implementation).

### A.2.2 Double buddies.

Double buddy systems [Wis78, PH86] use a different technique to allow a closer spacing of size classes. They use two different buddy systems, with staggered sizes. For example, one buddy system may use powers-of-two sizes (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 ...). Request sizes are rounded up to the nearest size class in either series. This reduces the internal fragmentation by about half, but means that a block of a given size can only be coalesced with blocks in the same size series. 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 the rules of that series. (This complicates the treatment of large objects.) To our knowledge, the implementation we built for the present study may actually be the only double buddy system in existence, though Page wrote a simulator that is almost an entire implementation of a double buddy allocator [PH86].

### A.3 Sequential Fits

Several classic allocator algorithms are based on having a single doubly-linked linear (or circularly linked) list of all free blocks of memory. Typically, sequential fit algorithms use Knuth's *boundary tag* technique to support coalescing of all adjacent free areas [Knu73].

#### A.3.1 First-fit

A first fit policy simply searches the list from the beginning, and uses the first free block large enough to satisfy the request. If the block is larger than necessary, it is split and the remainder is put on the free list. A problem with first fit is that the larger blocks near the beginning of the list tend to be split first, and the remaining fragments result in having a lot of small blocks near the beginning of the list. This can increase search times because many small free blocks accumulate, and the search must go past them each time a larger block is requested. First fit therefore may not *scale* well to systems in which many objects are allocated and many free blocks accumulate. In terms of policy, a first fit policy may tend over time toward behaving rather like best fit, because the smallest blocks end up near the front of the list, so that blocks are effectively searched in size order, and the smallest chosen first.<sup>37</sup>

#### A.3.2 Next-fit

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

#### A.3.3 Best-fit

A best-fit sequential fit allocator searches the free list to find the smallest free block large enough to satisfy a request. The basic strategy here is to minimize the amount of wasted space by ensuring that fragments are as small as possible. This strategy may backfire in practice, however, if the small fragments are reused for longer-lived objects, and prevent the coalescing of large free areas. 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.

#### A.3.4 Boundary Tags and Per-Object Overheads

Sequential fit techniques are usually implemented using *boundary tags* to support the coalescing of free areas [Knu73]. Each block of memory has a header and a footer field, both of which record the size of the block and whether it is in use. When a block is freed, the footer of the preceding block of memory is

---

<sup>37</sup>This has also been observed by Ivor Page (personal communication, February 1994).

examined to see if it is free; likewise, the header of the following block is examined. Adjacent free areas are merged into larger free blocks. (This is where the doubly-linked lists are useful—a block can be unlinked from anywhere in a doubly-linked list in constant time.)

### A.3.5 Discussion

The sequential fit algorithms have many possible variations. The usual implementations push freed blocks onto the front of the free list, where they will be the next considered for reuse. (In the case of next fit, this only works if the next request can be satisfied by that block; otherwise the roving pointer will rove past it.) A common variation is to impose a *splitting threshold*, so that blocks will not be split if they are already small. Blocks are generally not split if the resulting remainder is smaller than the minimum allocation size (big enough to hold the header and footer plus the free list link(s).) In addition, a block may not be split if the remainder is “too small” relative to the size of the object being allocated. This policy is intended avoid allocating a small object there that will prevent the coalescing of large free areas.

Another important variation is the use of sorted lists, typically sorted by address order. This encourages the reuse of objects near one end of the heap area, so that free areas within that region are filled first. If many objects live a long time, eventually the long-lived objects may accumulate in the lower address range and result in better memory use there. The problem with a sequential fit implementation using sorted addresses is that list insertion times are proportional to the number of free blocks of memory, rather than constant time. Results for such implementations are still interesting, however, because similar policies may be implemented by more efficient mechanisms.

More generally, an important point in understanding sequential fit is that a linear list supports *mechanisms* that have strange *policy* consequences. For example, best fit is really a *policy* (i.e., use the free block that fits best) which admits many different implementations that may be much faster than an exhaustive sequential search.

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. 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. On the other hand, low fragmentation can also increase splitting and coalescing costs if deferred coalescing is not used.

## B Allocator Descriptions

We obtained (or constructed) a variety of allocators, representative of the classes of allocators we described earlier: sequential fit, simple segregated storage, segregated fit, and buddy systems.

Descriptions of our basic allocators follow; unless otherwise noted, object sizes are rounded up to the nearest word (4 bytes or 32 bits) and the minimum object size is four words (16 bytes). Memory is requested from the operating system in units of 4KB, except for double buddy, which requests an average of 5KB at a time.<sup>38</sup> We believe that detailed descriptions are important—as we have found, seemingly minor variations on these algorithms can have major performance consequences.

- *SegrStorBinary* is a very simple segregated storage algorithm that does no coalescing. It maintains an array of free lists for size classes that are powers of two. There is no header or footer overhead, because no coalescing is done, and because all objects in a page are of the same size.<sup>39</sup> Expected average internal fragmentation is about 28%. Minimum object size is 8 bytes. (Originally written by Sheetal Kakkad for use in the Texas Persistent Store [SKW92], but very similar to the widely used and venerable BSD UNIX allocator by Chris Kingsley and studied by Zorn and Grunwald.)
- *SegrStor2:3* is very similar to *SegrStorBinary*, but the size classes are closer together, to decrease internal fragmentation at a possible expense in external fragmentation. Size classes are powers of

---

<sup>38</sup>Recall that double buddy actually uses two heap areas. In one heap area memory is requested from the operating system in units of 4KB, and in the other, memory is requested in units of 6KB

<sup>39</sup>Only one word is required per page (about a tenth of a percent of the heap size) for encoding the sizes of objects in a page to support freeing objects by putting them on the appropriate free list. We ignored this cost because it is negligible given the slight imprecision in our measurements.

two, plus intermediate size classes that are powers of two, times three. The minimum object size is 8 bytes. A simple table lookup technique is used to make size class determination fast for small objects. Expected internal fragmentation is about 14%. (Another version of the Texas allocator, implemented by Sheetal Kakkad and Michael Neely.)

- *First-fit* is a classic first-fit algorithm using a single free list and Knuth’s boundary tag technique. A one-word header and one-word footer are used to support coalescing.<sup>40</sup> Note: no roving pointer is used—the free list is searched from the beginning, which is also where freed objects are returned to the list. Remainders from splitting are left in the list position where the split block was found. When memory is requested from the operating system (in 4KB pages, or multiples of that for large objects), the remainder is put at the front of the free list as well. Minimum object size is 16 bytes. (Based on code from Douglas Lea’s g++ allocator, extracted and modified by Michael Neely.)
- *Next-fit* is a modified First-fit, using a roving pointer to avoid searching the list from the beginning each time, in an attempt to prevent the accumulation of small fragments at the beginning of the list. Freed blocks are inserted into the free list just ahead of the search pointer. (Based on the Lea/Neely First-fit code, with the same basic implementation details.)
- *Best-fit* is another modification of the First-fit code, with one-word headers and footers for boundary-tag coalescing. The free list is searched exhaustively or until an exact fit is found. This is not intended to be a realistic mechanism, but rather a test of the best fit policy. (Another variant on the Lea/Neely First-fit code.)
- *BinaryBuddy* is a classic binary buddy system, extracted from a large object storage manager in a garbage collector for Standard ML of New Jersey. One header word per block is used, to record the block size and whether is in use. Minimum object size is 16 bytes. (Code provided by Greg Morrisett, CMU)
- *DbdBuddy* is a “double buddy” system, using a pair of buddy systems to manage memory for two different (staggered) sets of power-of-two size classes. One buddy system manages memory for size classes that are powers of two, (i.e., 4, 8, 16, 32, . . . (words)) the other for powers of two times three (i.e., 6, 12, 24, 48, . . . (words)). In this implementation, memory reclaimed in one buddy system is not available for use in the other, sometimes limiting the effectiveness of coalescing. (Optimizations are possible that may allow some memory freed by one buddy series to be made available for the other.)
- *SegrFit* is Douglas Lea’s “g++” allocator, widely distributed and used with the GNU C++ compiler. It is a “segregated storage” algorithm in the (rather misleading) sense of Purdom, Stigler, and Cheam [PSC71]. Actual storage is *not* segregated, and one-word header and footer fields support boundary-tag coalescing. A set of free lists is maintained, “segregating” (indexing) free objects by approximate size so as to speed up searches. Size classes are powers of two divided linearly in 4. (Powers of two give a logarithmic set of size classes, and those sets are subdivided into 4 smaller ranges by simple linear division, i.e., 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, 28, . . . (words).) Each of the resulting size classes has a conventional doubly-linked free list searched using first-fit. Several optimizations support a limited form of deferred coalescing and deferred reuse. Note that despite using a first-fit mechanism, the use of fairly precise size classes ensures that it implements a policy that is very close to best fit.<sup>41</sup> (This has generally been overlooked.) Minimum object size is 16 bytes. This allocator, unlike the others, enforces 8-byte alignment (rather than 4-byte), thus its expected performance is a little worse than it would be if it were modified to ensure only 4-byte alignment for fairer comparisons.

We will now present results for our allocators on real and randomized traces. The results for real traces are significant in terms of actual expected performance, and the results for randomized traces are significant in evaluating the traditional methodology.

---

<sup>40</sup>When a block is split to satisfy a request, the result is always put on the free list if it is at least 4 words, i.e., large enough to hold a one-word object. No other “splitting threshold” is used to trade internal fragmentation for reduced external fragmentation.

<sup>41</sup>Actually, it is very close, because the “first fit” search within a size class looks for a very good fit (within less than the minimum object size) and forces coalescing if one is not found.

## C The Test Programs

For our test programs, we used a eight varied C and C++ programs that run under UNIX. These programs allocate between about 1.3 and 104 megabytes of memory during a run, and have a maximum of between 69 KB and 2.3 MB of live data at some point during execution. (On average they allocate 27 MB total data and have a maximum of about 966K live data at some point.) Three of our eight programs were used by Zorn and Grunwald *et al.* in earlier studies; we have attempted to provide some points of comparison while also using mostly new and different memory-intensive programs.

### C.1 Test Program Selection Criteria

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

More importantly, some of our allocators are likely to incur extra overhead for small heap sizes, because they allocate in more than one area; they may have several partly-used pages, and unused portions of those pages may have a pronounced effect when heap sizes are very small. We think that such relatively fixed costs are less significant than their scalability to medium and large-sized heaps.<sup>42</sup>

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

We ruled out some programs that appeared to “leak” memory, i.e., fail to discard objects at the proper point, and lead to a monotonic accumulation of garbage in the heap. One of our programs, the P2C program is known to leak under some circumstances, and we left it in after determining that it could not be leaking outrageously during the run we traced. (Its basic memory usage statistics are not out of line with our other programs—it deallocates over 90% of all allocated bytes, and its average object lifetime is lower than most.) Our justification for this is that many programs do in fact leak, so having one in our sample is not unreasonable. It is a fact of life that deallocation decisions are often extremely difficult for complex programs, and programmers often knowingly choose to let programs leak on the assumption that over the course of a run the extra memory usage is acceptable.<sup>43</sup> They choose to have poorer resource usage because attempts at plugging the leaks often result in worse bugs—dereferencing dangling pointers and corrupting of data structures.

We should note here that in choosing our set of traces, among the traces we excluded were three that did very little freeing, i.e., all or nearly all allocated objects live until the end of execution.<sup>44</sup> (Two were the PTC and YACR programs from Zorn *et al.*'s experiments.) We believe that such traces are less interesting because any good allocator will do well for them. This biases our sample slightly toward potentially more problematic traces, which have more potential for fragmentation, and away from allocators which have no header overheads. (Our suite does include one almost non-freeing program, LRUsim, which is the only non-freeing program we had that we were sure did not leak.<sup>45</sup> We do not believe that this is a large problem,

---

<sup>42</sup>Two programs used by Zorn and Grunwald [ZG92] and by Detlefs, Dosser, and Zorn [DDZ93] have heaps that are quite small—Cfrac only uses 21.4 KB and Gawk only uses 41 KB, which are only a few pages on most modern machines. Measurements of CPU costs for these programs are interesting, because they are allocation-intensive, but measurements of memory usage are less useful, and potentially obscure scalability issues with boundary effects.

<sup>43</sup>One very memory-intensive program we wanted to use (but did not) had leaks that survived three months of highly-skilled programmers' attempts at fixing it. Rather than restructuring their entire program and losing much of its modularity solely to allow objects to be correctly allocated, they eventually chose to use the Boehm-Weiser conservative garbage collector.

<sup>44</sup>Other programs were excluded because they had too little live data (e.g., LaTeX), or because we could not easily figure out whether their memory use was hand-optimized, or because we judged them too similar to programs we did choose.

<sup>45</sup>LRUsim actually must retain almost all allocated objects for the duration of a run, because it cannot tell—even in principle—which ones will be needed again.

program	KB alloc'd	run time	max obj's	max KB	avg lifetm	language	purpose
Espresso	104,388	146	4,390	263	15,478	C	PLA optimizer
GCC	17,972	167	86,872	2,320	926,794	C	compiler
Ghostscript	48,993	53	15,376	1,110	786,699	C	page renderer
Grobner	3,986	8	11,366	145	173,170	C++	Grobner basis
Hyper	7,378	131	297	2,049	10,531	C++	h'cube msg. sim.
LRUsim	1,397	29,940	39,039	1,380	701,598	C++	locality analyzer
P2C	4,641	30	12,652	393	187,015	Pascal/C	Pascal->C
Perl	33,041	114	1,971	69	39,811	Perl/C	formatting words
Average	27,725	3,823	21,495	966	355,137	—	—

Table 5: Basic Statistics for the Eight Test Programs

however, because good allocators will generally do well for such programs.

## C.2 The Selected Test Programs

We used eight programs because this is sufficient to obtain statistical significance for our major conclusions. (Naturally it would be better to have even more, but for practicality we limited the scope of these experiments to eight programs and a comparable number of allocators to keep the number of combinations reasonable.) Whether are programs are “representative” is a difficult subjective judgment; we believe they are reasonably representative of applications in conventional, widely-used languages (C and C++).

Table 5 gives some basic statistics for each of our eight test programs. The *KB alloc'd* column gives the total allocation over a whole run, in kilobytes; the *run time* column gives the running time in seconds on a Sun SPARC ELC, an 18.2 SPECint processor, when linked with the standard SunOS allocator. (This is a Cartesian-tree based “better fit” (indexed fits) allocator.) The *max obj's* column gives the maximum number of live objects, and the *max KB* column gives the maximum number of live bytes; note that these may not occur at the same point in a trace, if the average size of objects varies over time. The *avg lifetime* column gives the average number of bytes allocated between the birth and death of an object, weighted by the size of the object. (That is, it's really the average lifetime of an allocated byte of memory.)

In this paper, when we present averages, they are generally simple averages weighting each of our test programs equally—unless what is being presented is total memory allocation or memory usage, we normalize the numbers so that the more allocation-intensive and memory-intensive programs do not dominate the averages. This is based on an assumption that the other features of traces are not strongly correlated with the total allocation, and that better representativeness is achieved in this way. On the other hand, our sample is already biased toward memory-intensive programs, so in that sense the results are weighted toward such programs from the outset.

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

- *Espresso* is a widely used optimizer for programmable logic arrays. The file `largest.espresso` provided by Ben Zorn was used as the input.
- *GCC* is the main process (`cc1`) of the GNU C compiler (version 2.5.1). We constructed a custom tracer that records `obstack`<sup>46</sup> allocations to obtain this trace, and built a postprocessor to translate the use of `obstack` memory into equivalent `malloc()` and `free()` calls.<sup>47</sup> The input data for the compilation was the the largest source file of the compiler itself (`combine.c`).<sup>48</sup>

<sup>46</sup>Obstacks are an extension to the C language optimized for allocating and deallocating objects in stack-like ways. (A similar scheme is described in [Han90].)

<sup>47</sup>It is our belief that we should study the behavior of the program without hand-optimized memory allocation, because a well-designed allocator should usually be able to do as well or better than most programmers hand-optimizations. Some support for this idea comes from [Zor93], which showed that hand optimizations usually do little good compared to choosing the right allocator.

<sup>48</sup>Because of the way the GNU C compiler is distributed, this is in fact the most common workload—people frequently down-load a new version of the compiler and compile it with an old version, then recompile it with itself twice as a cross-check

- *Ghost* is GhostScript, a widely-used portable PostScript (page rendering language) interpreter written by Peter Deutsch, and modified by Zorn *et al.* to remove hand-optimized memory allocation [Zor93]. The input was `manual.ps`, the largest of the standard inputs available from Zorn's ftp site. This document is the 127-page manual for the Self system, consisting of a mix of text and figures.<sup>49</sup>
- *Grobner* is (to the best of our very limited understanding) a program that rewrites a mathematical function as a linear combination of a fixed set of Grobner basis functions.<sup>50</sup>
- *Hyper* is a hypercube network communication simulator written by Don Lindsay. It builds a representation of a hypercube network, then simulates random messaging, accumulating statistics about messaging performance. The hypercube itself is represented as a large array, which lives for essentially the entire run; each message is represented by a small heap-allocated object, which lives very briefly—only long enough for the message to get where it's going, which is a tiny fraction of the length of the run.
- *LRUsim* is an efficient locality analyzer written by Douglas Van Wieren. It consumes a memory reference trace and generates a grey-scale PostScript plot of the evolving locality characteristics of the traced program. Memory usage is dominated by a large AVL tree<sup>51</sup> which grows monotonically. A new entry is added whenever the first reference to a block of memory occurs in the trace. Input was a reference trace of the P2C program.<sup>52</sup>
- *P2C* is a Pascal-to-C translator, written by Dave Gillespie at Caltech. The test input was `mf.p` (part of the Tex release). Note: although this translator is from Zorn's program suite, this is *not* the same Pascal-to-C translator (PTC) they used in their studies. This one allocates and deallocates more memory, at least for this input.
- *Perl* is the Perl scripting language interpreter (version 4.0) interpreting a Perl program that manipulates a file of strings. The input, `adj.perl`, formatted the words in a dictionary into filled paragraphs. Hand-optimized memory allocation was removed by Zorn [Zor93].

---

to ensure that the generated code does not change between self-compiles (i.e., it reaches a fixed point).

<sup>49</sup>Note that this is not the same input set as used by Zorn *et al.* in their experiments; they used an unspecified combination of several programs. We chose to use a single, well-specified input file to promote replication of our experiments.

<sup>50</sup>Abstractly, this is roughly similar to a Fourier analysis, decomposing a function into a combination of other, simpler functions. Unlike a Fourier analysis, however, the process is basically one of rewriting symbolic expressions many times, something like rewrite-based theorem proving, rather than an intense numerical computation over a fixed set of array elements.

<sup>51</sup>The AVL tree is used to implement a least-recently-used ordering queue. The AVL tree implementation was enhanced to maintain a count at each node of the descendents to the left of the node; this allows computing the LRU queue position of a node in logarithmic time, as well as supporting logarithmic time deletion and insertion to move a node to the beginning of the queue when the block it represents is referenced.

<sup>52</sup>The memory usage of LRUsim is not sensitive to the input, except in that each new block of memory touched by the traced program increases the size of the AVL tree by one node. The resulting memory usage is always nondecreasing, and no dynamically allocated objects are ever freed except at the end of a run. We therefore consider it reasonable to use one of our other test programs to generate a reference trace, without fearing that this would introduce correlated behavior. (The resulting fragmentation at peak memory usage is insensitive to the input trace, despite the fact that total memory usage depends on the number of memory blocks referenced in the trace.)

## References

- [AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation* [PLD88], pages 11–20.
- [AM92] Antonio Albano and Ron Morrison, editors. *Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992. Springer-Verlag.
- [AP87] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In E. Chiricozzi and A. D’Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–246, L’Aquila, Italy, September 1987. Elsevier North-Holland.
- [Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.
- [Bak91] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA ’91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper. Also appears as *SIGPLAN Notices* 27(3):66–70, March 1992.
- [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.
- [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.
- [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.
- [Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [LFP84], pages 108–113.
- [Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.
- [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.
- [Coh81] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.
- [Col61] G. O. Collins. Experience in automatic storage allocation. *Communications of the ACM*, 4(10):436–440, October 1961.
- [DDZ93] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, August 1993.
- [Det90a] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, May 1990.
- [DeT90b] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, Palo Alto, California, August 1990.

- [DLM<sup>+</sup>78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.
- [Ede92] Daniel Ross Edelson. Smart pointers: They’re smart, but they’re not pointers. In *USENIX C++ Conference [USE92]*, pages 1–19. Technical Report UCSC-CRL-92-27, University of California at Santa Cruz, Baskin Center for Computer Engineering and Information Sciences, June 1992.
- [EV91] Steven Engelstad and Jim Vandendorp. Automatic storage management for systems with real time constraints. In *OOPSLA ’91 Workshop on Garbage Collection in Object-Oriented Systems [OOP91]*. Position paper.
- [FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.
- [Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1), January 1990.
- [Hay91] Barry Hayes. Using key object opportunism to collect old objects. In Paepcke [Pae91], pages 33–46.
- [Joh92] Ralph E. Johnson. Reducing the latency of a real-time garbage collector. *ACM Letters on Programming Languages and Systems*, 1(1):46–58, March 1992.
- [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.
- [LFP84] *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984. ACM Press.
- [LFP90] *Conference Record of the 1990 ACM Symposium on LISP and Functional Programming*, Nice, France, June 1990. ACM Press.
- [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.
- [McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.
- [Moo84] David Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming [LFP84]*, pages 235–246.
- [Nil88] Kelvin Nilsen. Garbage collection of strings and linked data structures in real time. *Software Practice and Experience*, 18(7):613–640, July 1988.
- [NOPH92] Scott Nettles, James O’Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [BC92], pages 357–364.
- [NR95] Bob Nimon and John Radford. Implementing a real-time, embedded, telecommunications switching system in smalltalk, October 1995. experience report at OOPSLA 95.
- [NS90] Kelvin Nilsen and William J. Schmidt. A high-level overview of hardware assisted real-time garbage collection. Technical Report TR 90-18a, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1990.
- [OOP89] *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA ’89) Proceedings*, New Orleans, Louisiana, 1989. ACM Press.

- [OOP91] *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/GC91.
- [OOP93] *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, October 1993. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/GC93.
- [Pae91] Andreas Paepcke, editor. *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, Phoenix, Arizona, October 1991. ACM Press. Published as *SIGPLAN Notices* 26(11), November 1991.
- [PH86] Ivor P. Page and Jeff Hagins. Improving the performance of buddy systems. *IEEE Transactions on Computers*, C-35(5):441–447, May 1986.
- [PLD88] *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988. ACM Press.
- [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.
- [PN77] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.
- [PSC71] P. W. Purdom, S. M. Stigler, and Tat-Ong Cheam. Statistical investigation of three storage allocation algorithms. *BIT*, 11:187–195, 1971.
- [RK68] Brian Randell and C. J. Kuehner. Dynamic storage allocation systems. *Communications of the ACM*, 12(7):297–306, May 1968.
- [SES84] *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, April 1984. Published as *ACM SIGPLAN Notices* 19(5), May, 1987.
- [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.
- [SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In Albano and Morrison [AM92], pages 11–33.
- [Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.
- [Str92] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1992.
- [Ung84] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* [SES84], pages 157–167. Published as *ACM SIGPLAN Notices* 19(5), May, 1987.
- [USE92] USENIX Association. *USENIX C++ Conference*, Portland, Oregon, August 1992.
- [Vo95] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 1995. To appear.
- [Wan89] Thomas Wang. MM garbage collector for C++. Master's thesis, California Polytechnic State University, San Luis Obispo, California, October 1989.
- [Wil88] Paul R. Wilson. Opportunistic garbage collection. *SIGPLAN Notices*, 23(12):98–102, December 1988.

- [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.
- [Wit91] P. T. Withington. How real is “real time” garbage collection? In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper.
- [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* [OOP93]. Expanded version of workshop position paper submitted for publication.
- [WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.
- [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* [OOP89], pages 23–35.
- [Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 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.
- [Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conference Record of the 1990 ACM Symposium on LISP and Functional Programming* [LFP90], pages 87–98.
- [Zor93] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.