QUANTIFYING AND IMPROVING THE PERFORMANCE OF GARBAGE COLLECTION

A Dissertation Presented

by

MATTHEW HERTZ

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2006

Computer Science

QUANTIFYING AND IMPROVING THE PERFORMANCE OF GARBAGE COLLECTION

A Dissertation Presented

by

MATTHEW HERTZ

Approved as to style and content by:

To Sarah for reminding me of everything I can do and to Shoshanna for inspiring me to do more.

ACKNOWLEDGMENTS

I am most grateful to my advisor, Emery Berger, for everything he has done throughout this thesis. I appreciate his guidance, suggestions, and inspiration. I feel especially fortunate for the patience he has shown with me throughout all the twists and turns my life took getting through this dissertation.

I must also thank Eliot Moss and Kathryn McKinley for their leadership and support. I will be forever grateful that they took a chance on a student with a less-than-stellar academic record and provided me with a fertile, inspiring research environment. They are both very knowledgeable and I benefited from our discussions in a myriad of ways. They have also served as members of my committee and I appreciate their helpful comments and suggestions. Thanks also to Scott Kaplan, another member of my committee, for his advice and feedback.

I would also like to thank all of the members of the ALI and PLASMA research groups over the years, especially Steve Blackburn, Alistair Dundas, Yi (Eric) Feng, Chris Hoffmann, Asjad Khan, and Zhenlin Wang, who always lent an ear when I needed to talk out my latest idea or blow off steam. I also owe thanks to Darko Stefanović and Amer Diwan. The thoughts and insights these gentlemen shared with me have helped in many ways, both big and small.

I am also indebted to my parents for their love and support over the years. They taught me the value of my ideas and the importance of staying true to my convictions. The love and support I received from my brother was also important. I could not have done this without them.

Finally, I want to thank Sarah and Shoshanna for everything they have done. They have shown an unbounded amount of love and support. I could not have asked for a more wonderful family.

ABSTRACT

QUANTIFYING AND IMPROVING THE PERFORMANCE OF GARBAGE COLLECTION

SEPTEMBER 2006

MATTHEW HERTZ

B.A., CARLETON COLLEGE

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Emery D. Berger

Researchers have devoted decades towards improving garbage collection performance. A key question is: what opportunity remains for performance improvement? Because the perceived performance disadvantage of garbage collection is often cited as a barrier to its adoption, we propose using explicit memory management as the touchstone for evaluating garbage collection performance.

Unfortunately, programmers using garbage-collected languages do not specify when to deallocate memory, precluding a direct comparison with explicit memory management. We observe that we can simulate the effect of explicit memory management by profiling applications and generating object reachability information and lifetimes to serve as zero-cost oracles to guide explicit memory reclamation. We call this approach *oracular memory management*. However, existing methods of generating exact object reachability are prohibitively expensive. We first present the object reachability time algorithm, which we call *Merlin*. We show how Merlin runs in time proportional to the running time of the application and can be performed either on-line or off-line. We then present empirical results showing that Merlin requires orders of magnitude less time than brute force trace generation.

We next present simulation results from experiments with the oracular memory manager. These results show that when memory is plentiful, garbage collection performance is competitive with explicit memory management, occasionally increasing program throughput by 10%. This performance comes at a price. To match the average performance of explicit memory management, the best performing garbage collector requires an average heap size that is two-and-a-half or five times larger, depending on how aggressively the explicit memory manager frees objects.

This performance gap is especially large when the heap exceeds physical memory. We show the garbage-collector causes poor paging performance. During a full heap collection, the collector scans all reachable objects. Which a reachable object resides on a non-resident page, the virtual memory manager reloads that page and evicts another. If the evicted page also contains reachable objects, ultimately it too will be accessed by the collector and reloaded into physical memory. This paging is unavoidable without cooperation between the garbage collector and the virtual memory manager.

We then present a cooperative garbage collection algorithm that we call *bookmarking collection* (BC). BC works with the virtual memory manager to decide which pages to evict. By providing available empty pages and, when paging is inevitable, processing pages prior to their eviction, BC avoids triggering page faults during garbage collection. When paging, BC reduces execution time over the next best collector by up to a factor of 3 and reduces maximum pause times by over an order of magnitude versus GenMS.

This thesis begins by asking how well garbage collection performs currently and which areas remain for it to improve. We develop oracular memory management to help answer this question by enabling us to quantify garbage collection's performance. Our results show that while garbage collection can perform competitively, this good performance assumes the heap fits in available memory. Our attention thus focused onto this brittleness, we design and implement the bookmarking collector. While performing competitively when not paging, our new collector's paging performance helps make garbage collection's performance must more robust.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Automatic memory management, or garbage collection, provides significant software engineering benefits. Automatically reclaiming unneeded heap data prevents memory leaks from unreachable objects, “dangling pointers” (when a programmer accesses previously freed memory), and security violations [90, 110]. Garbage collection also improves software modularity by eliminating object ownership and reclamation problems that arise when memory is passed across module boundaries. Because of these benefits, programmers are increasingly using garbage-collected languages such as Java and C#.

Despite these clear software engineering advantages, garbage collection’s impact on application performance remains controversial. Garbage collection requires that the run-time system determine which memory can be safely reclaimed. This additional work that garbage collection performs to find unreachable (or *garbage*) objects is believed to impose a substantial performance penalty [100, 102, 105, 109]. One study shows that applications spend up to 46% of their execution time in garbage collection [55]. This research, however, examines only the relative amount of time spent collecting the heap and offers no comparison with the overhead imposed by other methods of memory management.

However, other studies show that garbage collection can *improve* program performance by increasing cache program locality [26]. Research also shows copying collection can improve cache and page locality by dynamically reordering objects [68]. Unfortunately, these studies suffer the same methodological flaw as the research documenting garbage collection’s performance penalties: they examine garbage collection’s impact, but do not evaluate if its collection costs are higher than its benefits; neither do they perform any comparisons to other memory managers.

Those past studies that compare garbage collection performance to that of other memory managers, rely upon rough estimates of explicit memory manager performance [26, 68] or analyze only *conservative*, non-moving garbage collection [53, 114]. While these results are instructive, it is unclear how they apply to modern precise, copying and hybrid collectors. These best-of-breed collectors have the highest throughput and smallest memory footprints [26], and are currently the most commonly used.

1.1 Contributions

Understanding how garbage collection currently performs is a crucial step to improving its performance. The results of our analyses highlight areas where garbage collection already outperforms and areas where its performance suffers. By focusing research only on those areas which are the biggest bottlenecks or yield the biggest gains, this thesis improves upon researchers' current approach of using intuition or educated guesses in developing new approaches. Therefore, analyzing garbage collection not only provides useful performance data, but also provides a detailed map of how to improve its performance.

Unfortunately, comparing different memory managers is not simple, especially within garbage-collected languages. These languages lack the `free` calls needed for region-based or explicit memory management. While we could insert calls to `free` by hand, rewriting the code may not place these calls in the same location as would the original programmer. This solution examines our skill in rewriting code as much as it evaluates memory manager performance. We instead simulate different locations where a programmer *could* insert `free` calls. We use object reachability information to insert these calls at the last possible time a programmer could free objects during an execution. Using object lifetime information, we insert calls to free at the earliest moment a programmer could free an object. While these simulations are imperfect (e.g., iterations a loop may only occasionally reclaim an object), these imperfections reflect the program's behavior and are both predictable and repeatable.

While object lifetime and reachability information is usually generated via *program heap traces*, relying upon these traces introduces new problems. The *brute force* method of computing object reachability that had been used is prohibitively expensive, because it must analyze the entire heap each time it determines which objects are no longer reachable. Researchers previously minimized these costs by instead generating traces that are only periodically accurate (*granulated traces*). While granulation makes trace generation tractable, it yields significant inaccuracies.

We develop a new algorithm that computes object reachability in asymptotically optimal time, significantly improving upon the brute force approach. This new algorithm, called *Merlin*, uses a two pass approach. A forward pass records the last time each object was provably reachable. A backward pass then propagates these last known reachable times to compute when each object became unreachable. We show that trace generation with Merlin computes object reachability nearly three orders of magnitude faster than generating traces with brute force. In fact, Merlin computes allocation-accurate reachability information faster than brute force computes reachability accurate to a granularity of only 16KB of allocation. Using Merlin allows researchers using program heap traces to generate accurate results quickly.

Merlin can therefore generate the object reachability information we need to insert `free` calls into the program. But while Merlin's backward pass enables the speedy computation of object reachability, it completes generating reachability times only when a program terminates. We therefore cannot get results from the Merlin algorithm until after the program runs. We therefore require at least two identical runs of each program. A first profiling run generates the program heap trace, while later runs can then use the reachability and lifetime information as a zero-cost oracle specifying when to insert `free` calls.

We use this profile-based approach, which we call *oracular memory management*, to quantify garbage collection performance by comparing it with explicit memory management. We use explicit memory management as a touchstone, because it is often cited as top performing method of managing heap memory [20, 59, 95]. By comparing with the best performing explicit memory managers, we measure if garbage collection speeds up performance or slows performance down.

This thesis evaluates memory manager performance on a range of Java programs using a cycle-level processor simulator. Using a simulator not only ensures we have identical benchmark runs, but also enables collection of a variety of performance metrics. We show that a generational collector with non-copying mature space provides a modest throughput improvement over a similar explicit memory manager, on average, and that garbage collection can outperform the explicit memory manager by up to 10% on certain benchmarks. We also show that this performance comes at a price. Garbage collection achieves this top performance by using a heap that is two-and-a-half or five times larger, on average, than that of an explicit memory manager freeing memory in the most conservative and most aggressive manners, respectively. Runs with explicitly managed heaps perform orders-of-magnitude better compared with garbage collection when limited memory forces the eviction of heap pages to disk, dominating garbage collection at all reasonable memory sizes.

Garbage collection's poor paging performance compared to explicit memory management is due to the tracing phase of collection. While explicit memory managers touch pages only when allocating or freeing objects, garbage collectors periodically collect the heap by tracing through the reachable objects. This tracing can trigger page faults, as existing collectors do not track which pages are in memory, and thus cannot avoid touching objects on evicted pages. This poor paging behavior is exacerbated by the virtual memory manager's keeping pages with which the collector is finished (as these pages have recently been touched) and evicting pages that the collector will soon visit. This combination causes a cascade of page faults, stalling progress and increasing garbage collection times.

This thesis presents a new algorithm, called *bookmarking collection*, that improves garbage collection's paging performance. Bookmarking collection (BC) works with the virtual memory manager in evicting pages. The virtual memory manager initiates communication by notifying a process whenever it schedules one of its pages for eviction. Upon receiving this notification, the bookmarking collector tries to reduce its memory footprint and prevent the need for an eviction. BC first attempts to provide the virtual memory manager an unused (*discardable*) page. When all pages are in use, the system collects the heap. After completing the collection, BC tries to provide the virtual memory manager with enough newly discardable pages to prevent heap pages from being evicted. If there are still not enough empty pages, BC processes the pages that will be evicted and records summaries of the references to other pages. With these remembered references, the collector can preserve all objects reachable through references evicted pages and will not fault the page in for future collections. This entire process enables the bookmarking collector to avoid page faults during garbage collection and potentially eliminate application page faults.

We show that the bookmarking collector performs well in a variety of environments. We first demonstrate that BC matches or exceeds the performance of the best performing collector we test when there is no memory pressure. When paging, bookmarking collection reduces execution time over the next best collector by up to a factor of three and reduces the maximum pause times versus GenMS by more than an order of magnitude. This collector dramatically improves overall application usability and substantially improves garbage collection utility.

1.2 Summary of Contributions

This thesis presents a method by which we can generate exact object reachability information and then show how we can use this information to compare garbage collection's performance with that of explicit memory management. We use this system to provide the first apples-to-apples comparison within a garbage-collected language of which we are aware. We find that while garbage collection performance can match that of explicit memory management, but also that this good performance requires the system be able to hold the larger heap without paging. Using these results, we focus our attention on the area where garbage collection suffers most: its paging performance. We then develop and implement a garbage collector that performs well when not paging and cooperates with the virtual memory manager to continue providing good throughput and pause times even when increased memory pressure prevents the entire heap from fitting in memory.

This thesis starts by answering questions about how well garbage collection performs currently, and what opportunities remain for its performance to be improved. Absent head-to-head performance comparisons, developers used incorrect, but widely-held, assumptions of garbage collection's poor performance to reject using this method of memory management. This work presents methods by which we can quantify the performance of different memory managers and compare where each performs well and where each performs poorly. This work shows that garbage collection can already perform well, but that this good performance is brittle and relies upon certain assumptions about the state of the system. We develop a garbage collector that cooperates with the system to provide good performance in a variety of states. More importantly, we feel this thesis provides a methodology by which researchers can examine memory manager performance and focus their attention on those areas where the memory manager quantifiably suffers most.

CHAPTER 2

RELATED WORK

We now discuss the prior research on which this thesis builds. While there is a large body of garbage collection research [72], we focus on specific areas which are most relevant: methods of computing object reachability, analysis of memory manager performance, and efforts to improve garbage collection paging behavior.

2.1 Object Reachability

While there is little previous research into methods of computing exact object reachability, previous work has examined alternative methods of approximating the object reachability information included in a heap trace. There have also been other algorithms that collect the heap in a manner similar to how the Merlin algorithm computes object lifetimes. We discuss these now.

2.1.1 Reachability Approximation

To cope with the cost of computing object reachability, researchers have investigated methods of approximating the reachability of objects. These approximations model the object allocation behavior and patterns of when objects become unreachable found in actual programs. One paper describes mathematical functions that model object reachability characteristics based upon actual reachability information computed from 58 Smalltalk and Java programs [99]. Other research compares several different methods of approximating the object allocation and reachability behavior of actual programs [115]. Neither study attempts to generate actual object reachability information; rather, these studies describe alternative methods of generating the information needed for memory management simulations.

2.1.2 Reference Counting

While the Merlin algorithm does not perform reference counting (RC), issues that arise from Merlin's timestamping are similar to the issues that arise from counting references. As a result of these similarities, we describe the RC collectors most closely related to the Merlin algorithm here.

Reference counting associates a count of incoming references with each object; when the count is 0, it frees the object [46]. To improve efficiency, *deferred* reference counters do not count the numerous updates to stack variables and registers [54], but like other algorithms, enumerates the stacks and registers to compute correct counts only when it collects the heap. Since objects in a cycle never have a reference count of 0 [107], modern implementations add periodic tracing collection or perform *trial deletion* to reclaim cycles of unreachable objects [17, 106]. Trial deletion places objects that lose a pointer but whose count did not reach 0 into a "candidate set". It then recursively performs trial deletions on the objects in this set and those objects reachable from them. If, by reducing this count for one of these objects, all of the reference counts go to zero, the objects form a dead cycle and can be reclaimed.

Unlike RC, the Merlin algorithm is not a garbage collector, but merely computes object reachability. While there are similarities between Merlin and RC (deferring reference counts to limit processing is similar to Merlin's timestamping), Merlin relies upon an underlying collector to actually reclaim objects whereas RC performs this reclamation itself.

2.2 Analyzing Memory Manager Performance

We first discuss past work analyzing the space and time overheads that different memory managers impose. We then examine work that uses models to analyze memory manager performance. Finally, we discuss work that uses conservative garbage collection to compare the performance of garbage collection and explicit memory management. We now discuss past work that examined program overheads as a means of exploring how well memory managers perform.

2.2.1 Garbage Collection Overheads

Several studies have attempted to measure what fraction of program execution time is consumed by garbage collection. Ungar [103] and Steele [98] observe that garbage collection overheads in LISP account for around 30% of application running time. Diwan et al. use trace-driven simulations of six SML/NJ benchmarks to conclude that generational garbage collection accounts for 19% to 46% of application execution time [55]. Blackburn et al. use several whole-heap and generational collectors to measure the space-time trade off when selecting garbage collection [26]. Unfortunately, none of these studies measure the running time and space overhead garbage collection has on the program (e.g., delays resulting from the order objects are laid out in the heap, or due to a collection polluting the cache). It is therefore difficult to ascertain how this overhead relates to those imposed by other memory managers.

2.2.2 Explicit Memory Management Overheads

Researchers have also examined the cost of explicit memory managers, starting with Knuth's simulation-based study [75, p. 435–452]. Korn and Vo find that, of the explicit memory management algorithms they studied, buddy allocation was the fastest and most space-efficient [76]. Johnstone and Wilson measure the space overhead of several conventional explicit memory managers and find that they wasted little memory beyond that induced by object headers and alignment restrictions (they exhibited nearly zero *fragmentation*) [70]. They also conclude that the Lea allocator performs best when considering combined space and time overheads. Berger et al. measure the execution time and space consumption for a range of benchmarks using the actual Lea and Kingsley allocators and their counterparts written in the Heap Layers framework [22]. They find that programs using general purpose memory managers can spend up to 13% of program running time in memory operations. Because these studies measure different programs and cannot account for changes to metrics such as cache locality, we cannot compare these memory manager overhead costs to those from garbage collection directly, however. While explicit memory management has a lower apparent cost, it is unclear if this would translate into improved execution times.

2.2.3 Comparisons Using Explicit Memory Manager Performance Estimates

Comparing different memory managers running identical benchmarks is difficult; one way that past researchers avoided this problem is to estimate explicit memory management performance. Huang et al. [68] use the mutator time from runs of a full heap MarkSweep collector as an estimate of explicit memory manager performance. They show that a copying collector using their on-line object reordering optimization is slightly faster than their estimate of explicit memory management on a majority of benchmarks and provides substantial speedups on most other benchmarks. On only one benchmark does garbage collection perform significantly worse. Blackburn et al. [26] use an identical estimation method to hypothesize that the top performing generational collector at any heap size provides a performance improvement for _202_jess. Unlike this thesis, however, these estimates cannot include the cost of explicit memory management. More importantly, this estimate ignores the improved locality resulting from the immediate reuse of freed objects [60] and the high proportion of very short-lived objects allocated by Java programs [69]. While we see similar results to these past experiments, it is unclear how accurate this estimate of explicit memory management actually is.

2.2.4 Comparisons Using Program Heap Models

Researchers have also developed mathematical models by which they could evaluate and compare different memory managers' performance on similar benchmarks. Zorn and

Grunwald develop and evaluate several different mathematical methods of approximating the object lifetime characteristics of actual programs [115]. Similarly, Appel develops a mathematical model in which he can derive the relative cost for allocating objects into the heap versus allocating them on the stack [13]. Using these models, he shows that in certain environments garbage collection should outperform explicit memory management. Unlike this thesis, these models could not match actual program behavior nor could they be used to compare cache performance. Instead, these works attempted to roughly emulate program behavior in such a way as to then enable them to then compare memory managers.

2.2.5 Comparisons With Conservative Garbage Collection

Past work comparing garbage collection with explicit memory management have understandably taken place using conservative, non-relocating garbage collectors for C and C++. In his thesis, Detlefs compares the performance of garbage collection and explicit memory management for three C++ programs [53, p.47]. He finds that while garbage collection usually results in poorer performance (adding 2%-28% to the total program runtime), the garbage-collected version of cfront performs 10% faster than a version that exclusively uses a general-purpose allocator. However, the garbage-collected version still runs 16% slower than the original cfront (which uses custom memory allocators).

Zorn also performs a direct empirical comparison of conservative garbage collection and explicit memory management using a suite of C programs [114]. He finds that the version of the Boehm-Demers-Weiser conservative garbage collector [36] he tests performs faster than explicit memory allocation on occasion. He also finds, however, that the BDW collector normally consumes more memory than the explicit memory managers, in one case needing 228% more memory to hold the heap.

Unlike these prior studies, this thesis focuses on the performance of programs written for a garbage-collected language. We can therefore examine the impact of the precise, copying garbage collectors that usually provide higher throughputs and better performance [26].

2.3 Improving Garbage Collection Paging Behavior

We now discuss past work examining garbage collection's paging performance or how garbage collection interacts with virtual memory. We do this by characterizing and discussing algorithms by their level of involvement with virtual memory managers. We first describe *VM-sensitive* garbage collectors, algorithms whose design indirectly limits paging. We then consider *VM-cooperative* garbage collectors, which interact with the virtual memory manager or leverage virtual memory to improve their performance. We know of no other algorithm that actively cooperates with the virtual memory manager to eliminate page faults during garbage collection.

2.3.1 VM-Sensitive Garbage Collection

Soon after the development of virtual memory in the sixties, researchers saw the need to improve garbage collectors' virtual memory behavior. Section 3 of Cohen's survey on garbage collection describes these early efforts [45], and we discuss the most relevant work here.

Initial work focused on using compaction to reduce an application's working set [52]. Bobrow and Murphy developed an algorithm that improves virtual memory performance and data locality by allocating and linearizing LISP lists onto individual pages [33, 34]. Baker presented a more general approach, based on the *semispace* copying collector proposed by both the team of Fenichel and Yochelson and by Cheney [18, 40, 58]. Semispace collection compacts the heap by copying the survivors of a collection into a separate memory region. While these algorithms reduce the application's working set, they cannot avoid paging during garbage collection. In particular, semispace collection's looping behavior can trigger the worst-case behavior for the LRU-based page replacement policies used in modern operating systems. The garbage collection algorithm we propose in this thesis not only reduces the application's working set size, but also does so without relying solely on heap compaction. Unlike these previous algorithms, our proposed collector can prevent touching non-resident pages.

Later work improved garbage collection's paging performance by collecting separate regions of memory independently. Bishop presented the first partitioned garbage collector of which we are aware [24]. His collector divides the heap into many different regions, each of which consists of two semispaces. In Bishop's collector, hardware-supported write barriers trap the creation of inter-region pointers and enable the maintenance of *remembered sets* (lists of references into and out of a region). The system limited the lists' size by automatically copying and merging areas that contain a sufficient number of entries. By collecting each area separately, Bishop sought to reduce page faults during garbage collection. While our proposed bookmarking collector, like Bishop's system, organizes the heap into independent regions, our collector manages these regions quite differently. Notably, the system presented in this thesis does not impose any space or execution time overheads except when heap pages are evicted. Once heap pages are written to disk, the collector works with the virtual memory manager to avoid visiting evicted regions.

Ephemeral garbage collection was another approach to improve paging behavior of garbage collection [83]. Ephemeral collection uses hardware-supported write barriers to scan pages when they are evicted and record the "levels" to which the page contains pointers. With this knowledge, the collector need only scan pages containing pointers to the levels being collected. Lacking knowledge of the targets of these pointers, however, ephemeral garbage collectors must reload evicted pages containing references into the level being collected and so cannot avoid page faults during garbage collection.

Generational garbage collection achieves the same result as ephemeral collection without hardware support [14, 79, 86, 103]. Generational collectors reclaim short-lived objects without performing whole-heap garbage collections. These collectors cannot avoid page faults, however, especially when performing whole-heap garbage collections.

A number of researchers focus on improving reference locality by modifying object layouts and groupings using the garbage collector [8, 29, 44, 51, 68, 77, 99, 101, 112]. Other work reduces the large amount of memory typically used by garbage collectors [30, 88, 89]. These studies demonstrate some reduction in the total number of page faults, but do not address the paging problems caused by garbage collection itself, which we identify as the primary culprit. These approaches are orthogonal and complementary to the work presented in this thesis.

Kim and Hsu use the SpecJVM98 suite of benchmarks to examine metrics including garbage collection's memory usage [74]. They found paging causes garbage collection performance to suffer significantly, but that there existed an optimal heap size for each benchmark that minimized both the costs of garbage collection and paging. These optimal heap sizes rely upon applications maintaining a fixed size of reachable data and being run on a dedicated machine; it is unclear how to apply these results to the majority of situations where the assumptions are not met.

2.3.2 VM-Cooperative Garbage Collection

Several garbage collection algorithms reduce paging by communicating with the virtual memory manager. The closest work to that presented in this thesis is that of Cooper, Nettles, and Subramanian [48]. That research uses external pagers in the Mach operating system [81] to allow the garbage collector to direct the removal of *discardable* (empty) pages from main memory and thereby limit the need to evict pages. This system imposes a constant overhead as it is unable to limit the discarding of pages to only those times when memory pressure is high. Additionally, this system does not prevent the garbage collector from visiting evicted pages during collection.

Alonso and Appel present a collector that may shrink the heap after each garbage collection depending upon the current level of available memory. The system uses an "advisor" that calls `vmstat` to obtain the current amount of physical memory available [10]. Their system cannot grow the heap nor can it eliminate paging when the reachable data exceeds available memory. Similarly, Brecht et al. present a collector that adapts Alonso and Appel's work by developing a mechanism of growing the heap to limit the amount of time spent in garbage collection [37].

Several systems adjust their heap size depending on the current environment, primarily to reduce paging costs. MMTk [27] and BEA JRockit [4] adjust their heap size from a predefined set of ratios that use the percentage of the heap used by reachable data or garbage collection pause times. HotSpot [1] has the ability to adjust the heap size based upon pause times, throughput, and footprint limits. Novell Netware Server 6 [3] uses the current level of memory pressure to adjust how often it should collect and compact the heap. All of these systems rely on pre-defined parameters or command line arguments, which makes adaptation slow and inaccurate, and does not eliminate garbage collection-induced paging. Yang et al. report on an automatic heap sizing algorithm that uses information from a simulated virtual memory manager and a model of GC behavior to choose a good heap size [113]. Like these systems, BC can shrink its heap (by giving up discardable pages), but it does not need to wait until a full heap garbage collection to do so. BC also eliminates paging even when the size of live data exceeds available physical memory.

CHAPTER 3

COMPUTING OBJECT REACHABILITY

languages lack the explicit calls to reclaim memory needed by our memory management touchstone, explicit memory management, we could insert these calls whenever the garbage collector could remove the object from the heap. That these *object unreachable* times are already captured in program heap traces would seem to make this process more attractive.

Unfortunately, the traditional *brute force* method of computing these object unreachable times can take over 3 months to generate the trace of for a 60-second program. To limit the trace generation time, researchers have resorted to computing object unreachable times periodically [31, 64, 91, 92, 93]. While improving the time needed to compute object reachability times, we have shown that this can yield inaccurate results [63].

In this chapter, we present a new, faster method of computing these times, which we call *Merlin*. Unlike the *brute force* method, which must analyze the entire heap any time it needs to determine if an object is unreachable, the Merlin algorithm analyzes each heap object only once. Merlin therefore reduces the time needed to compute object reachability information from months to hours.

Arthurian legend says Merlin began life as an old man. The wizard then lived backwards in time. Merlin's great wisdom arose from his making decisions having already experienced the outcome. Like its namesake, the Merlin algorithm starts its processing at program termination and works in reverse chronological order. Working backwards through time, the algorithm can compute object reachability quickly, because the first time it encounters an object is when the object was last reachable.

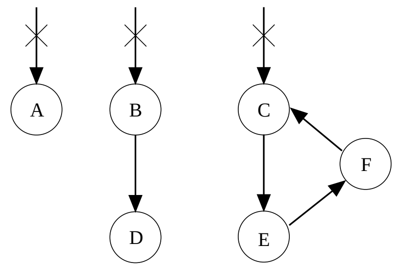


Figure 3.1. Objects A and B transition from reachable to unreachable when their last incoming references are removed. Object C becomes unreachable when it loses an incoming reference, despite having another incoming reference. Objects D, E, and F become unreachable when the referring objects lose an incoming reference and become unreachable.

This chapter first presents the two phases of the Merlin algorithm and how they compute object reachability. We next discuss algorithmic and implementation issues and on- and off-line configurations. We also discuss methods of computing approximate reachability information (when the data are accurate only periodically) and present a detailed performance analysis of an on-line implementation of Merlin.

3.1 Objects Becoming Unreachable

Figure 3.1 illustrates different methods by which objects transition from reachable to unreachable. In this figure, objects A and B become unreachable when they lose their final incoming references. Object C also becomes unreachable when it loses an incoming reference, despite continuing to have more incoming references. Objects D and E, however, transition from reachable to unreachable when their referring objects (B and C, respectively) lose an incoming reference. Object C becoming unreachable also causes object F to transition to unreachable. It is important to note that, in each of these cases, only the object losing a reference and objects in the transitive closure set of their references become unreachable.

While Merlin could process objects losing incoming references (and all those in the transitive closure of its reachability) each time they lose an incoming reference, the algo-

```
void PointerStoreInstrumentation(ADDRESS source, ADDRESS newTarget)
   ADDRESS oldTarget = getMemoryWord(source);
   if (oldTarget ≠ null)
      oldTarget.timeStamp = currentTime;
```

Figure 3.2. Pseudo-code to process pointer stores during Merlin's forward phase

rithm only inserts into the trace the *last time* an object is reachable. Merlin computes these last reachable times efficiently using a two phase approach.

3.2 Forward Phase

Merlin's processing begins with the forward phase. During this phase, Merlin maintains a *timestamp* for each object. The timestamp records the latest time the object may have been last reachable. Whenever a pointer store removes an incoming reference to an object, Merlin replaces the object's timestamp with the current time. This timestamp therefore records the last reachable time for objects that become unreachable when they lose an incoming reference, e.g., objects A, B, and C in Figure 3.1.

3.2.1 Instrumented Pointer Stores

Most pointer stores can be instrumented by a write barrier. When a reference is updated, Merlin timestamps the object losing an incoming reference (the old target of the pointer) with the current time. Since time increases monotonically, writing the current time over the previous timestamp guarantees that Merlin records the final time the object loses an incoming reference. The code in Figure 3.2 illustrates the processing Merlin performs at each pointer store.

3.2.2 Uninstrumented Pointer Stores

Because root references (especially those in registers or thread stacks) are updated frequently, instrumenting root references would be prohibitively expensive. Merlin instead

```
void ProcessRootPointer(ADDRESS rootAddr)
  ADDRESS rootTarget = getMemoryWord(rootAddr);
  if (rootTarget ≠ null)
    rootTarget.timeStamp = currentTime;
```

Figure 3.3. Pseudo-code to process root pointers during Merlin's forward phase

scans the program roots at the start of each granule of time. During this root scan, Merlin stamps the targets of the root references with the current time. While root-referenced, objects are always stamped with the current time; if an object is reachable until losing an incoming root reference, the object's timestamp contains the last time the object was reachable. Figure 3.3 shows the Merlin pseudo-code that updates the targets of root pointers.

3.3 Forward Phase Limitations

Because the Merlin algorithm is concerned with *when* an object was last reachable and cannot always determine *how* the object became unreachable, it is difficult to find a single method that computes every object's last reachable time. The timestamps from the forward phase (e.g., generated by the pseudo-code in Figures 3.2 and 3.3) compute the last reachable time for those objects that are last reachable when they lose an incoming reference (e.g., objects A, B, and C in Figure 3.1). However, the timestamps recorded for objects D, E, and F may not be computed when these objects become unreachable. These latter three objects become unreachable only when the object referring to them transitions to unreachable.

To handle reference chains during the forward phase, updating the last reachable times for an object requires recomputing the last reachable times of the objects to which it points. In Figure 3.1, this would require updating object D's timestamp whenever Merlin updates object B's timestamp. Similarly, Merlin would need to update the timestamps for objects C, E, and F together. Since each object could transitively refer to every heap object, updating all of these timestamps during a forward phase would make this process too expensive to be useful.

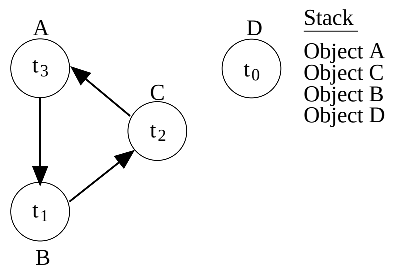
3.4 Backward Phase

To avoid this cost, Merlin propagates timestamps along references during a second phase. This second phase processes only provably unreachable objects, since they are the only objects that Merlin knows cannot lose any further incoming references. Processing these provably unreachable objects ensures that Merlin need only compute each object's transitive closure set once.

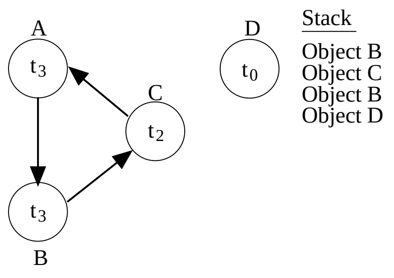
Computing full transitive closure sets is still a time consuming process, however. But Merlin need not compute the full transitive closure. The algorithm must compute only the *last* time each object was reachable. Therefore, only the latest timestamp reaching an object — the last time the object was reachable from the program roots — is needed. To speed processing, Merlin instead finds only this latest reaching timestamp for each object.

To compute only the latest last reachable time for each object, Merlin's second phase runs backward in time (e.g., begins determining the transitive closure set for the object with the latest timestamp back to the object with the earliest timestamp). The algorithm sets up this backward processing by first sorting all the unreachable objects by their timestamp. Merlin then initializes a priority queue such that the object with the latest timestamp will always be processed first.

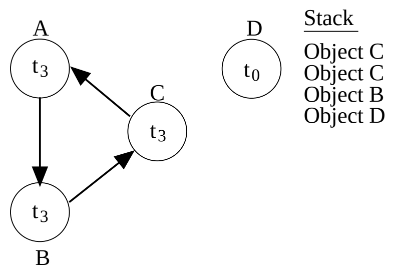
We now describe how this backward phase works using the illustrations in Figure 3.4 as an example. Figure 3.4(a) shows the processing state after Merlin sorts unreachable objects and initializes its worklist. Merlin then processes object A (the object with the latest timestamp). In the backward phase, Merlin propagates timestamps by scanning objects for references to other (target) objects. When a target object's timestamp contains an earlier time, Merlin replaces the target object's timestamp with that of the source object. The target object is also moved to the front of the priority queue, so the timestamp can continue to be propagated. Figure 3.4(b) shows what happens after Merlin processes object A. Because it had a later timestamp, Merlin copied object A's timestamp into the target object (e.g., object B) and added the target object onto the top of the worklist. Figure 3.4(c) shows how Merlin



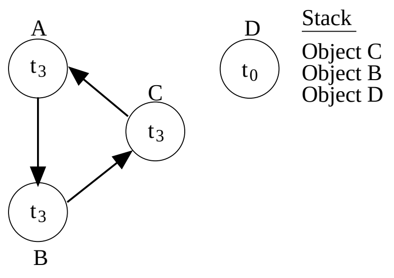
(a) Before Processing Object A



(b) Before Processing Object B



(c) Before Processing Object C



(d) After Processing Object C

Figure 3.4. Computing object death times, where $t_i < t_{i+1}$. Since Object D has no incoming references, Merlin's computation will not change its timestamp. Because Object A was last reachable at its timestamp, the algorithm will not update its last reachable time when processing the object's incoming reference. In (a), Object A is processed finding the pointer to Object B. Object B's timestamp is earlier, so Object B is added to the stack and last reachable time set. We process Object B and find the pointer to Object C in (b). Object C has an earlier timestamp, so it is added to the stack and timestamp updated. In (c), we process Object C. Object A is pointed to, but it does not have an earlier timestamp and is not added to the stack. In (d), the cycle has finished being processed. The remaining objects in the stack will be examined, but no further processing is needed.

has continued propagating this time by processing object B. When processing object C (in Figure 3.4(c)), Merlin ignores the target object, since the target's timestamp (object A's t_3) is equal to the timestamp being propagated. Merlin can safely ignore these objects because they will already propagate the timestamp. While not shown in this example, Merlin also ignores target objects with later timestamps, since it already computed that those targets were last reachable after the time being propagated. Figure 3.5 shows the pseudo-code for the backward phase.

Merlin's backward phase works from the latest timestamp to the earliest; after a first analysis, repeat visits to an object are propagating earlier last reachable times and can be

```
void ComputeObjectDeathTimes()
  Time lastTime = ∞
  sort unreachable objects from earliest to latest timestamp
  push each unreachable object onto a stack from earliest
    timestamp to the latest
  while (!stack.empty())
    // pop obj w/ next earlier timestamp
    Object obj = stack.pop();
    Time objTime = obj.timeStamp;
    // don't reprocess relabelled objects
    if (objTime <= lastTime)
      lastTime = objTime;
      for each (field in obj)
        if (isPointer(field) && obj.field ≠ null)
          Object target = getMemoryWord(obj.field);
          Time targetTime = target.timeStamp;
          if (isUnreachable(target) && targetTime < lastTime)
            target.timeStamp = lastTime;
            stack.push(target);
```

Figure 3.5. Pseudo-code for Merlin's backward phase. This phase uses the timestamps generated during the forward phase to compute the time each object became unreachable.

ignored. In Appendix A, we present a proof that this method of finding last reachable times is asymptotically optimal requiring only $\Theta(n \log n)$ time, where n is the number of objects available for processing, and is limited only by the sorting of the objects.

3.5 Algorithmic Details

This section expands on the key insights and implementation issues for Merlin. It first discusses algorithmic requirements and then presents several assumptions on which the Merlin algorithm is based.

3.5.1 Algorithmic Requirements

The in-order brute-force method computes object reachability times as the program executes. Merlin instead computes these times during its backward phase and so must introduce *timekeeping* to correlate when each object is last reachable with events in the program execution. Time is related to the granularity of the reachability information and must advance whenever Merlin could determine an object was last reachable.[1]

3.5.2 Merlin Assumptions

This algorithm relies upon several assumptions. First, Merlin assumes that any object to which a reachable object refers is also treated as reachable. Most garbage collection algorithms also make this assumption. Second, we assume after an object becomes unreachable its incoming and outgoing references do not change. Both of these preconditions are important for our timestamp propagation, and garbage-collected languages such as Java, C#, and Smalltalk satisfy them.

Merlin also assumes that the order in which it outputs object reachability information is unimportant. Because last reachable times are computed only during its backward phase,

[1]For many collectors, time need only advance at object allocations. To simulate collectors that can reclaim objects more frequently, e.g., reference counting collectors, time would advance at each location where the collector could scan the program roots and begin a collection.

these times are not generated in any particular order. If order is important, the algorithm reorders the records after program termination in a post-processing step. While this post-processing adds another step, it adds very little overhead.

3.6 Implementing Merlin

We now shift the focus of this discussion from the algorithm to issues arising during implementation. We present on-line and off-line implementations separately, because they face very different challenges.

3.6.1 On-Line Implementations

Since Merlin periodically scans for root pointers and must process object reference updates, it can be much easier to perform the entire analysis on-line. We now discuss the technical hurdles that this implementation raises.

The pseudo-code in Figure 3.2 shows Merlin analyzing references during a pointer store operation. On-line implementation can insert this processing into the write barrier already defined in most garbage-collected systems. While this assumes that the write barrier is executed prior to overwriting the reference, many reference counting collectors also make this assumption about the write barrier and it usually holds.

Merlin's performance depends on processing only unreachable objects during the backward phase, but the algorithm cannot differentiate reachable and unreachable objects. Most on-line implementations can instead rely upon the host system's garbage collector to perform this analysis for them. Immediately after a collection, but before memory is cleared, all the heap objects and the garbage collector's reachability analysis are accessible. While Merlin's piggybacking onto the garbage collector can save a lot of duplicative work, many systems may not reclaim all unreachable objects (e.g., objects last reachable just before program termination). In these cases, on-line implementations of Merlin stamp the target of root references with the current time and then propagate the last reachable times throughout the entire heap. Any object whose timestamp matches the current time is still reachable from the program roots. All other objects are unreachable and the implementation can record their last reachable times. This combination limits the overhead of on-line Merlin implementations, while allowing them to work with any garbage collector.

3.6.2 Using Merlin Off-line

Merlin can also compute object lifetime information from an otherwise complete program heap trace [62, 65]. Off-line implementations of Merlin process the allocation, reference update, and root enumeration records included within a trace to perform the forward pass. Because they cannot access the objects themselves, off-line implementations must also build and maintain an exact copy of the heap during this processing. Once it has finished processing the records in the trace, the implementation can use its copy of the heap to perform the backward processing pass. Especially when seeking reachability information for a subset of the object (e.g., objects allocated during a particular phase or in the second of two runs of a benchmark) or modifying the host system is difficult, off-line implementations of Merlin can be very useful.

3.7 Granularity of Object Reachability

We now consider how Merlin determines the granularity of the object reachability information it computes. Some research needs only coarse-grained object reachability, while other research could consider generating very detailed information (e.g., computing object reachability at the granularity of method boundaries for use in a dynamic "escape-analysis" optimization).

Merlin's timestamping of root-referenced objects, discussed in Section 3.2.2, must occur at the start of each new time granule. While this processing occurs at object allocations when computing allocation-accurate reachability information, Merlin could perform these actions at method exits to generate reachability information for an escape-analysis optimization.[2] When computing different object reachability information at different granule sizes, the only difference is when to trigger the root scan. No matter how fine- or coarse-grained the reachability information desired, the algorithm behaves identically at pointer stores and when processing root-referenced objects.

3.8 Evaluation of Merlin

We made several optimizations to the Merlin algorithm to limit the time it spends scanning the program roots. Our first optimization inserted a write barrier into the code storing references in fields marked static. This instrumentation allows our Merlin implementation to treat static references as it does heap references and not as roots. Java allows methods to access only their own stack frame, so within a single method scanning lower stack frames will always enumerate the same references. We implemented a *stack barrier* that is called when frames are popped off the stack, enabling Merlin to scan less of the stack [42]. Because the static write barrier and stack barrier will only improve Merlin's performance, and not that of brute-force, our implementation of the brute-force method does not include them.

3.8.1 Experimental Methodology

We implemented a trace generator in Jikes RVM version 2.0.3 that uses either the brute-force method or the Merlin algorithm to compute object reachability. Whenever possible, the two object reachability computation methods use identical code, including having the Merlin-based trace generator use the same semispace collector as the brute-force-based implementation. We time each implementation of the trace generator on a Macintosh Power Mac G4, with two 533 MHz processors, 32KB on-chip L1 data and instruction caches, 256KB unified L2 cache, 1MB L3 off-chip cache and 384MB of memory, running PPC

[2]By perfoming the root scans less often, Merlin could similarly generate coarser-grained reachability information.

Linux 2.4.3. These experiments were run using only one processor and the system in single-user mode with the network card disabled.

We examined performance across a range of granularities and benchmarks. We selected granularities reflecting the range that has been used in experiments. Our benchmarks are a subset of those available in the SPECjvm98 [97] and jOlden [39] suites. We chose these benchmarks to explore a diverse set of program behavior and allocation sizes. We ran some benchmarks for only the initial 4 or 8 megabytes of allocation because of the time required for brute-force trace generation (despite this limitation, runs still needed as much as 34 hours to complete).

3.8.2 Computing Allocation-Accurate Results

Figure 3.6 shows the speedup when using Merlin to compute allocation-accurate object reachability relative to the brute-force method computing lifetimes at various granularities. As the graph shows, the allocation-accurate Merlin runs execute faster than brute-force needs when computing reachability as coarse as 16KB to 1MB. When comparing only allocation-accurate runs, Merlin always improves overall performance by several orders of magnitude. For Health(5 256), the Merlin-based implementation offers a speedup factor of 816.

While Merlin can reduce the time needed to generate a trace, Figure 3.6 shows that this speedup decreases as granularity increases. The brute-force method acts only to determine which objects are now unreachable. Within a benchmark its running time is largely a factor of the number of garbage collections it triggers — the granularity of the object reachability being computed. While the Merlin-based implementation performs fewer actual collections computing allocation-accurate lifetimes than brute-force at even large granularities, Merlin's performance also depends on the cost of updating timestamps. As the results from the largest granularities in Figure 3.6 show, the additional actions at each pointer store and allocation can exceed the cost of a few extra collections.

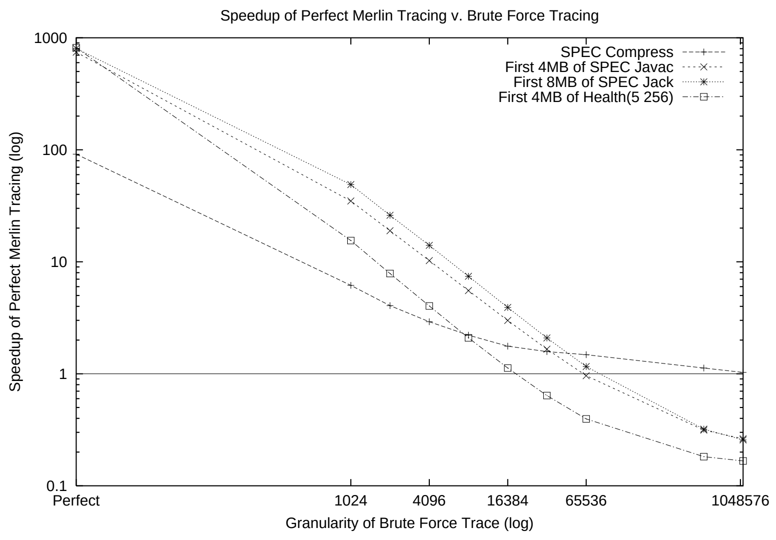


Figure 3.6. Speedup of using Merlin-based allocation accurate trace generation. This graph uses a log-log scale.

3.8.3 Computing Granulated Results

While the previous results show that Merlin is quicker when computing object reachability at a very fine-grain, it does not show if Merlin is faster when only *large granularities* are needed. While we have shown that inaccurate object reachability can cause statistically significant distortions for simulation results [63], Hirzel et al. show that using larger granularities of reachability information does not affect their analyses substantially [64]. Even with the improvement that Merlin provides in computing object reachability, trace generation runs 70–300 times slower than when the program runs without tracing. Figure 3.7 shows that using granulated results can improve performance by over an order of magnitude and could be useful when this approximation will not distort results. Given a program with a maximum live size of 10MB, for example, a 10KB granularity overestimates the maximum live size by at most 0.1%.

Figure 3.7 shows that while introducing some granularity improves performance, there is little gain in computing reachability at a granularity above 4096 bytes. However, Figure 3.6 shows that the performance of the brute-force based trace generator continues im-

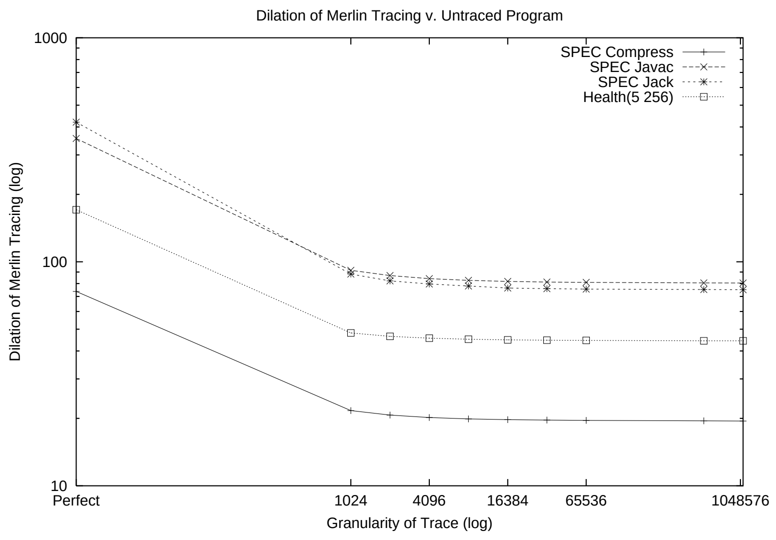


Figure 3.7. Slowdown from Merlin-based trace generation relative to running the program normally. Merlin imposes a substantial slowdown when computing allocation-accurate object reachability. If approximate object reachability is desired, generating a slightly granulated trace requires $\frac{1}{5}$ the time. This graph uses a log-log scale.

proving even at granularities as large as 1MB. It is therefore unclear whether Merlin or brute-force is better when computing coarsely granulated reachability information.

Figure 3.8 compares the performance of generating granulated traces using brute-force relative to generating a similarly granulated trace using Merlin. As the graph shows, the Merlin-based approach dominates the brute-force-based approach at all tested granularities and benchmarks. Some of Merlin's workload (enumerating and scanning the root pointers) is related to granularity and some work is constant (timestamping objects losing incoming references via instrumented pointer stores). This constant overhead causes the improvement from using Merlin to drop from a speedup factor of 817 for allocation-accurate reachability to a factor of 5 at a granularity of 64KB and finally a factor of 1.14 at 1MB. Even at this very high granularity, however, not needing to perform repeated garbage collections makes Merlin the clear winner. Combining Figures 3.7 and 3.8, we find a persuasive argument for always using Merlin to compute object reachability information.

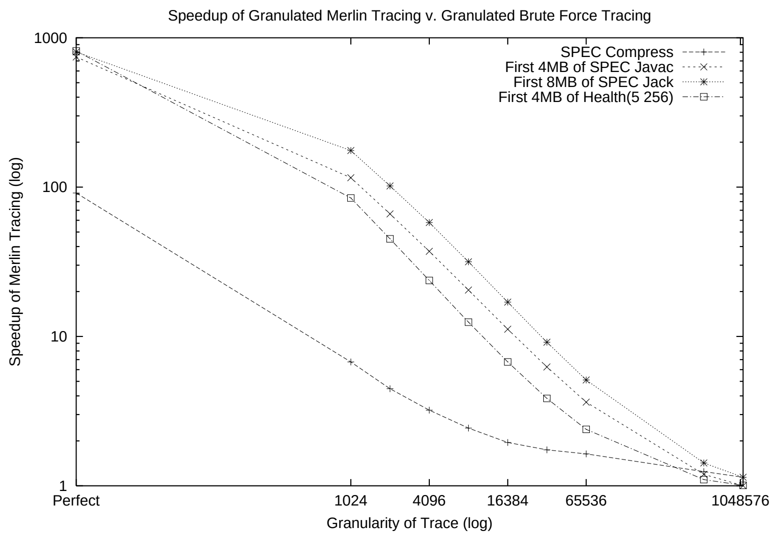


Figure 3.8. Speedup of Merlin-based trace generation relative to brute force-based trace generation at equal levels of granularity. Using Merlin to compute object reachability information is faster in every instance tested. This graph uses a log-log scale.

3.9 Conclusion

In this chapter, we present the Merlin algorithm and discuss how it can be implemented either on-line or off-line. We then compare the time needed to generate traces when using an on-line implementation of Merlin versus the brute force method of computing object reachability times. We show that using the Merlin algorithm can improve the time needed to generate an allocation-accurate trace by a factor of over 800. We also show that the Merlin algorithm helps generate allocation-accurate traces faster than the brute force method can generate traces that are only accurate at every 16KB of allocation. We next discuss how to use the Merlin algorithm to generate traces at both finer and coarser granularities. The Merlin algorithm thus makes computing object reachability times practical.

In this next chapter, we describe how we use Merlin to compare garbage collection's performance with that of the top explicit memory managers. Unlike past comparisons, the accurate object reachability information Merlin computes enables us to perform this comparison using both the best performing garbage collection algorithms and explicit memory managers running unaltered Java applications.

CHAPTER 4

QUANTITATIVE ANALYSIS OF AUTOMATIC AND EXPLICIT MEMORY MANAGEMENT

We now introduce a novel experimental methodology that uses program heap traces to quantify precise garbage collection's performance relative to explicit memory management. Our system treats unaltered Java programs as if they used explicit memory management by using information found in these traces as an *oracle* to direct us where to insert calls to `free`. By executing inside an architecturally-detailed simulator, this "oracular" memory manager eliminates the effects of consulting an oracle, while still including the costs of calling `malloc` and `free`. We evaluate two different oracles: a liveness-based oracle that aggressively frees objects immediately after their last use, and a reachability-based oracle that uses the results of the Merlin algorithm to free objects just after they are last reachable. Because these represent the most aggressive and conservative insertions of `free` calls, we evaluate a range of possibilities.

We then compare explicit memory management with both copying and non-copying garbage collectors across a range of benchmarks using the oracular memory manager. Non-simulated runs of our oracular memory manager lend further validity to our results. Our real and simulated results quantify the time-space trade-off of garbage collection: with a heap size nearly four-and-a-half times larger, the best performing generational collector improves average performance versus the reachability-based explicit memory management by 4%. To match performance, the garbage collector needs a heap size two-and-three-quarters or slightly more than five times that of the explicit memory manager using the most conservative and more aggressive oracles, respectively. In the smallest heap size in which it completes, the garbage collector not only has a heap size 11% or 28% larger

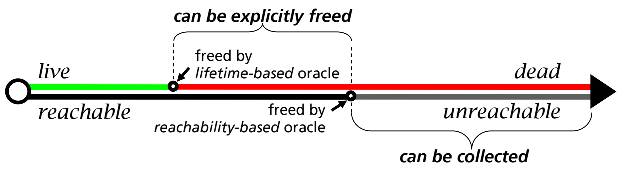


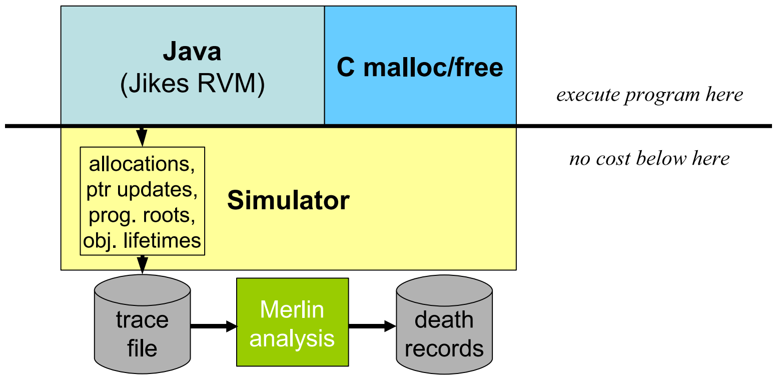
Figure 4.1. Object lifetime. The lifetime-based oracle frees objects after their last use (the earliest safe point), while the reachability-based oracle frees them after they become unreachable (the last possible moment an explicit memory manager could free them).

than the explicit memory manager, but also degrades throughput by an average of 62% to 71%. We also show the difficulty paging causes garbage collection: when paging, garbage collection runs orders-of-magnitude slower than explicit memory management.

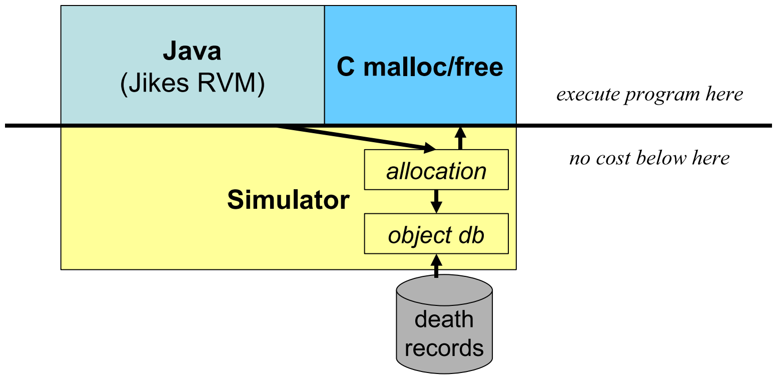
4.1 Oracular Memory Management

Figure 4.2 presents an overview of the oracular memory management framework. As this figure shows, the framework runs each benchmark twice. During the first run, shown in Figure 4.2(a), the framework executes the Java program to calculate object lifetimes and generate a garbage collection trace. The Merlin algorithm then processes the trace to compute object reachability times. We can then use the object reachability times and object lifetimes as the lifetime- and reachability-based oracles (see Figure 4.1). Figure 4.2(b) illustrates how the oracular memory manager uses the oracles. During the second run, the framework allocates objects using `malloc` and invokes `free` when directed by the oracle to reclaim an object. As these figures show, trace and oracle generation occurs outside of the simulated space; the system measures only the costs of allocation and deallocation.

We now describe our oracular memory manager framework in detail. We first discuss our solutions to the challenges of generating the oracles. We then discuss how our framework detects memory allocations and inserts `free` calls without modifying the program state. We address the limitations of our approach in Section 4.2.



(a) Step one: collection and derivation of object lifetime and reachability



(b) Step two: execution with explicit memory management.

Figure 4.2. The oracular memory management framework.

4.1.1 Step One: Data Collection and Processing

For its Java platform, the oracular memory manager uses an extended version of Jikes RVM version 2.3.2, configured to produce PowerPC Linux code [11, 12]. Jikes RVM is a widely-used research platform written almost entirely in Java. A key advantage of Jikes RVM and its accompanying Memory Management Toolkit (MMTk) is that it already includes well-tuned implementations of a number of garbage collection algorithms [27] for ready incorporation into these experiments. The oracular memory manager executes inside Dynamic SimpleScalar (DSS) [67], an extension of the SimpleScalar superscalar architectural simulator [38] that permits the use of dynamically-generated code.

4.1.1.1 Repeatable Runs

The oracular memory manager identifies objects based upon the order in which the objects are allocated. The sequence of allocations must therefore be identical for both the profiling and measurement runs. We now discuss the steps we take in Jikes RVM and the simulator to ensure repeatable runs.

The Jikes RVM just-in-time compilation system provides a considerable source of non-determinism. We enforce predictability of this system in two ways. First, we use the “fast” configuration of Jikes RVM, which optimizes as much of the system as possible and compiles it into a prebuilt virtual machine. Second, we disable Jikes RVM’s “adaptive” system, which uses timer-based sampling at runtime to find and optimize “hot” methods, and instead use the *Replay* compilation methodology [68, 89, 113]. Our implementation of this methodology records the hot methods and to what level each method was compiled for 5 runs. We then generate an advice file that stores the mean result of these 5 runs. Jikes RVM uses this file to optimize the hot methods when they are first compiled. By using a single advice file, we preserve both adaptive-like optimization and deterministic behavior.

Several other changes are required to provide the framework with repeatable runs. Jikes RVM normally switches threads at regular time intervals. We instead employ deterministic thread switching, which switches threads after executing a specified number of methods. We also modify DSS to update the simulated OS time and register clocks deterministically. Rather than use cycle or instruction counts, which change when we add calls to `free`, our modifications advance these clocks a fixed amount every time we make a system call. Together, these changes ensure that all runs are repeatable.

4.1.1.2 Tracing for the Liveness-Based Oracle

The framework generates the liveness-based oracle by calculating object lifetimes during the first (profiling) run. The per-object lifetime information is obtained by having the simulator track where each object is allocated. The simulator then determines the object being used at each memory access and records a new last-use time (in allocation time) for the object. Unfortunately, this approach does not capture object lifetimes perfectly. For example, while an equality test uses both argument objects, Java's equality operation compares addresses only and without examining memory locations. To capture these object uses, we traverse the program roots at each allocation and record root-referenced objects as being in use. This extended definition potentially overestimates object lifetimes, but eliminates the risk of freeing an object too early.[1]

The oracular framework also preserves objects that we feel a programmer could not reasonably free. For instance, while our framework detects the last use of a code block or type information, in a real program, developers would not be able to deallocate these objects. Similarly, our oracular framework does not free objects used to optimize class loading, such as mappings of class members to their Jikes RVM internal representation. The oracular framework also extends the lifetime of objects that enable lazy method compilation or reduce the time spent scanning jar files. The oracular framework prevents reclaiming these

[1]Agesen et al., present a more detailed discussion of the limitations of liveness analysis in Java and propose several alternative solutions [9].

objects (and the objects to which they refer) by having the simulator artificially record a use for each of these objects at the end of the program run.

4.1.1.3 Tracing for the Reachability-Based Oracle

To generate the reachability-based oracle, the framework generates a garbage collection trace during the profiling run. We then process the trace using the off-line implementation of the Merlin lifetime analysis algorithm described in Section 3.6.2.

A key difficulty in generating the garbage collection trace is capturing the trace information while maintaining perfectly repeatable runs. Since Jikes RVM allocates compiled code onto the heap and our tracing runs must be identical to our execution runs, we require the code executing during the first (profiling) run be the same as the code used in future (simulation) runs. Jikes RVM does not, however, generate code that would enable the framework to record all of the information the Merlin algorithm needs to compute object reachability.

The key concept behind our approach obtaining the necessary heap events is to replace normal opcodes with illegal ones. We introduce a new opcode to replace every instruction that stores an intra-heap reference or that calls the allocation code. These events are uniquely identified by the illegal opcodes; because they cannot run on a real processor, the new opcodes do not appear anywhere else in the program. When DSS encounters one of the new opcodes, it outputs an appropriate record into the garbage collection trace. The simulator then executes the illegal opcode just as it would execute the legal variant.

To enable generation of these illegal opcodes, we extend Jikes RVM's compiler intermediate representations. Jikes RVM's IR nodes differentiate between method calls within the VM and calls to the host system. This differentiation exists to minimize the modifications needed to support different OS calling conventions. We further extend this separation to create a third set of nodes representing calls to `malloc`. This extension allows the compiler to treat object allocations like any other function call, but enables us to emit an illegal opcode rather than the usual branch instruction. We also modify the Jikes RVM to emit intra-heap reference stores as illegal instructions. These opcodes allow us to generate the garbage collection trace within the simulator without adding code that distorts instruction cache behavior or makes program behavior less predictable.

4.1.2 Step Two: Simulating Explicit Memory Management

After completing step one, we can run the program using explicit memory management within the oracular framework. During these runs, the simulator consults the oracle before each allocation to determine if any objects should be freed. When freeing an object, the simulator saves the parameter value (the size request for `malloc`) and instead jumps to `free`, but sets the return address to again execute the `malloc` call rather than the following instruction. Execution then continues as normal, with the program freeing the object presented by the oracle. When the simulator returns from the `free` call, it will again be at a call to `malloc`. The simulator again consults the oracle and continues in this cycle until there are no objects left to be reclaimed. Once no further objects remain to be reclaimed, the allocation and subsequent program execution continues as normal. When `malloc` and `free` functions are implemented outside of the VM, they are called using the Jikes RVM foreign function interface (`VM_SysCall`); we discuss the impact of using the foreign function interface in Section 4.2.2.

4.1.3 Validation: Live Oracular Memory Management

In addition to the simulation-based framework described above, we also implemented a "live" version of the oracular memory manager. This live version uses oracles generated as before, but executes on a real machine. Like the simulation-based oracular memory manager, the live runs use unique IR nodes to track intra-heap references and object allocations, but emit legal instructions rather than the illegal instructions discussed above. The live oracular memory manager uses the reachability information stored on disk and an array tracking object locations to fill a special "oracle buffer." The live oracle memory manager fills this oracle buffer with the addresses of the objects the oracle specifies should be reclaimed. To determine the program's running time, we measure the total execution time and then subtract the time spent consulting the oracle and refilling the oracle buffer with the addresses of the objects to free.

Because oracle consultation and buffer usage affect the system state, we compared the cost of running garbage collection as usual to running garbage collection with a *null oracle*. The null oracle consults the oracle and loads the buffers like the real oracular memory manager, but never `free`s any objects. Our results show that even the null oracle causes unacceptably large and erratic distortions. For example, using the GenMS collector with the null oracle to run _228_jack yields an adjusted runtime 12% to 33% greater than an identical configuration running without any oracle. By contrast, GenMS executes _213_javac only 3% slower when using the null oracle. Other collectors also show distortions from the null oracle, but not in any obvious or predictable pattern. We attribute these distortions to pollution of both the L1 and L2 caches induced by processing the oracle buffers.

While the live oracular memory manager is too noisy to be reliable for precise measurements, its results closely mirror the trends of the simulation results. Section 4.4.1.2 compares the trends exhibited by the live and simulation-based oracles and discusses their similarities.

4.2 Discussion

In the preceding sections, we focused on the experimental methodology. In our methodology, we strive to eliminate measurement noise and distortion. Here we discuss some of the key assumptions of our approach and address possible concerns. These include invoking `free` on unreachable and dead objects, the cost of using foreign function calls for memory operations, the effect of multithreaded environments, unmeasured costs of explicit memory management, the role of custom memory allocators, and the effects of memory managers on program structure. While our methodology may appear to hurt explicit memory management (i.e., making garbage collection look better), we argue that the differences are negligible.

4.2.1 Reachability versus Liveness

The oracular memory manager analyzes explicit memory management performance using two very different oracles. The liveness-based oracle deallocates objects aggressively, invoking `free` at the first possible opportunity it can safely do so. The reachability oracle instead frees objects at the last possible moment in the program execution, since calls to `free` require a reachable pointer as a parameter.

As described in Section 4.1.1, the liveness-based oracle preserves some objects beyond their last use. While beyond their last use, all of these objects are root-reachable and so will also be preserved by the reachability-based oracle. However, the liveness-based oracle does reclaim a small number of objects the reachability-based oracle will not. These objects are ones that a knowledgeable programmer, not knowing that the program will soon terminate, would reclaim. The number of additional `free` calls needed to reclaim these objects is quite small. Only for pseudoJBB at 3.8% and _201_compress at 4.4% are more than 0.8% of all objects involved.

Real program behavior likely falls between the two extremes represented by these oracles. We expect few programmers to be capable of reclaiming every object immediately after its last use, and similarly, we do not expect programmers to wait until the last possible moment before freeing an object. These two oracles thus bracket the range of explicit memory management options.

We show in Section 4.4.1.3 that the gap between the two oracles is small. Both oracles provide similar run-time performance, while the liveness-based oracle reduces the reported heap size by at most 20% versus the reachability oracle for all but one benchmark (pseudoJBB sees a heap size reduction of 26%). These results generally coincide with previous studies of both C and Java programs. In a comparison of seven C applications, Hirzel et al. find that using an aggressive interprocedural liveness analysis improves average object lifetime (measured in allocation time) over 2% for only 2 of their benchmarks [66]. In a study including five of the benchmarks we examine here, Shaham et al. measure the average impact of inserting `null` assignments in Java code, simulating nearly-perfect placement of explicit deallocation calls [93]. They report space consumption gains of only 15% for the aggressive reclamation of objects versus a reachability-based approach.

4.2.2 `malloc` Overhead

We examine an allocator that uses code within Jikes RVM to perform explicit memory management. The oracular memory manager invokes allocation and deallocation methods using the normal Jikes RVM method call interface. Because we need explicit method calls to determine when an allocation occurs and, where appropriate, to insert calls to `free` however, the oracular memory manager cannot inline the allocation fast path. While not inlining the allocation fast path may prevent some potential optimizations, we are not aware of any explicitly-managed programming language that implements memory management operations without function call overhead.

We also use allocators implemented in C. With these allocators, the oracular memory manager implements `malloc` and `free` calls using the Jikes RVM `VM_SysCall` foreign function call interface. While not free, these calls cost 11 instructions: six loads and stores, three register-to-register moves, one load-immediate, and one jump. This cost is similar to that of invoking memory operations in C and C++, where `malloc` and `free` are usually functions defined in an external library (e.g., `libc.so`).

4.2.3 Multithreaded versus Single-threaded

In the experiments presented here, we assume a single-processor environment and disable atomic operations both for the Jikes RVM and the Lea allocator. In a multithreaded environment, most thread-safe memory allocators require at least one atomic operation for every call to `malloc` and `free`: either a test-and-set operation for lock-based allocators, or a compare-and-swap operation for non-blocking allocators [82]. These atomic operations can be very costly. On the Pentium 4, for example, the atomic compare-and-swap operation (`CMPXCHG`) requires around 124 cycles. Because garbage collection can amortize the cost of atomic operations by performing batch allocations and deallocations, Boehm observes that it can be much faster than explicit memory allocation [35].

The issue of multithreaded versus single-threaded environments is orthogonal to the comparison of garbage collectors and explicit memory managers, however, because explicit memory allocators can also avoid atomic operations for most memory operations. In particular, since version 3.2, Hoard [19, 21] maintains thread-local freelists and typically uses atomic operations only when flushing or refilling them. Use of these thread-local freelists is cheap, normally using a register reserved for accessing thread-local variables.

4.2.4 Smart Pointers

Explicit memory management can have other performance costs. For example, C++ programs may use smart pointers to help manage object ownership. Smart pointers transparently implement reference counting, and therefore add expense to every pointer update. For example, the performance of the Boost "intrusive pointer" that embeds reference-counting within an existing class is up to twice as slow as the Boehm-Demers-Weiser collector on the gc-bench benchmark [71].

Smart pointers do not appear to be in widespread use, at least in open source applications. We searched for programs using the standard `auto_ptr` class or the Boost library's `shared_ptr` [47] on the open-source website `sourceforge.net` and found only two large programs using them. We attribute this lack of use both to smart pointers' cost (C++ programmers tend to be particularly conscious of expensive operations) and inflexibility. For example, the same smart pointer class cannot manage all objects, because C++ uses different syntax to reclaim scalars (`delete`) and arrays (`delete []`).

Instead, C and C++ programmers generally use one of the following conventions: a function caller either allocates objects that it then passes to its callee, or the callee allocates objects that it returns to its caller (as in `strncpy`). These conventions impose little to no performance overhead in optimized code, but place the burden of preventing memory leaks on the programmer.

Nonetheless, some patterns of memory usage are inherently difficult to manage with `malloc` and `free`. For example, the allocation patterns of parsers make managing individual objects an unacceptably-difficult burden. In these situations, C and C++ programmers often resort to custom memory allocators. Past research examined the performance impact of custom memory allocators [23] and we consider evaluation of these issues beyond the scope of this thesis.

4.2.5 Program Structure

While the programs we examine in this chapter were written for a garbage-collected environment, most were inspired by programs written for explicit memory management. While this may affect how they are written, we unfortunately cannot see how to quantify this effect. We could attempt to measure it by manually (re-)rewriting the benchmark applications to use explicit deallocation, but would somehow have to factor out the impact of individual programmer style.

In fact, it may not always be possible to insert `free` calls statically such that all the objects allocated by a garbage-collected program are reclaimed. In a study of the effectiveness of escape analysis, Blanchet found that escape analysis could identify and stack allocate only 13% of objects on average [32]. Using a different analysis, Guyer et al. found that they could allocate only 21% of objects onto the stack. They further developed a static analysis which could free an average of 32% and a maximum of 80% of allocated objects. Adding three manually-placed calls to `free` increased the percentage of freed objects to 89% on one benchmark, but the researchers found another benchmark for which they were

Benchmark	Total Bytes Alloc	Max. Bytes Reach	Bytes Alloc/Max. Reach
		Benchmark statistics	
		SPECjvm98	
_201_compress	125,334,848	13,682,720	9.16
_202_jess	313,221,144	8,695,360	36.02
_205_raytrace	151,529,148	10,631,656	14.25
_209_db	92,545,592	15,889,492	5.82
_213_javac	261,659,784	16,085,920	16.27
_228_jack	351,633,288	8,873,460	39.63
		DaCapo	
ipsixql	214,494,468	8,996,136	23.84
		SPECjbb2000	
pseudoJBB	277,407,804	32,831,740	8.45

Table 4.1. Memory usage statistics for the benchmark suite used to compare explicit and automatic memory management

never able to reclaim more than 27% of the object statically [60]. While not an issue for the oracles in this thesis, which insert calls to `free` dynamically, this affects non-oracle-based comparisons of explicit memory management and garbage collection.

Despite the apparent difference in program structure that one might expect, we observe that Java programs often assign `null` to references that are no longer in use. In fact, there are many locations in the SPEC suite of benchmarks where programmers have gone back and added `null` assignments with the explicit goal of making objects unreachable. In this sense, programming in a garbage-collected language is at least occasionally analogous to explicit memory management. In particular, explicit `null`-ing of pointers resembles the use of `delete` in C++, which can trigger a chain of class-specific object destructors.

4.3 Experimental Methodology

We quantify garbage collector performance versus that of explicit memory managers by examining the performance of eight benchmarks across a range of algorithms and allocators. Table 4.1 presents the benchmarks used for these experiments. We include the relevant

	Simulated PowerPC G5 system	**Actual PowerPC G4 system**
	Memory hierarchy	
L1, I-cache	64KB, direct-mapped, 3 cycle lat.	32KB, 8-way assoc., 3 cycle lat.
L1, D-cache	32KB, 2-way assoc., 3 cycle lat.	32KB, 8-way assoc., 3 cycle lat.
L2 (unified)	512KB, 8-way assoc., 11 cycle lat.	256KB, 8-way assoc., 8 cycle lat.
L3 (off-chip)	*N/A*	2MB, 8-way assoc., 15 cycle lat.
	all caches have 128 byte lines	*all caches have 32 byte lines*
	Main memory	
RAM	270 cycles (135ns)	95 cycles (95ns)

Table 4.2. The memory timing parameters for the simulation-based and "live" experimental frameworks (see Section 4.1.3). The simulator is based upon a 2GHz PowerPC G5 microprocessor, while the actual system uses a 1GHz PowerPC G4 microprocessor.

SPECjvm98 benchmarks [97].[2] ipsixql is a persistent XML database system from the DaCapo benchmark suite[3] [28], and pseudoJBB is a fixed-workload variant of SPECjbb [96]. pseudoJBB executes a fixed number of transactions (70,000), which simplifies performance comparisons. It is widely considered to be the most representative of a server workload [8] and has a significant memory footprint.

For each benchmark, we run each garbage collector with heap sizes ranging from the smallest needed by that specific collector to complete, up to a heap size four times larger. For our simulated runs, we use the memory and processor configuration of a PowerPC G5 processor [5], and assume a 2 GHz clock. We use a 4KB page size, as used in Linux and Windows. We run the "live" oracle on a 1GHz PowerPC G4 microprocessor and list this memory and processor configuration for comparison [6]. Table 4.2 presents the exact architectural parameters.

Table 4.3 lists the garbage collectors and explicit memory managers we examine in these experiments. All of the garbage collectors are high-throughput "stop-the-world" collectors included in the MMTk memory management toolkit [27]. They define a range

[2]We exclude only the multi-threaded raytracer and non-memory-intensive benchmarks.

[3]In Section 5 we also include jython. We do not include this benchmark in these experiments, because jython cannot be run within DSS.

Memory Managers Examined	
	Garbage collectors
GenCopy	two generations with copying mature space
GenRC	two generations with reference counting mature space
GenMS	two generations with non-copying mature space
CopyMS	whole-heap variant of GenMS
MarkSweep	whole-heap non-relocating, non-copying
SemiSpace	whole-heap two-space
	Allocators
MSExplicit	MMTk's MarkSweep with explicit freeing
Lea	combined quicklists and approximate best-fit
Kingsley	segregated fits

Table 4.3. Allocators and collector examined in these experiments. Section 4.3 presents a more detailed description of each memory manager.

algorithms, including copying, non-copying, and reference counting, whole-heap and generational collectors. The generational collectors (whose names start with "Gen") use an Appel-style variable-sized nursery [14]: the nursery shrinks as survivors fill the heap.[4] We also consider three allocators for use with the oracular memory manager. These include the two allocators often cited as the "fastest" or "best" [20, 70, 111], the Kingsley [111] and Lea (GNU libc, "DLMalloc") [78] allocators. We now describe each of the memory managers in more detail.

GenCopy: GenCopy also uses bump pointer allocation. It allocates into a *nursery* (young space) and copies survivors into a SemiSpace mature space. The collector uses a write barrier to record pointers from mature to nursery objects. GenCopy collects when the nursery is full, and reduces the nursery size by the size of the survivors (an Appel-style variable-sized nursery). When the mature space fills the heap, GenCopy collects the entire heap.

[4]While we wish to examine MarkCopy [88], and MC^2 [89] the implementations of these are not yet stable on the version of Jikes we examined. We expect that MarkCopy and MC^2 would perform faster and consume less space than the collectors we test, but do not expect the differences would be large enough to alter our conclusions.

GenRC: GenRC is another generational collector that differs from GenCopy only in the handling of the mature space. GenRC reclaims objects from the mature space using deferred reference counting and a backup tracing collector for cycles. [30]

GenMS: This hybrid generational collector is similar to GenCopy except that GenMS uses a MarkSweep mature space.

CopyMS: CopyMS is a collector that is similar to GenMS. CopyMS uses a bump pointer to allocate into an Appel-style variable-sized nursery. Survivors from the nursery are copied into a MarkSweep mature space. Unlike GenMS, CopyMS collects both the nursery and mature space on every collection.

MarkSweep: MarkSweep organizes the heap into fixed-size chunks which are then subdivided into blocks. Blocks are assigned a size class and contain only objects of that size class. Space on a block is managed using a freelist and the size classes each maintain a list of their blocks. During collections, MarkSweep traces and marks the reachable objects. Empty blocks are reclaimed and may be reassigned to any size class. Otherwise, individual blocks are returned to their size class's list and swept lazily during allocation.

SemiSpace: SemiSpace maintains two semi-spaces into which it allocates using a bump pointer. It allocates into one of the semi-spaces until full. SemiSpace then copies the reachable objects into the other semi-space and swaps the semi-spaces.

MSExplicit: The MSExplicit allocator isolates the impact of explicit memory management from other changes by adding individual object freeing to MMTk's MarkSweep collector and large object manager ("Treadmill") [18]. Each block of memory maintains its own stack of free slots and reuses the slot that has been most recently freed. MSExplicit uses the allocation routines already in MMTk.

Lea: We used version 2.7.2 of the Lea allocator. It is a hybrid allocator whose behavior depends on the object size, although objects of different sizes may be adjacent in memory. Objects less than 64 bytes in size are allocated using exact-size quicklists (one list exists for each multiple of 8 bytes). Lea coalesces objects in these lists in response to several conditions, such as requests for medium-sized objects. Medium-sized objects are managed with immediate coalescing and splitting of the memory on the quicklists and thus approximates best-fit. Large objects are allocated and freed using `mmap`.

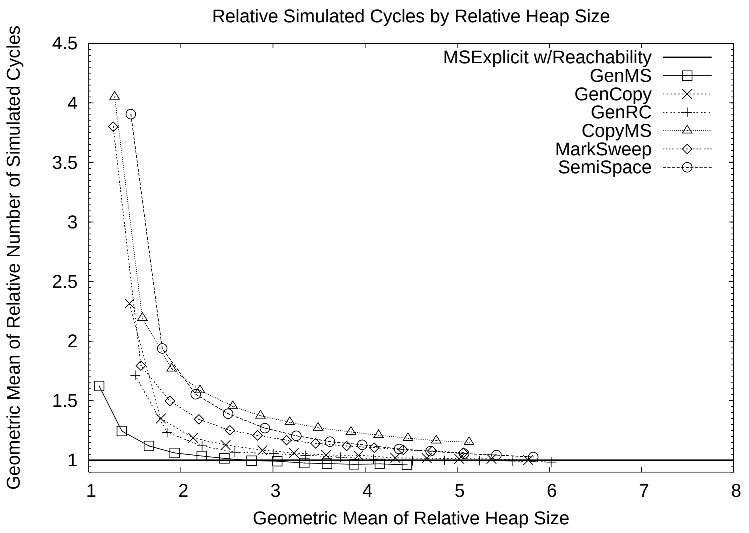
Kingsley: The Kingsley allocator is a segregated fits allocator. Allocation requests are normally rounded up to the nearest power of two, but we modified it slightly to use exact size classes for every multiple of 4 bytes up to 64 bytes and powers-of-two only afterward. We added these size classes to match the segregated size class behavior in MMTk. Once space is allocated for a given size, it can never be reused for another size: the allocator does not break large blocks into smaller ones (*splitting*) or combine adjacent free blocks (*coalescing*).

4.4 Experimental Results

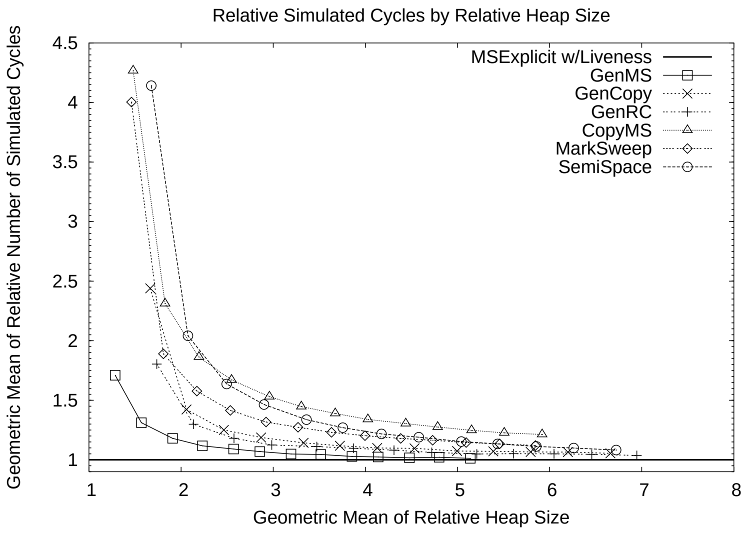
In this section, we explore the impact of garbage collection and explicit memory management on total execution time, memory consumption, and page-level locality.

4.4.1 Run-time and Memory Consumption

Figure 4.3 presents the geometric mean of garbage collection performance relative to the MSExplicit allocator. Each point in these graphs represent the relative heap size (as reported by MMTk) along the x-axis versus the relative number of simulated cycles needed to execute the benchmark along the y-axis. We present the run-time versus reported heap size results for individual benchmarks across the most commonly used garbage collectors



(a) Mean performance of each garbage collector relative to the MSExplicit allocator using the reachability oracle



(b) Mean performance of each garbage collector relative to the MSExplicit allocator using the liveness oracle

Figure 4.3. Geometric mean of garbage collector performance relative to the MSExplicit explicit memory allocator.

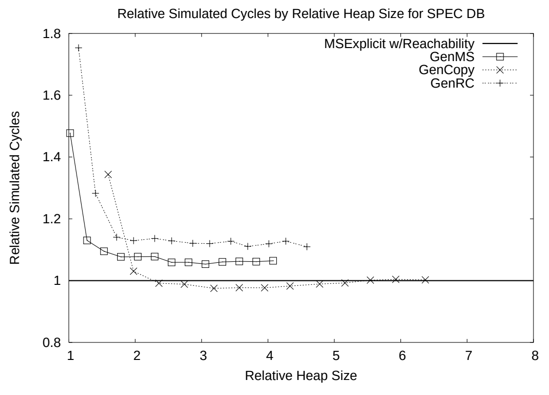
in Figure 4.4. Table 4.4 summarizes the results for the relative performance of GenMS, the best-performing garbage collector.

As the figures show, all of the garbage collectors exhibit similar trends. Initially, in small heaps, the cost of frequent garbage collection dominates run-time. As heap sizes increase, the number of garbage collections decreases correspondingly. Eventually, total execution time asymptotically approaches a fixed value. For GenMS, this value is lower than the cost of the Jikes RVM-based explicit memory manager. At its largest heap size, GenMS averages 4% fewer cycles than MSExplicit using the reachability oracle. GenMS's best performance relative to MSExplicit on each benchmark ranges from executing ipsixql up to 10% faster to running _209_db, a benchmark that is unusually sensitive to locality effects, 5% slower.

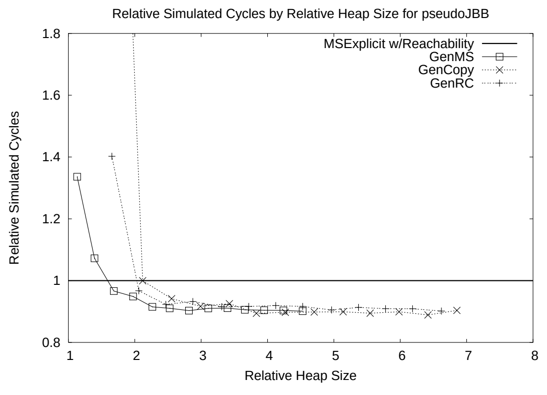
The performance gap between the collectors is smallest for benchmarks with a low allocation intensity (the ratio of total bytes allocated over maximum reachable bytes). For these benchmarks, MarkSweep tends to provide the best performance, especially at smaller heap multiples. Unlike the other collectors, MarkSweep does not need a copy reserve, and so makes more effective use of the heap. As allocation intensity grows, the generational garbage collectors generally exhibit better performance, although MarkSweep provides the best performance for ipsixql until the heap sizes become twice as large as that needed by MSExplicit. The generational collectors (GenMS, GenCopy, and GenRC) exhibit similar performance trends throughout, although GenMS is normally faster. GenMS's MarkSweep mature space also enables the collector to be more space-efficient than both GenCopy (with its SemiSpace mature space) and GenRC's deferring of reference counts.

4.4.1.1 Reported Heap Size Versus Heap Footprint

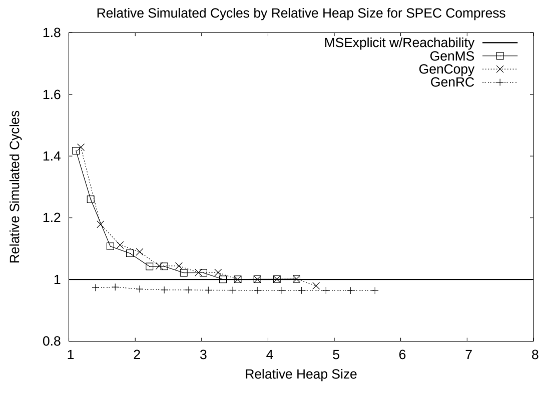
The results reported above use the heap sizes reported by MMTk. While this space metric is useful for comparisons with the MSExplicit allocator, MMTk's reported heap size has three limitations. First, since neither the Kingsley nor Lea allocators record the number



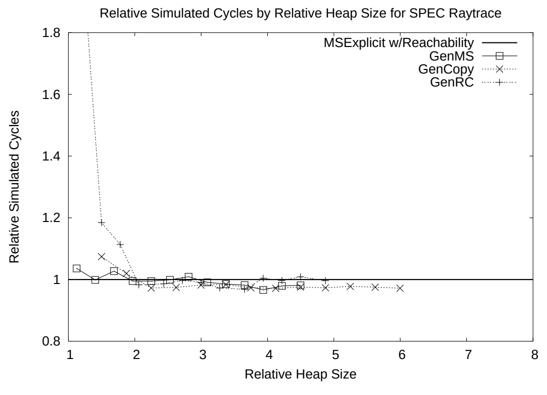
(a) _209_db (5.82)



(b) pseudoJBB (8.45)

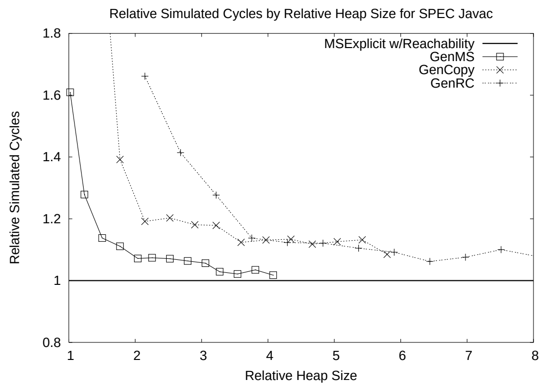


(c) _201_compress (9.16)

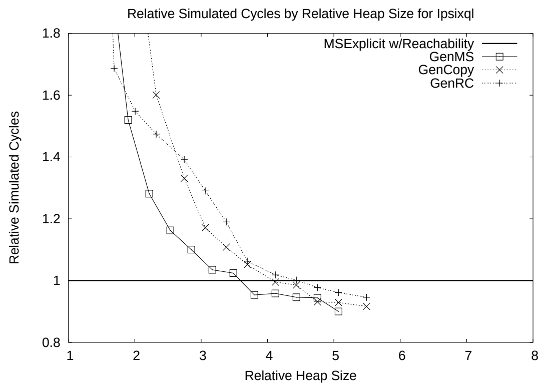


(d) _205_raytrace (14.25)

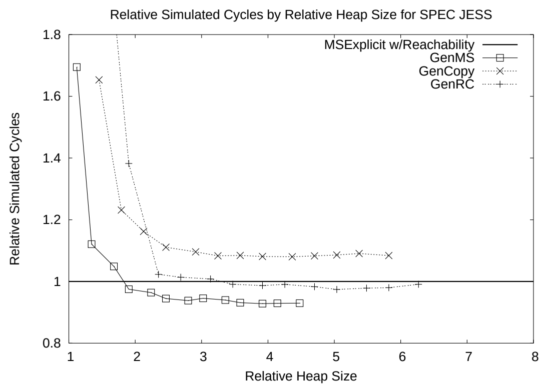
Figure 4.4. Run-time of garbage collectors relative to the MSExplicit allocator using the reachability oracle. These graphs are presented in increasing allocation intensity (allocation/max reachable, given in parentheses); allocation intensity increases going right and down.



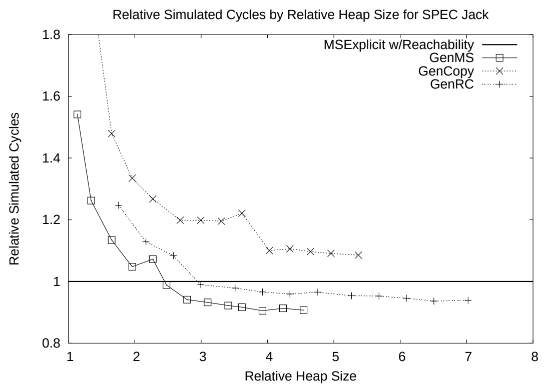
(e) _213_javac (16.27)



(f) ipsixql (23.84)



(g) _202_jess (36.02)



(h) _228_jack (39.63)

Figure 4.4. (continued)

	GenMS			
	vs. MSExplicit w/ Reachability		vs. MSExplicit w/ Liveness	
Heap size	**Rel. Heap Size**	**Execution time**	**Rel. Heap Size**	**Execution time**
1.00	1.11	1.62	1.28	1.71
1.25	1.36	1.25	1.57	1.31
1.50	1.65	1.12	1.91	1.18
1.75	1.93	1.06	2.23	1.12
2.00	2.23	1.03	2.57	1.09
2.25	2.47	1.02	2.85	1.07
2.50	2.77	1.00	3.19	1.06
2.75	3.05	0.99	3.52	1.04
3.00	3.34	0.98	3.85	1.03
3.25	3.59	0.97	4.14	1.02
3.50	3.88	0.97	4.48	1.02
3.75	4.16	0.97	4.80	1.02
4.00	4.45	0.96	5.14	1.01

Table 4.4. Geometric mean of reported heap size and execution times for GenMS relative to MSExplicit. Along the side, we list the GenMS heap sizes in terms of multiples of the minimum heap size required for GenMS to run.

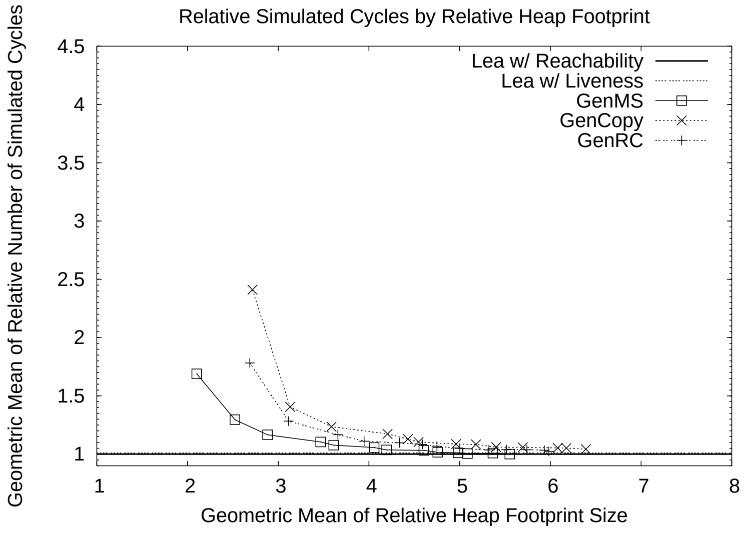
of heap pages they use, we cannot rely on self-reports to measure memory usage. Another problem is that while the Lea and Kingsley allocators include memory Jikes RVM allocates with `malloc` (used for JNI), MMTk does not include these pages in its own accounting. Finally, MMTk does not account for every page used, making comparisons with Kingsley and Lea difficult. When accounting for space consumed, MMTk excludes metadata pages allocated during garbage collection, pages being used for double-buddy allocation that are not holding heap objects, and pages of memory and pages that previously held heap objects but are not currently in use.

We therefore need a more general way of measuring space consumption that compares memory demands more fairly. While difficult to measure within a system, within the simulator we can measure *heap footprints* for each run of an application. A heap footprint is the high watermark of heap pages in use. Pages are *in use* only if they have been both allocated from the kernel and *touched* (allocated in memory). The heap footprint therefore measures the pages that are either in RAM or are stored on disk. Besides serving as a useful metric of space consumption, heap footprints consider the memory demands actually placed on the system.

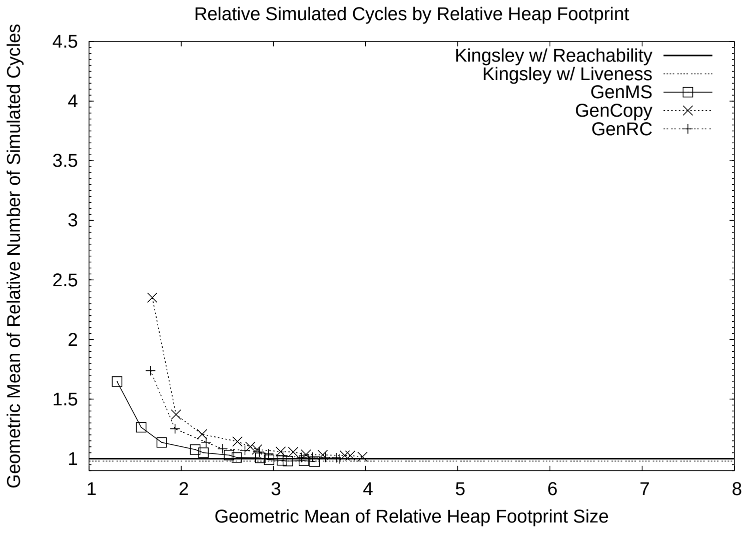
Unlike MMTk's reported heap size, heap footprints measure exactly the number of pages needed to hold the entire heap. Copying collectors maintain copy reserves that are included in the heap size, but may not ever be used. Because these pages are not included in the heap footprint, reported heap sizes can be greater than the footprint. Another problem arises when using MarkSweep spaces. The current implementation of MarkSweep makes reusing heap pages difficult. Thus, runs of MarkSweep exhibit heap footprints up to twice as large as their heap size. Table 4.5 shows both the heap footprint and reported heap size for runs of GenMS. The additional memory demands caused by previously used pages is most noticeable at the smallest heap sizes which perform more collections and therefore have a larger relative turnover of heap pages. Heap footprints thus include exactly (and only) the pages holding the heap and can are especially useful when comparing different memory managers.

Figure 4.5 compactly summarizes the run-time results and presents the time-space tradeoff involved when comparing garbage collection with the Kingsley and Lea explicit memory managers. Because these graphs use heap footprints when measuring space consumption, the will graphs occasionally exhibit a "zig-zag" effect. This effect is surprising as increasing the heap size normally increases the heap footprint. Because of fragmentation and alignment restrictions, however, a larger heap size may prevent an object from straddling two pages and thereby may *reduce* the footprint. MarkSweep, which cannot reduce fragmentation by compacting the heap, is most susceptible to this zig-zag effect. Figures 4.6 and 4.7 present results from individual benchmarks.

Finally, Table 4.6 compares the footprints and run-times of the MSExplicit and the Lea allocators when using the same oracle. This table shows that some of the collectors' space inefficiencies are not due to garbage collection: even the explicitly-managed MSExplicit requires between 38% and 52% more space than Lea. The two allocators perform similarly

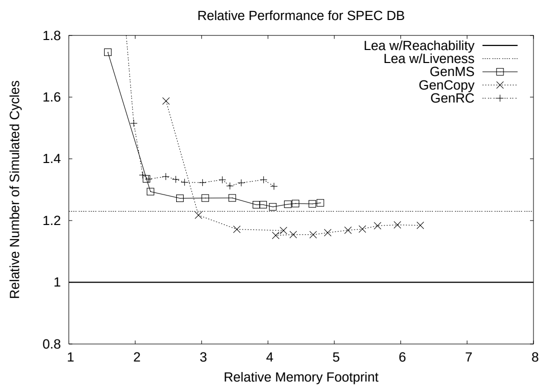


(a) Mean performance of each garbage collector relative to the Lea allocator using the reachability oracle

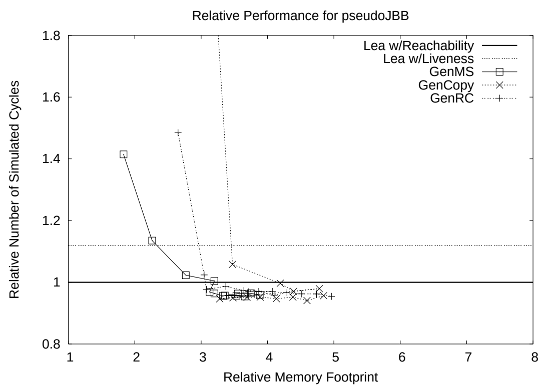


(b) Mean performance of each garbage collector relative to the Kingsley allocator using the reachability oracle

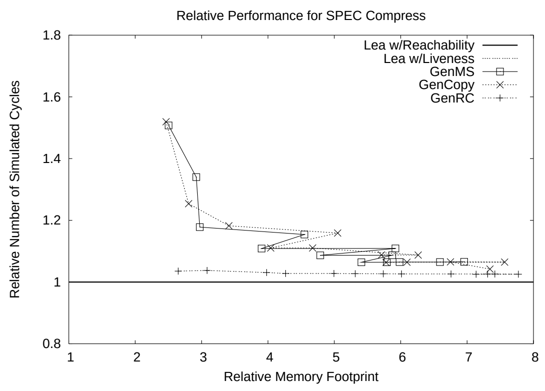
Figure 4.5. Geometric mean of garbage collector performance relative to the Kingsley and Lea explicit memory allocators using the reachability oracle



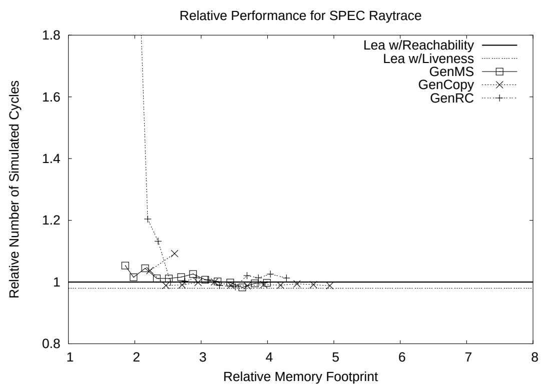
(a) _209_db (5.82)



(b) pseudoJBB (8.45)

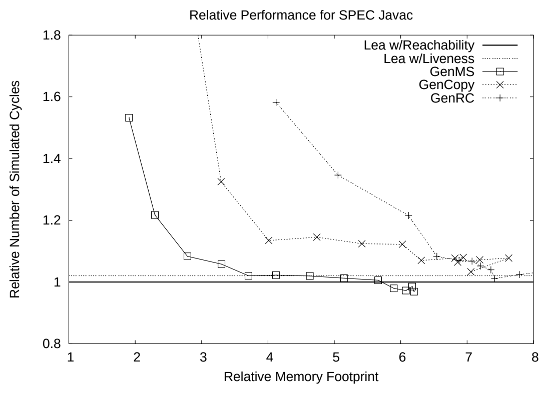


(c) _201_compress (9.16)

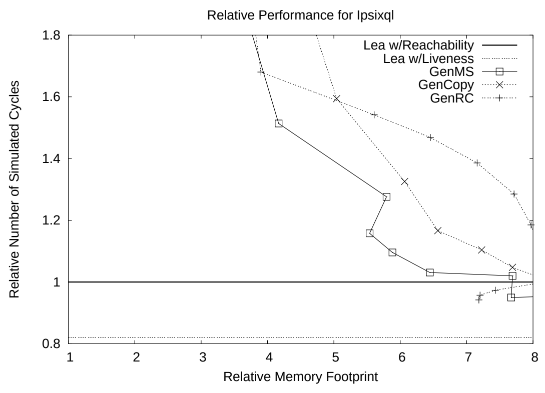


(d) _205_raytrace (14.25)

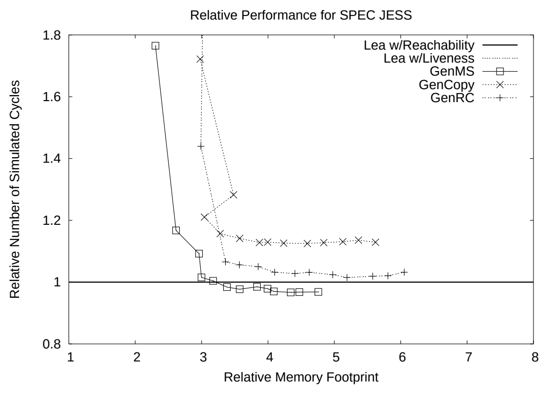
Figure 4.6. Run-time of garbage collectors relative to the Lea allocator using the reachability oracle. The "zig-zag" results from fragmentation's causing the maximum number of heap pages in use to vary (see Section 4.4.1). These graphs are presented in increasing allocation intensity (allocation/max reachable, given in parentheses); allocation intensity increases going right and down.



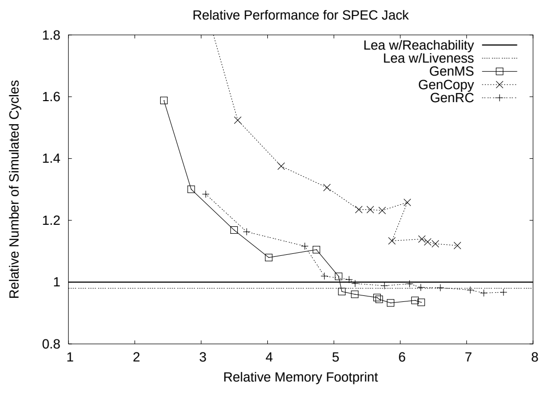
(e) _213_javac (16.27)



(f) ipsixql (23.84)

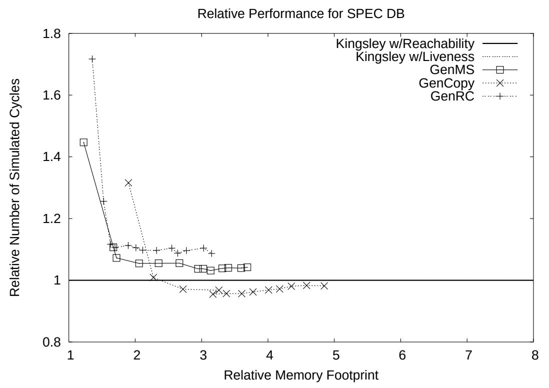


(g) _202_jess (36.02)

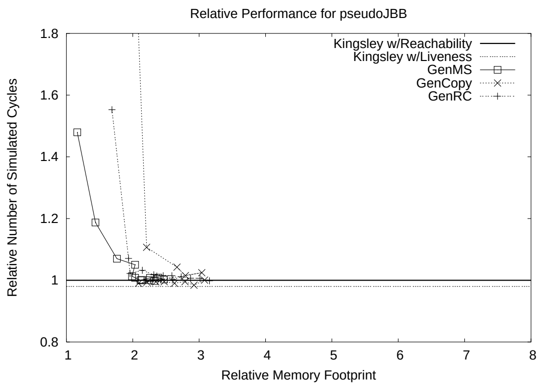


(h) _228_jack (39.63)

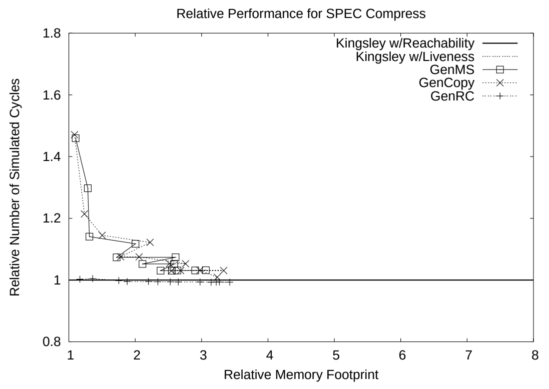
Figure 4.6. (continued)



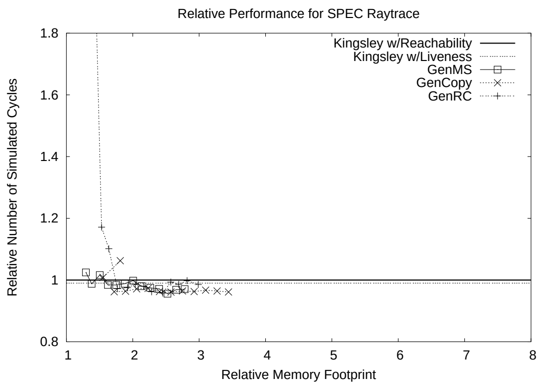
(a) _209_db (5.82)



(b) pseudoJBB (8.45)

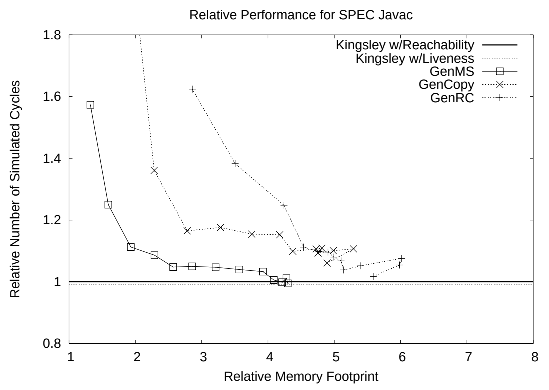


(c) _201_compress (9.16)

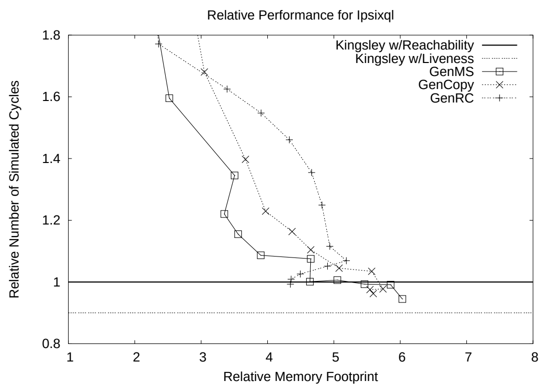


(d) _205_raytrace (14.25)

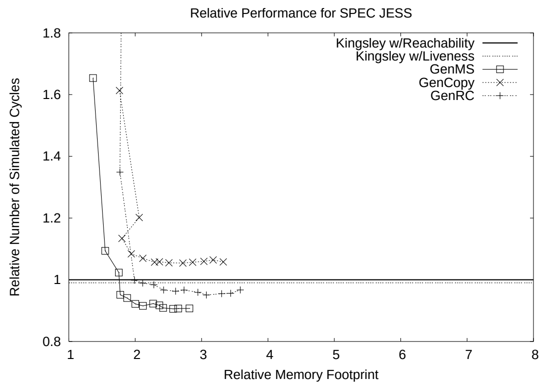
Figure 4.7. Run-time of garbage collectors relative to the Kingsley allocator using the reachability oracle. The zig-zag results from fragmentation's causing the maximum number of heap pages in use to vary (see Section 4.4.1). These graphs are presented in increasing allocation intensity (allocation/max reachable, given in parentheses); allocation intensity increases going right and down.



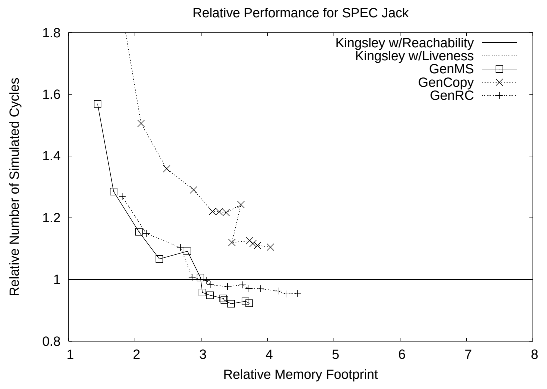
(e) _213_javac (16.27)



(f) ipsixql (23.84)



(g) _202_jess (36.02)



(h) _228_jack (39.63)

Figure 4.7. (continued)

Heap Footprint v. Reported Heap Size Relative to MSExplicit				
	pseudoJBB		Geo. Mean	
Heap Multiplier	Footprint	Heap Size	Footprint	Heap Size
1.00	1.64	1.13	1.53	1.11
1.25	2.02	1.39	1.83	1.36
1.50	2.48	1.68	2.09	1.65
1.75	2.86	1.97	2.52	1.93
2.00	2.79	2.16	2.62	2.23
2.25	2.86	2.53	2.95	2.47
2.50	2.98	2.82	3.05	2.77
2.75	3.18	3.11	3.35	3.05
3.00	3.35	3.40	3.46	3.34
3.25	3.47	3.66	3.62	3.59
3.50	3.00	3.95	3.70	3.88
3.75	3.16	4.24	3.90	4.16
4.00	3.27	4.53	4.03	4.45

Table 4.5. Comparison of the space impact of GenMS relative to MSExplicit using the reachability oracle. We show results when comparing footprints and heap sizes. Along the side we list the GenMS heap size in terms of multiples of the minimum heap size required for GenMS to run.

when comparing run-times, however. With the reachability oracle, MSExplicit runs an average of 4% slower than Lea; with the liveness-based oracle, it runs 2% faster. Segregated size classes cause MSExplicit to run the locality-sensitive _209_db 18% slower when using the reachability oracle. MSExplicit runs 5% faster than Lea on _213_javac, however, because this benchmark stresses raw allocation speed.

With the notable exception of _209_db, the two allocators are roughly comparable in performance, confirming the good performance characteristics both of the generated Java code and of the MMTk infrastructure. Figure 4.6(h) is especially revealing. While the run-time performance of MSExplicit is just 3% slower than Lea, MarkSweep, which uses the same allocation infrastructure, runs at least 50% slower. These experiments demonstrate that the performance differences between explicit memory management and garbage collection are due to garbage collection itself and not underlying differences in allocator infrastructure.

	MSExplicit vs. Lea			
	w/ Reachability		w/ Liveness	
Benchmark	**Footprint**	**Run-time**	**Footprint**	**Run-time**
_201_compress	1.62	1.06	2.51	1.01
_202_jess	1.54	1.04	1.65	1.03
_205_raytrace	1.31	1.02	1.47	1.00
_209_db	1.12	1.18	1.18	0.96
_213_javac	1.33	0.95	1.24	0.93
_228_jack	1.58	1.03	1.68	1.05
ipsixql	1.49	1.00	1.63	0.97
pseudoJBB	1.12	1.06	1.16	0.87
Geo. Mean	1.38	1.04	1.52	0.98

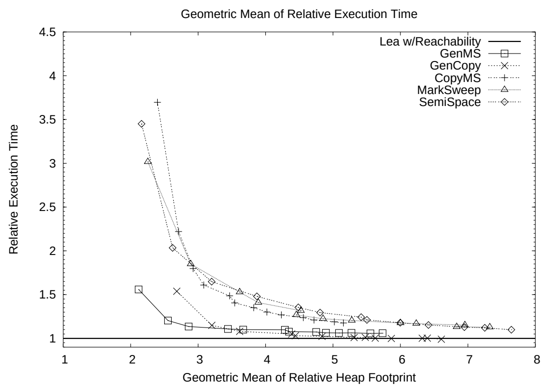
Table 4.6. Relative memory footprints and run-times for MSExplicit versus Lea. In this table, we present results comparing results when run with similar oracles.

4.4.1.2 Comparing Simulation to the Live Oracle

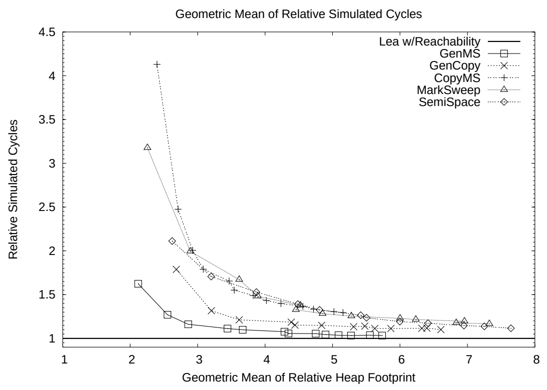
We also compare the run-time performance of our garbage collectors with the live oracle described in Section 4.1.3. For these experiments, we use a PowerPC G4 with 512MB of RAM running Linux in single-user mode and report the mean value of 5 runs. The architectural details of our experimental machine are listed in Table 4.2.

A comparison of the results of our live oracle experiments and simulations can be seen in Figure 4.8. These graphs compare the geometric means of executing all but three of the benchmarks. Because of the memory demands of the garbage collection trace generation process and difficulty in duplicating time-based operating system calls, we are currently unable to run pseudoJBB, ipsixql, and raytrace with the live oracle.

Despite their different environments, the live and simulated oracular memory managers achieve strikingly similar results. Differences between the graphs could be accounted for by the G4's L3 cache and smaller main memory latency compared to our simulator. While the null oracle adds too much noise to our data to justify its use over our simulator, the similarity of the results is strong evidence for the validity of the simulation runs.



(a) Live oracular results



(b) Simulated oracular results

Figure 4.8. Comparison of the results using the "live" oracular memory manager and the simulated oracular memory manager: geometric mean of execution time relative to Lea using the reachability oracle across identical sets of benchmarks.

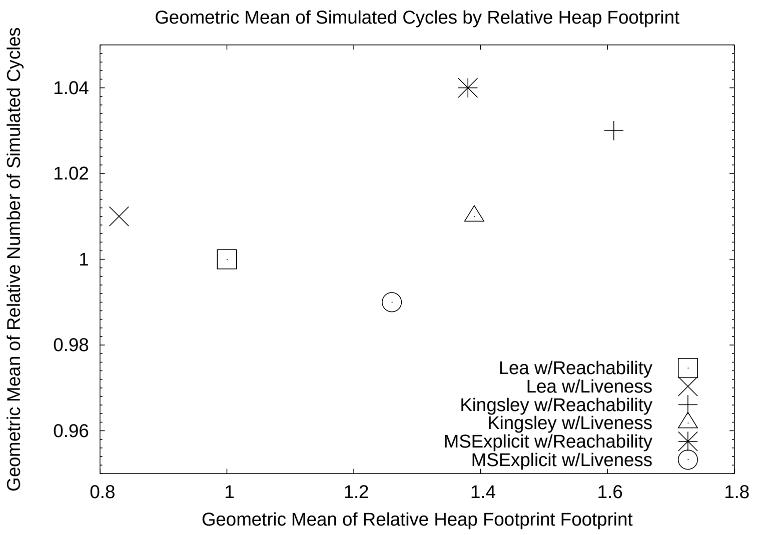


Figure 4.9. Geometric mean of the explicit memory managers relative to the Lea allocator using the reachability oracle. While using the liveness oracle substantially reduces the heap footprint, it has little impact on execution time.

4.4.1.3 Comparing the Liveness and Reachability Oracles

Figure 4.9 shows the effect of using the liveness and reachability-based oracles. This graph presents the mean relative execution time and space consumption of the explicit memory managers using both the liveness- and reachability-based oracles. As usual, the results are normalized to the Lea allocator using the reachability-based oracle. The x-axis shows relative execution time; note the compressed scale, ranging from just 0.98 to 1.04. The y-axis shows the relative heap footprint, and here the scale ranges from 0.8 to 1.7. The summary and individual run-time graphs (Figures 4.5 and 4.6) also include a data point for the Lea allocator with the liveness oracle.

We find that the choice of oracle has little impact on execution time. We had expected the liveness-based oracle to improve performance by recycling memory as soon as possible and thereby improving locality. This recycling, however, has at best a mixed effect on runtime. The liveness oracle actually degrades performance by 1% for the Lea allocator while improving it by up to 5% for MSExplicit.

When the liveness-based oracle does improve runtime performance, it does so by reducing the number of L1 data cache misses. Figure 4.6(f) shows Lea executes ipsixql 18% faster when using the liveness-based oracle. This improvement is due to a halving of the L1 data cache miss rate. But the liveness-based oracle significantly degrades cache locality in both _209_db and pseudoJBB, running them 23% and 13% slower, respectively. While we have already seen that _209_db is highly susceptible to cache effects, the pseudoJBB result is surprising. For pseudoJBB, the lifetime-based oracle results in poor object placement and increases the L2 cache miss rate by nearly 50%. These two benchmarks are outliers. Figure 4.5 shows that, on average, the Lea allocator with the liveness-based oracle runs only 1% slower than with the reachability oracle.

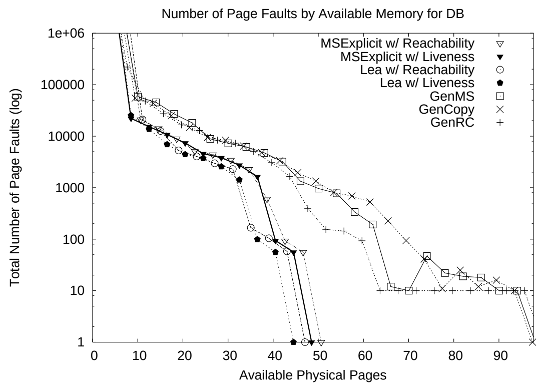
The liveness-based oracle has a much more pronounced impact on space consumption, reducing heap footprints by up to 15%. Using the liveness-based oracle reduces Lea's mean heap footprint by 17% and MSExplicit's mean footprint by 12%. While _201_compress's reliance on several large objects limits the amount the liveness-based oracle can improve its space utilization, all of the other benchmarks see their memory usage drop by at least 10%.

4.4.2 Page-level locality

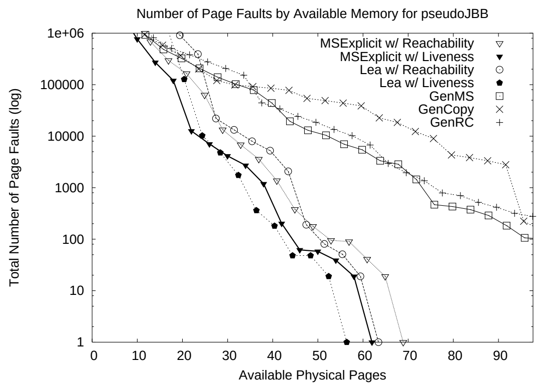
For virtual memory systems, page-level locality can be more important for performance than total memory consumption. We present the results of our page-level locality experiments in the form of augmented *miss curves* [94, 113]. Assuming that the virtual memory manager observes an LRU discipline, these graphs show the time taken (y-axis) at each number of pages allocated to the process (x-axis). We assume a fixed 5-millisecond page fault service time. These graphs are generated by multiplying the number of page faults by the page fault service time to compute the total time a run spent handling page faults. We then added this time to the simulated time needed to run the benchmark shown earlier. In each of these graphs, we present results for each garbage collector assuming we select the heap size that performs best for the given amount of available memory. Note that these graphs use a log-scale for the y-axis.

Figure 4.11 presents the total execution times for the Lea and MSExplicit allocators and each garbage collector across all benchmarks. For each garbage collector, we show only the fastest-performing heap size.

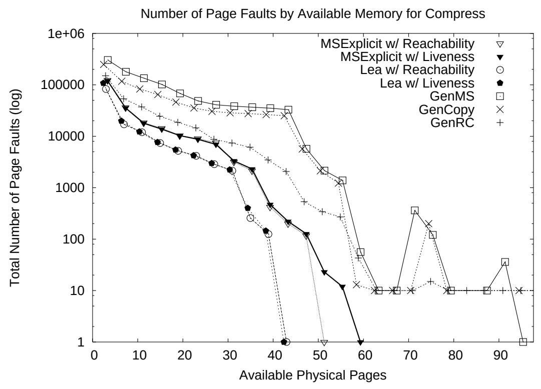
As these graphs show, both explicit memory managers substantially outperform all of the garbage collectors for reasonable ranges of available memory. For instance, the Lea allocator using the reachability oracle with 63MB of available memory executes pseudoJBB in 25 seconds. Given the same amount of available memory, GenMS requires an order of magnitude longer (255 seconds) to complete. This pattern is even more pronounced for _213_javac: at 36MB of available memory, the Lea allocator using the reachability oracle provides a 15-fold decrease in total execution time versus GenMS (14 seconds versus 211 seconds).



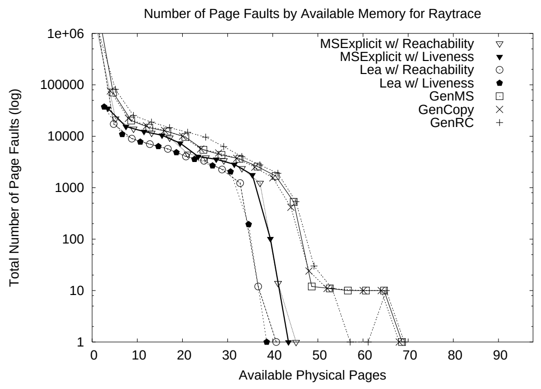
(a) _209_db



(b) pseudoJBB

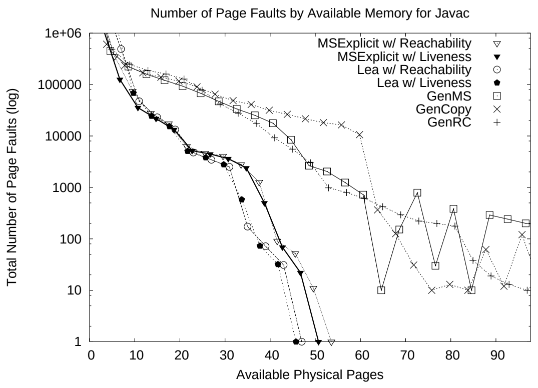


(c) _201_compress

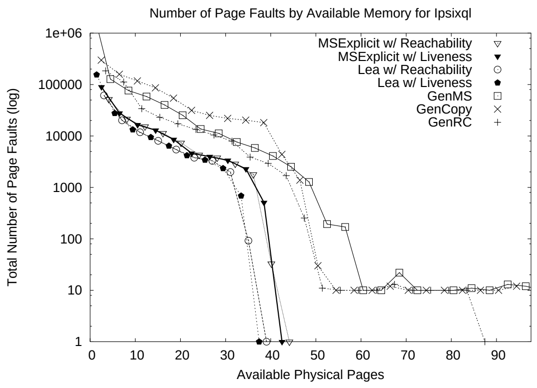


(d) _205_raytrace

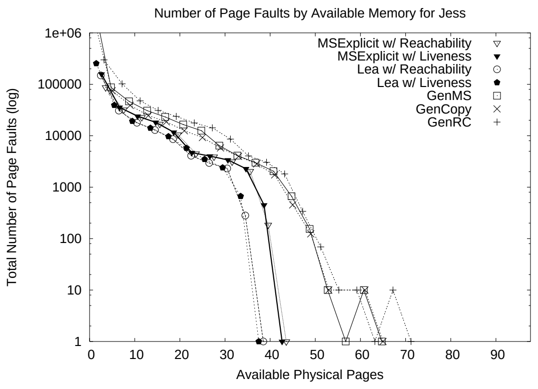
Figure 4.10. Total number of page faults (note that the y-axis is log-scale). For the garbage collectors, each point presents the total number of page faults for the heap size that minimizes the total execution time (including page fault service times) for the collector at each available memory size. Because the best performing heap size changes depending on the available memory size, these curves are not monotonic at small numbers of page faults. These graphs are presented in increasing allocation intensity.



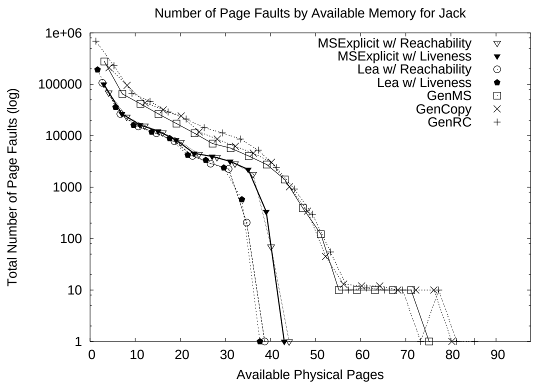
(e) _213_javac



(f) ipsixql

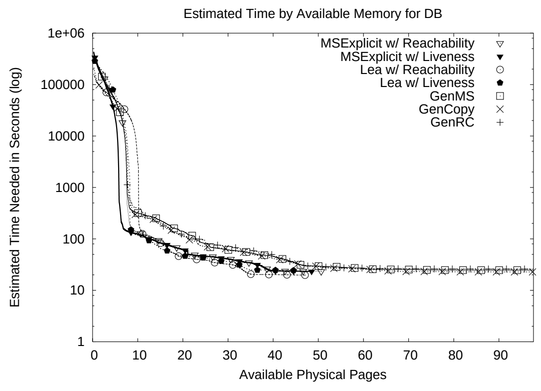


(g) _202_jess

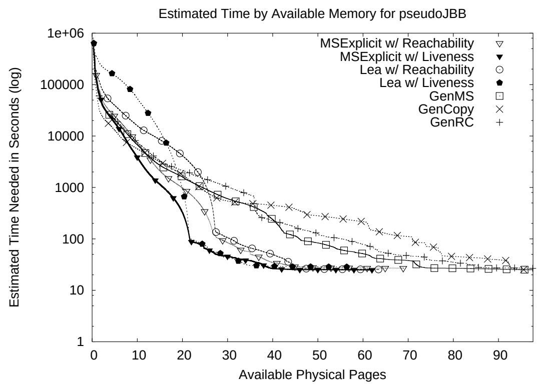


(h) _228_jack

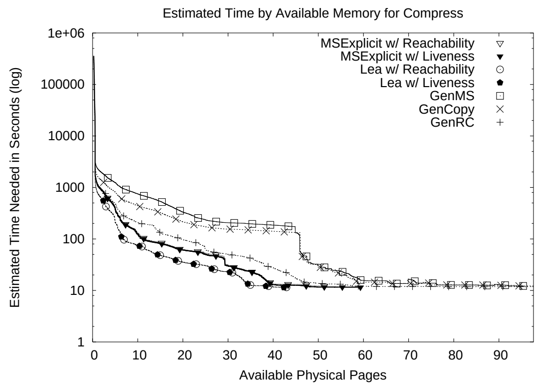
Figure 4.10. (continued)



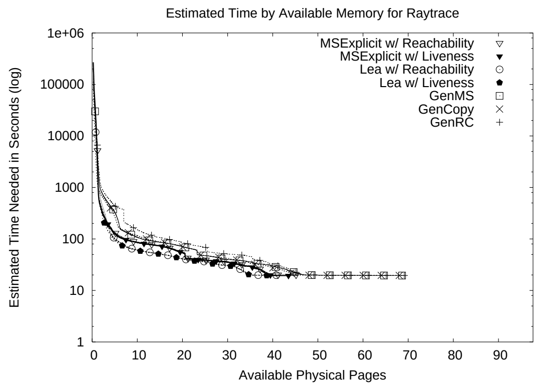
(a) _209_db



(b) pseudoJBB

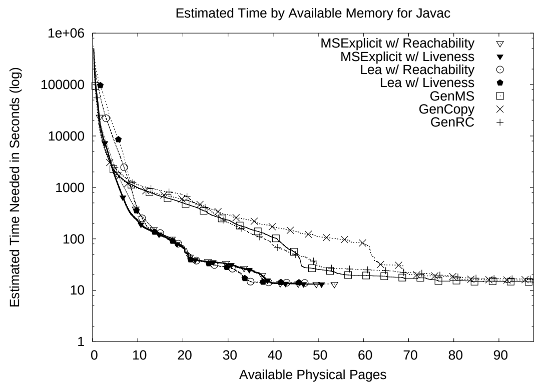


(c) _201_compress

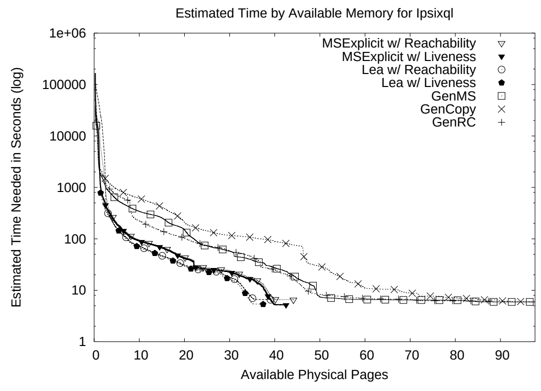


(d) _205_raytrace

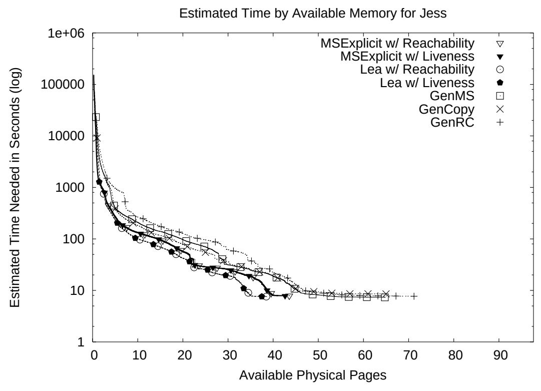
Figure 4.11. Estimated execution times, including page fault service time (note that the y-axis is log-scale). For the garbage collectors, these curves present the estimated execution time for the heap size that minimizes this time for the collector at each available memory size. These graphs are presented in increasing allocation intensity.



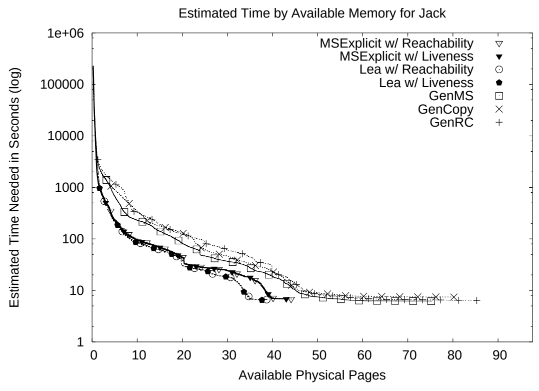
(e) _213_javac



(f) ipsixql



(g) _202_jess



(h) _228_jack

Figure 4.11. (continued)

The culprit here is garbage collection activity. During a collection, the garbage collector visits far more pages than the application itself [113]. As allocation intensity increases, the number of collections (and for generational collectors, major collections) increases. Since each collection is likely to visit evicted pages, the performance gap between the garbage collectors and explicit memory managers grows with each collection.

4.5 Conclusion

This chapter presents the oracular memory manager with which we perform the first apples-to-apples quantification of garbage collection performance relative to that of another memory manager of which we are aware. This process begins with our system generating traces and computing the lifetime and reachability information for all of the objects. Using this information as an oracle specifying when to insert calls to `free`, our experimental methodology executes unaltered Java programs as if they used explicit memory management. Executing within a cycle-accurate simulator, the oracular memory manager eliminates the cost of consulting these oracles while including the costs of object allocation and reclamation.

We then use this framework to quantify the time and space performance of garbage collection relative to of explicit memory management. The results in this chapter show that when memory is plentiful, garbage collection performance can be slightly better than that of explicit memory management. With a heap size nearly four-and-a-half times larger, GenMS provides a 4% improvement in execution time versus a similarly implemented explicit memory manager using the reachability oracle. At a heap size between nearly three and slightly above five times as large, the best performing collector matches the explicit memory manager's performance with the reachability and liveness oracles, respectively. In the smallest heap size in which it can complete, the garbage collector not only continues to have a heap size 11% larger than the explicit memory manager using the most conservative oracle, but also degrades throughput by an average of nearly 62%. While using the lifetime oracle does allow the explicit memory manager to report heaps that are 20% smaller, on average, we also show that it has little impact on execution times. We also show that this good performance relies upon the system having sufficient physical memory to hold the entire heap. When insufficient available memory causes paging, garbage collection runs orders-of-magnitude slower than explicit memory management.

We use these results to improve garbage collection's performance where it is most brittle – its paging performance. The next chapter presents a new garbage collection algorithm that works to avoid touching evicted pages and therefore provides good performance in a variety of situations.

CHAPTER 5
BOOKMARKING COLLECTOR

Our results show that garbage collection performance can match, if not exceed, explicit memory management performance when the heap fits in available memory. When memory is scarce, the situation is quite different. Garbage collectors visit many more pages than the application itself [113]. This increased memory pressure is compounded by the garbage collector accessing pages regardless of whether they reside in memory. As we show in the simulation results in Section 4.4.2 and in real-world experiments in Section 5.7.3, this behavior induces paging. Because of this paging, throughput plummets and pause times spike up to seconds or even minutes.

In this chapter, we present a garbage collector that avoids paging. The *bookmarking collector* cooperates with the virtual memory manager to guide eviction decisions. The bookmarking collector records summary information (the *bookmarks* for which the collector is named) from evicted pages. Using these bookmarks, the collector can perform whole-heap in-memory collections. In this chapter, we first present an overview of the bookmarking collector. We then discuss key implementation details, including necessary extensions to the Linux virtual memory manager. Next, we compare the new collector's performance with that of a variety of existing collectors. We finally use the oracular memory manager to examine the relative performance of the bookmarking collector and several explicit memory managers.

5.1 Bookmarking Collector Overview

The bookmarking collector (BC) was designed to achieve three goals: low space consumption, high throughput, and the elimination of garbage collection-induced page faults. It accomplishes these goals through a number of mechanisms. BC provides high throughput by using a nursery generation to manage short-lived objects. While BC normally uses mark-sweep collection for the mature space, it minimizes space consumption by compacting the heap when memory pressure rises. Finally, and most importantly, it reacts to signals sent by the virtual memory manager whenever the virtual memory manager schedules a page for eviction to disk or reloads a page from disk into main memory.

Unlike existing garbage collectors, BC avoids paging in several ways. This process starts with the virtual memory manager alerting BC that it scheduled one of the virtual machine's pages for eviction. When notified of the impending eviction, BC first attempts to avoid the eviction by providing the virtual memory manager with an alternate physical page that is not currently being used. If BC cannot find such a *discardable* page, it collects the heap. After this garbage collection, BC looks to see if it now has an empty page to discard. When there are still not any empty pages, so a page must be evicted, BC works to ensure that only appropriate pages are evicted. Prior to the page's eviction, BC scans each of its objects and *bookmarks* the target of any references. BC also increments the counter of referring evicted page for all pages containing a target object. After processing all the page's objects, BC informs the virtual memory manager that this page is a good page to evict. In later collections, BC can use the bookmarks to remember the evicted references without reloading the objects. Thus, BC can collect the heap without touching the evicted pages.

Figure 5.1 shows the mature space of an idealized heap after a collection using bookmarks. Objects are presented as the small rectangles inside the larger rectangles, which represent heap pages. Slots available for allocation (provably unreachable objects) are in white, while other objects are black. The light gray (middle) page in this figure has been

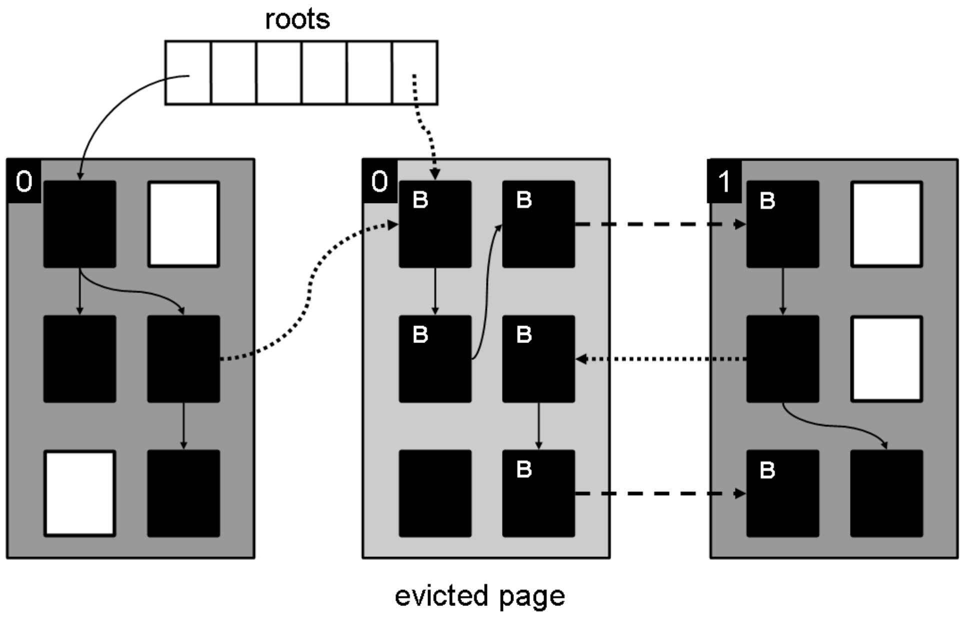


Figure 5.1. An example of bookmarking collection. In this illustration, provably unreachable objects are white and all other objects are black. Bookmarks are shown as a letter "B" in the bookmarked object's top-left corner. The 0 or 1 in each page's top-corner is the value of the counter of evicted pages containing references to it. Because nursery pages cannot be evicted or contain bookmarked objects, it is not shown here.

evicted to disk. We use lines to represent intra-heap references. Dotted lines refer to objects on pages processed for eviction; BC will ignore these references during collection, because it first checks the reference target against its record of evicted pages. The dashed lines denote references from the non-resident page: these references induce bookmarks, which are represented by the letter "B". As Figure 5.1 shows, BC conservatively bookmarks the objects on a page before nominating the page for eviction. This conservative bookmarking enables BC to find the empty slots should the page be reloaded and is discussed in more detail in Section 5.4. This figure also shows a number in the upper-left hand corner of each page. This number records how many evicted pages contain references to objects on the page. Thus the first two pages contain a count of "0" and the last page contains a count of "1." Section 5.4.2 details these counters in more detail. BC treats all bookmarked objects as reachable, even if all references to an object have been evicted.

5.2 Bookmarking Collector Details

The bookmarking collector is a generational collector with a bump-pointer nursery, compacting mature space, and page-based large object space. BC divides the mature space into *superpages*, page-aligned groupings of four contiguous pages. In Linux, each superpage is 16KB. BC manages mature objects using *segregated size classes* [15, 16, 21, 78, 80, 87, 111]: objects of different sizes are allocated onto different superpages. Once empty, a superpage can be reassigned to any size class [21, 80].

When evicting or reloading a page, it is important that BC identify all of its objects. BC can do this quickly, because each superpage holds objects of a single size class. Given a page, BC uses a bitmask to determine its superpage. The start of each superpage then contains a small amount of metadata (the *superpage header*) including the superpage's size class. BC uses this size class to compute where objects on the page may lie. While MMTk already includes a segregated- size-class-based allocator, this allocator uses a range of block sizes and would not be able to do this object scanning. BC therefore uses its own allocator.[1]

BC uses own size classes designed to minimize fragmentation for its superpage-based allocation. We begin with a hard 25% bound for external fragmentation and then select size classes. Each allocation size up to 64 bytes has its own size class. Larger object sizes fall into a range of 37 size classes; all but the largest five have a worst-case combined internal and page-internal fragmentation of 15%. The five largest classes have between 16% and 33% worst-case combined internal and page-internal fragmentation; we cannot do better without violating our bound on external fragmentation. BC allocates objects that are larger than half the usable space of a superpage (8180 bytes) into the page-based large object space.

[1]Earlier comparisons of the two allocators, performed using Jikes RVM 2.2.2, showed little time or space performance differences. As the primary design focus of the MMTk allocator is reducing fragmentation [25], we expect this still holds.

When the mature space fills, BC typically performs mark-sweep garbage collection. We use mark-sweep for two reasons. First, it provides good program throughput and short GC pauses. More importantly, mark-sweep does not need a copy reserve of pages that would increase memory pressure.[2]

5.2.1 Managing Remembered Sets

Like all generational collectors, BC must remember references from the older generation to the nursery. It normally stores these references in page-sized sequential store buffers. Sequential store buffers provide fast storage and processing, but BC cannot limit the amount of space they require. To reduce this space overhead, BC processes each buffer when it fills. During this processing, BC removes from the buffer references from objects in the mature space and instead marks the card for the source object in the card table BC uses during the marking phase. BC does keep references from objects in the large object space, because of the time that could be required scanning these object. BC also preserves references from objects in Jikes RVM's boot image, because they are not included in any card tables. After pruning all possible references and compacting the remaining entries in the buffer, BC makes the removed slots available for future storage. When a nursery collection begins, BC processes both the sequential store buffers and objects whose cards are marked. By filtering the remembered sets, the bookmarking collector gains the fast processing of sequential store buffers while often fitting the buffer onto a single page.

5.2.2 Compacting Collection

Using mark-sweep, the mature collector cannot reclaim a superpage if it contains a single reachable object. This external fragmentation can lead mark-sweep collection to increase memory pressure. BC avoids this increased memory pressure by performing a two-

[2]Usually this copy reserve is half the mature space size, but Sachindran et al. present copying collectors that allow the use of smaller copy reserves [88, 89]. As we show in Section 5.2.2, BC performs compaction without any copy reserve.

pass compacting collection whenever a mark-sweep collection does not recover enough pages to preserve the allocation reserve MMTk requires.

BC begins the compacting collection with a marking phase. Each time it marks an object, BC also increments a counter for the object's size class. After it has marked the mature space, BC computes the minimum number of superpages needed to hold the marked objects for each size class and then selects a minimum set of superpages. BC performs this selection by setting a bit in the headers of these *target* superpages onto which BC will compact all of the marked objects (Section 5.4.1 discusses the selection of target superpages when the heap contains bookmarked objects).

BC then begins a second pass. This pass also uses a Cheney scan to find the reachable objects. When processing an object in this pass, BC first checks if the object is on a target superpage. If the object is already on a target superpage, BC only scans it for references. BC copies all other objects (e.g., objects not on a target superpage) onto the next available slot on a target superpage of the same size class. When this pass completes, BC will have compacted the reachable objects onto the target superpages. BC therefore unsets the bit marking the target superpages and frees the remaining (garbage) superpages.

5.3 Cooperation with the Virtual Memory Manager

The approach described so far allows BC to avoid increasing memory pressure during garbage collection. BC further cooperates with the virtual memory manager to reduce memory pressure in the face of paging. The bookmarking collector first reduces memory pressure by shrinking the heap and then using its knowledge of the heap to help the virtual memory manager make good eviction decisions. This cooperation requires an extended virtual memory manager that can communicate with the garbage collector. Table 5.1 lists the interfaces that drive this communication. We describe the kernel changes these extensions require in Section 5.6.1.

BC ⇔ Virtual Memory Manager Interface
`int vm_register(1)`   Call to register a process with the virtual memory manager. This causes the virtual memory manager to send the process a signal whenever the process has a page scheduled for eviction.
`int madvise(void *start, size_t length, MADV_DONTNEED)`   Previously existing call in Linux to advise the virtual memory manager that the pages listed in `start` are discardable and can be reclaimed.
`int vm_relinquish(void *start, size_t length)`   System call informing the virtual memory manager that the pages in start are good eviction candidates and should be moved to the end of the LRU queue.

Table 5.1. Interface through which the bookmarking collector communicates with the extended virtual memory manager.

5.3.1 Reducing System Call Overhead

Communication with the virtual memory manager imposes its own overhead on BC. We therefore designed the collector to limit this overhead. BC maintains a bit array that records whether pages are memory resident; it also tracks the current heap footprint. Whenever it assigns a superpage to a size class and makes the superpage available for allocation, BC checks in the bit array if the superpage is already memory-resident (e.g., from a previous use in the heap). When the superpage is not resident, BC updates its current heap footprint size and marks the pages of the superpage as *in-core* (memory-resident). BC also uses the bit array during garbage collection. When determining whether a page has been evicted, BC need only check the bit array and does not need to make more expensive calls into the virtual memory manager.

5.3.2 Discarding Empty Pages

We further reduce the overhead by initiating communication when the virtual memory manager sends a signal that it has scheduled a page for eviction. Upon receiving this signal, BC scans a bit array for a memory-resident but empty page. If found, BC directs the virtual memory manager to reclaim this *discardable* page. Otherwise, BC triggers a collection and directs the virtual memory manager to discard a newly-emptied page (if one exists). Most operating systems (e.g., Linux and Solaris) already include the capability for a process to direct the reclamation of a page by the `madvise` system call using the `MADV_DONTNEED` flag; the Windows virtual memory manager offers similar functionality.

5.3.3 Keeping the Heap Memory-Resident

When notified of a pending eviction, BC is also alerted that the current heap footprint cannot fit within the memory available to the application. Unlike previous collectors, BC will not to grow at the expense of paging, but instead uses that footprint as an upper bound. If an allocation request would require that BC increase the heap footprint, BC will do so only if it has sufficient discardable pages to return to the virtual memory manager. Otherwise, BC collects to try and make space for the allocation. Thus heap footprint acts as a new smaller limit to heap growth. If memory pressure continues to increase (e.g., if BC continues to receive further notifications), BC discards more empty pages and replaces its previous heap footprint bound with the new, lower estimate. BC thus shrinks the heap to keep it entirely in-core and avoid evicting a page and the possibility of incurring page faults.

Unfortunately, it may be possible that the program requires a larger heap than will fit into available memory. If after a collection there is still not enough available memory to fulfill an allocation, BC allows this allocation to continue even at the potential cost of having to evict pages. Barring this limiting possibility, however, BC will not allow the footprint to grow beyond what is available in memory.

5.3.4 Utilizing Available Memory

BC may need to discard empty pages, only later to have the memory become available again. While growing the heap can cause paging, it is important that a brief drop in available memory not limit program throughput throughout the entire execution. Rather than simply grow the heap blindly, BC occasionally ends a collection by polling the virtual memory manager to determine how much memory is currently available. Using the amount of memory available, BC increases its heap memory footprint bound to use all of the available memory that will not induce paging. BC uses the Linux /proc/meminfo device file to compute the amount of available memory; similar functionality is available in most operating systems.

5.4 Bookmarking

When it cannot find a discardable page, BC needs an appropriate *victim* page. The page the virtual memory manager schedules for eviction is usually a good victim, since the virtual memory manager approximates LRU order and we can expect that this page is unlikely to be used soon. Not all pages are appropriate victims, however. Despite their usage patterns, BC often touches nursery pages and pages containing needed metadata (such as superpage headers, which we discuss below). Therefore, BC will not nominate these pages even if they are scheduled for eviction.

Upon receiving a signal that the virtual memory manager scheduled a page for eviction, BC touches the page to prevent its eviction. This touching also raises the page in the virtual memory manager's LRU ordering and thereby leads to a new victim page being scheduled for eviction. By touching pages, BC generates alternate victims when inappropriate pages are scheduled for eviction. Additionally, touching these pages moves to the top of the LRU queue and thus provides BC time to collect the heap and, hopefully, obviate the need for this eviction by discarding pages emptied during the collection.

If collecting the heap does not yield enough discardable pages, BC scans the pages that had been scheduled for eviction and are appropriate victims. BC processes each object on this page in turn. In examining each object, BC looks for any references to objects on other superpages. If it finds such a reference, BC marks the superpage in a on which it found the target object in a bit array and then sets the bookmark in the target object. These bookmarks (a single bit in the status word in the target object's header) serve as a secondary set of root references, recording a conservative set of references from the evicted pages without needing to fault the page into memory.

BC also sets a bookmark in each reachable object on the page being processed for eviction. Setting these bookmarks serves two purposes. If the page is eventually reloaded, BC uses these bookmarks to determine where the reachable objects exist on the page. After this page has been nominated for eviction, we may evict other pages containing references to objects on the current page. Without BC bookmarking the reachable objects prior to their eviction, we would then have to reload the page simply to set this bookmark. By instead setting these bookmarks when we process each the object, BC will not need to reload the page.

Once BC finishes scanning all of the page's objects, it goes through the bit array recording the superpages in which it just set a bookmark. For each of these superpages, BC increments their *incoming bookmark counter*. As we discuss in Section 5.4.2, BC uses this counter (which records the number of evicted pages containing one or more references to an object on the superpage) to release a superpage's bookmarks.

BC stores superpage metadata in each superpage's header. This placement permits accessing this metadata in constant-time by bit-masking an address. This access time is important, because the metadata is needed for both allocation and collection. While this metadata could be stored off to the side instead, we would similarly be unable to evict this large pool of metadata pages, including information for superpages that have not been allocated. While storing this metadata in the superpage header prevents BC from evicting the first page in each superpage, it also reduces memory overhead and simplifies memory layout, along with the corresponding page eviction and page reloading code.

Keeping the superpage headers resident means that incrementing the incoming bookmark counters cannot trigger a page fault. Setting the bookmarks could trigger a page fault, however, if BC bookmarks an object residing on an evicted page. BC therefore avoids setting bookmarks for evicted objects by conservatively bookmarking all objects on a page before it is evicted. This solution ensures that bookmarks recreate all of the evicted references without triggering any page faults.

After setting these bookmarks the victim page can be evicted. Having just been touched, however, it is no longer scheduled for eviction. BC thus communicates with the virtual memory manager one more time, informing the virtual memory manager that the page should be evicted next. While the stock Linux virtual memory manager does not provide support for this operation, BC uses a new system call provided by our extended kernel, `vm_relinquish`. This call allows user processes to surrender a list of pages voluntarily. The virtual memory manager places these *relinquished* pages at the end of the inactive queue where it selects pages to swap out to disk.

A race condition could arise if the relinquished pages are touched before the virtual memory manager evicts them. While BC would have processed these pages for bookmarks, they would not be evicted and BC would not be informed that they are memory-resident. To eliminate this possibility, BC ends each scan by disabling access to the page via the `mprotect` system call. When the page is next accessed, the virtual memory manager notifies BC. Once notified, BC can not only re-enable access to the page, but also remove the bookmarks as we discuss in Section 5.4.2.

5.4.1 Collection After Bookmarking

When the heap is not entirely resident in memory, BC starts the marking phase of each whole-heap collection by scanning the heap for memory-resident bookmarked objects. While this scan is expensive, the orders of magnitude difference in the cost of accessing main memory versus the hard disk makes this tradeoff worthwhile. BC reduces this cost further by scanning for bookmarked objects only on superpages with a nonzero incoming bookmark count. During this scan, BC marks and processes bookmarked objects as if they were root-referenced. BC thus reconstructs all the references from evicted objects during this scan. BC now follows its usual marking phase, except that it can now safely ignore references to evicted objects. After marking completes, all reachable objects are either marked or evicted. A sweep of the memory-resident pages completes this collection.

When BC compacts the heap, slightly more processing is required. During marking, BC updates the object counts for each size class to reserve space for every object that may reside on an evicted page. BC then preferentially selects superpages containing bookmarked objects or evicted pages as compaction targets and selects other superpages only as needed. While this would not work if there exists a bookmarked object on every superpage, we have not seen this happen in practice. BC scans the heap to process bookmarked objects once more at the start of compaction. Because the bookmarked objects reside on target superpages, BC will not move them and does not need to update the (evicted) reference to these bookmarked objects.

5.4.2 Clearing Bookmarks

While BC largely eliminates page faults caused by the garbage collector, it cannot prevent mutator page faults. As described in Section 5.4, the virtual memory manager notifies BC whenever the mutator accesses an evicted (and protected) page. Upon receiving this notification, BC processes the page again, but this time looks to clear bookmarks. BC scans a reloaded page's objects and decrements the incoming bookmark counter of the referenced objects' superpages. When a superpage's counter drops to zero, its objects are referenced only by objects in main memory. BC therefore clears the now-unnecessary bookmarks on that superpage. If the reloaded page's superpage also has a zero incoming bookmark count, BC clears the bookmarks conservatively set when the page was evicted. If the reloaded page's superpage does not have a zero incoming bookmark count, then objects on evicted pages contain references to objects on the reloaded page and the bookmarks must be preserved.

5.4.3 Complications

Previously, we described the virtual memory manager as if it schedules evictions on a page-by-page basis to maintain a set number of pages in memory. In fact, the virtual memory manager schedules page eviction in large batches to hide disk latency. As a result, the size of available memory fluctuates wildly. The virtual memory manager also operates asynchronously from the collector, meaning that it can run ahead and evict a page before BC can process it or even before BC gets scheduled to run.

BC avoids this problem using two complementary techniques. First, it maintains a store of discardable pages and triggers a collection when the only empty pages in the heap are the pages in this store. When pages are scheduled for eviction during a collection, BC keeps the useful page in memory by discarding a page from this reserve. Once the collection completes, BC replenishes the reserve from the newly emptied pages. If the collection does not empty enough pages, however, BC makes space in physical memory by preventively evicting pages. For this *preventive bookmarking*, BC examines, processes, and evicts pages that had been scheduled for eviction and are appropriate pages to evict. In particular, BC evicts enough of these pages to create sufficient space in available memory for BC to refill its page reserve.

BC employs a second technique to insure that it processes pages before the virtual memory manager evicts them. When BC tries finding a discardable page, it may discard more than one page. BC discards all contiguous empty pages recorded on the same word in BC's bit arrays as the first discardable page BC finds. While this can reduce the *actual* footprint quickly, BC reduces its footprint *bound* by only the single page. This aggressiveness prevents BC from being swamped by eviction notifications. By often discarding extra pages, the virtual memory manager will not need to schedule as many pages for eviction and BC will not need to process as many of these notifications.

5.5 Preserving Completeness

The key principle behind bookmarking collection is that the garbage collector must avoid touching evicted pages. A natural question is whether *complete* garbage collection, which is guaranteed to reclaim all garbage objects is possible without touching *any* evicted pages.

In general, it is impossible to perform complete garbage collection without traversing non-resident objects. Consider the case when we must evict a page full of one-word objects that all refer to objects on other pages. A lossless summary of reachability information for this page requires as much memory as the objects themselves, for which we no longer have room.

This limitation means that, in the absence of other information about reachability, we must rely on conservative summaries of object connectivity. Possible summaries include summarizing references on a page-by-page basis, compressing references in memory, or maintaining reference counts. Note that we already use the smallest summarization heuristic: a single bit previously available in the object's header. When the bookmark bit is set, BC treats the object as the target of at least one reference from an evicted page.

While this summary information is "free", bookmarking imposes a potential space cost. BC must select all superpages containing bookmarked objects or evicted pages as targets and so cannot always minimize the size of the heap through its compacting collection. Because BC treats all bookmarked objects as reachable (even ones that may be garbage), BC is further limited in how far it can shrink the heap.

Despite these apparent costs, bookmarking does not substantially increase the minimum heap size that BC requires. Section 5.7.3 shows that, even in the face of megabytes of evicted pages, BC continues running in very tight heap sizes. Such tight heaps mean that BC collects the heap more frequently, but the time needed for these collections is still much less than the cost of the page faults that would otherwise occur.

In the event that the heap is exhausted, BC preserves completeness by performing a whole heap garbage collection (including evicted pages). BC performs this collection in a virtual memory oblivious manner and does not check for bookmarks nor does it ignore references to evicted objects. BC then removes bookmarks lazily when pages are swept following the collection. Note that this worst-case situation for bookmarking collection (which we have yet to observe) is the common case for many garbage collectors. We show in Section 5.7 that BC's approach effectively eliminates collector-induced paging.

5.6 BC Implementation Details

We implemented the bookmarking collector using MMTk [26] and Jikes RVM version 2.3.2 [11, 12]. When implementing the BC algorithm within MMTk and Jikes RVM, we make three minor modifications. One optimization allows Jikes RVM to compute an object's hash code based upon its address. While address-based hashing provides many benefits, it also requires that an object which has used this hash code grow by one word to remember this hash code the first time the object is copied. This ultimately means the object counts BC uses to compute the number of target superpages depends on the superpages selected as targets. This address-based hashing complicates BC's selecting target superpages and must be disabled.

Two other optimizations within Jikes RVM concern the placement of object headers. While Jikes RVM places object headers for scalars at the end of the object, it places object headers for arrays at the start of an object.[3] This placement is useful for optimizing null pointer checks [12], but to find an object's header, BC must first determine whether the object is a scalar or an array. Since this information is normally stored in the object header, BC would be unable to perform its object processing. We solve this problem by further segmenting our allocation to allow superpages to hold either only scalars or only

[3]More recent releases of Jikes RVM places all headers at the beginning of the object.

arrays. BC then stores the type of objects contained on the superpage within each superpage header. Using the type and size class information in the superpage header, BC can quickly locate all objects on a page. Jikes RVM also aligns objects containing longs or doubles differently than it aligns other objects. While these different alignments improve performance, it would also require BC to know the type of an object to find the object's header. BC therefore aligns all objects as if they contain a long or double.

While we needed to remove these optimizations from BC builds, we did not want to bias our results by removing them from builds that would benefit from their presence. Therefore, all builds but BC include these optimizations.

5.6.1 Kernel Support

The bookmarking collector improves garbage collection paging performance primarily by cooperating with the virtual memory manager. We extended the Linux kernel to enable this cooperative garbage collection. This extension consists of changes or additions to approximately six hundred lines of code (excluding comments), as measured by SLOCcount [108].

The modifications are on top of the 2.4.20 Linux kernel. This kernel uses an approximate global LRU replacement algorithm. User pages are kept in either the active list (managed by the clock algorithm) or the inactive list (a FIFO queue). Pages are evicted from the end of the inactive list.

When the application begins, it registers itself with the operating system so that it will receive notification of paging events. The virtual memory manager then notifies the runtime system just before any page is scheduled for eviction from the inactive list (specifically, whenever the page table entry is unmapped). The signal sent includes the address of the relevant page.

To maintain information about process ownership of pages, we applied Scott Kaplan's lightweight version of Rik van Riel's reverse mapping patch [73, 104]. This patch allows

Benchmark statistics			
Benchmark	Total Bytes Alloc	Min. Heap	Bytes Alloc/Min. Heap
SPECjvm98			
_201_compress	109,190,172	16,777,216	6.51
_202_jess	267,602,628	12,582,912	21.27
_205_raytrace	92,381,448	14,680,064	6.29
_209_db	61,216,580	19,922,944	3.07
_213_javac	181,468,984	19,922,944	9.11
_228_jack	250,486,124	11,534,336	21.72
DaCapo			
ipsixql	350,889,840	11,534,336	30.42
jython	770,632,824	11,534,336	66.81
SPECjbb2000			
pseudoJBB	233,172,290	35,651,584	6.54

Table 5.2. Memory usage statistics for the benchmarks used to evaluate bookmarking collection. Because of differences in the processor used and compiler methodology, these values differ from those in Table 4.1

the kernel to determine the owning process of pages currently in physical memory. We extended this reverse mapping to include pages on the swap partition (evicted to disk). While BC does not use this feature, the kernel also extends the `mincore` call, which returns if a page is in physical memory. While not needed for BC, these extensions account for almost one-third of the lines of code that we added to the kernel.

BC needs timely updates of changing memory pressure from the virtual memory manager. To ensure the timeliness of this communication, our notifications use Linux real-time signals. Real-time signals in the Linux kernel are queueable [2]. Unlike other notification methods, these signals cannot be lost due to other process activity and will always be delivered.

5.7 BC Results

We first evaluated BC by comparing it to five garbage collectors included with MMTk and Jikes RVM: MarkSweep, SemiSpace, GenCopy, GenMS, and CopyMS.[4] More detailed descriptions of these collectors can be found in Section 4.3.

Table 5.2 describes the benchmarks we use for these analyses. Because these experiments are run on a different processor and use a different methodology, the total bytes allocated and minimum heap size in this table differ from those in Table 4.1. In addition to the benchmarks described in Section 4.3, we use jython, an additional benchmark from the DaCapo suite. It is the unfortunate situation that few publicly-available benchmarks are of a size sufficient to provide a good test of paging [84]. For analysis of paging performance, we compare execution and pause time results for runs of pseudoJBB on our extended Linux kernel. Besides being of a size sufficient to adequately test paging performance, pseudoJBB is widely considered to be the most representative of a server workload [8, 56, 84].

After these analyses, we compare BC with several explicit memory managers using the oracular memory management system described in Chapter 4. More detailed descriptions of the explicit memory managers and benchmarks we used can be found in Section 4.3.

5.7.1 Methodology

We perform all measurements on a 1.6GHz Pentium M Linux machine with 1GB of RAM and 2GB of local swap space. This processor includes a 32KB L1 data cache and a 1MB L2 cache. We report the mean and variance of five runs with the system in single-user mode and the network disabled. A detailed listing of the architecture of our experimental platform can be in Table 5.3.

[4]We also wish to examine MarkCopy and MC^2 in these experiments, but could not find stable implementations for the version of Jikes RVM we are using. We did include measurements of GenRC within our paging experiments, but were unable to get reliable results from these runs. While this behavior was repeatable, we could not find a reason explaining it.

Pentium M system	
Memory hierarchy	
L1, I-cache	32KB, 8-way associative, 3 cycle latency
L1, D-cache	32KB, 8-way associative, 3 cycle latency
L2 (unified)	1024KB, 8-way associative, 5 cycle latency
	All caches have 64 byte lines
Main memory	
RAM	12+ cycles (7.5ns)

Table 5.3. The memory timing parameters for the machine used to test the bookmarking collector. This machine contains a 1.6GHz Pentium M processor.

Our experimental methodology differs from that in Section 4.3. We run two iterations of each benchmark, but report results only from the second iteration. This two iteration process is useful as Eeckhout et al. have shown that compilation can have a more significant impact than is found in real systems [57]. Following Bacon et al. [16], and duplicating the environment typically found in servers, the first iteration optimizes all of the code. Because compilation allocates into the heap, we allow the heap size to change during this iteration. Once the first iteration completes, Jikes RVM performs a whole heap collection and removes compilation objects and any data remaining from the first run. We then turn off Jikes RVM's adaptive system to prevent any further code from being compiled and measure the second iteration of the benchmark. Using this methodology, we compare the performance of collectors in an environment similar to that found in a large server.

To examine the effects of memory pressure on garbage collection paging behavior, we simulate increasing memory pressure caused by another application starting up or increasing its working set size. We begin these experiments by making available only enough memory for Jikes RVM to complete the compilation phase without any memory pressure. We then use an external process that we call signalmem to generate memory pressure. Jikes RVM notifies signalmem after completing the first iteration of a benchmark. Once alerted, signalmem uses `mmap` to allocate a large array, touches these pages, and then pins them in memory using `mlock`. The initial amount of memory pinned, total memory pinned, and rate of growth of pinned memory are specified via command-line parameters. With signalmem we perform repeatable measurements under memory pressure while limiting the disruption to the CPU load. This approach also allows us to compare the effects of different levels of memory pressure and a variety of page eviction rates.

5.7.2 Performance Without Memory Pressure

While the key goal of bookmarking collection is to avoid paging, BC would not be practical if it did not provide competitive throughput in the absence of memory pressure. We therefore first compare the performance of BC with that of the other collectors when there is no memory pressure. Figure 5.2 summarizes the results when there is sufficient memory to run the benchmarks without any external memory pressure. This graph presents the geometric mean increase in execution time relative to BC across of heap sizes relative to the smallest heap size needed by BC. Figure 5.2 includes all the garbage collectors we tested and results for "BC w/o Compaction" which differs from the normal bookmarking collector only in being unable to perform compactions. BC without compaction is therefore like GenMS in being unable to compact the heap, but BC without compaction uses BC's superpages, size classes, and collection routines rather than those included with MMTk.

As expected, BC is closest in performance to GenMS, although BC occasionally runs in a smaller heap size. Both collectors perform nursery collection and have a segregated-fit mark-sweep mature space, and so both show similar behavior at large heap sizes without external memory pressure. At the largest heap size (where heap compaction is rarely used), the two collectors provide nearly identical throughput. At smaller heap sizes, BC is able to outperform GenMS slightly. More importantly, BC's compaction allows it to complete executing some benchmarks in heaps to small for GenMS.

Table 5.4 shows the average number of collections and compactions BC performs. The results are shown at each heap size relative to the minimum needed to run the benchmark. BC only performs compactions at the smallest heap size and then only needs to compact the

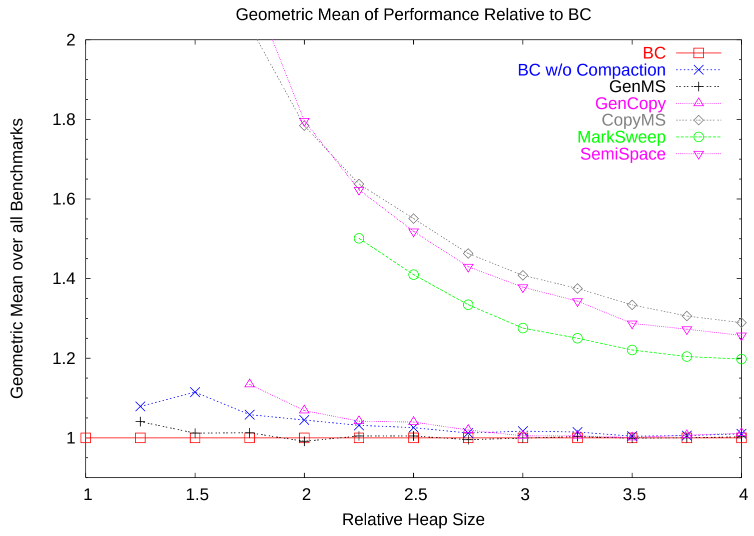


Figure 5.2. Geometric mean of execution time across all benchmarks relative to BC and absent any external memory pressure. All of the generational collectors use the Appel-style flexible nursery that Jikes RVM enables by default. At the smaller heap sizes, heap compaction allows BC to require less space while providing the best performance. Compaction is not needed at larger heap sizes, but BC continues to provide high performance.

Geo. Mean of Number of Compactions and Collections for BC		
Relative Heap Size	Mean Compactions	Mean Collections
1.00	32.89	305.01
1.25	0.00	112.37
1.50	0.00	58.66
1.75	0.00	45.16
2.00	0.00	30.29
2.25	0.00	25.87
2.50	0.00	20.28
2.75	0.00	17.52
3.00	0.00	14.70
3.25	0.00	11.50
3.50	0.00	9.72
3.75	0.00	8.45
4.00	0.00	7.29

Table 5.4. Mean number of compactions and collections across all benchmarks that BC performs at each heap size relative to the collector minimum. While it rarely compacts the heap, this compaction can improve BC's performance.

heap for one-half of the benchmarks (_213_javac, _228_jack, ipsixql, jython, and pseudoJBB). When it does compact the heap, the number of compactions ranges from a single compaction for _228_jack, but an average of 205.5 compactions when executing ipsixql at the smallest possible heap size. While not critical in most heaps, compacting the heap can have a substantial impact in the tightest heap sizes.

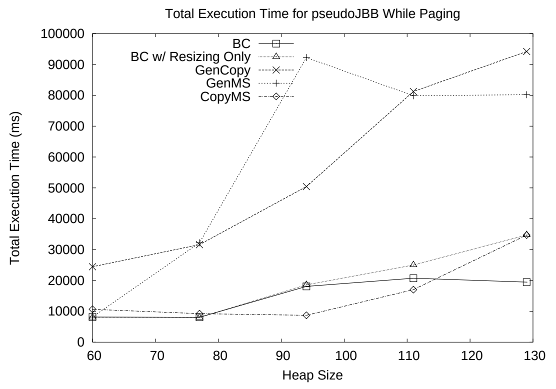
The next best collector is GenCopy, which runs as fast as BC at the largest heap sizes but averages 7% slower at heaps as large as twice the minimum needed heap size. The fact that GenCopy generally does not exceed BC's performance suggests that BC's segregated size classes do not significantly impact locality. Unsurprisingly, BC performs better than the single-generation collectors. At the largest heap size, MarkSweep averages a 20% and CopyMS a 29% slowdown. No collector at any heap size performs better on average than BC, demonstrating that BC provides good throughput when memory pressure is low.

5.7.3 Performance Under Memory Pressure

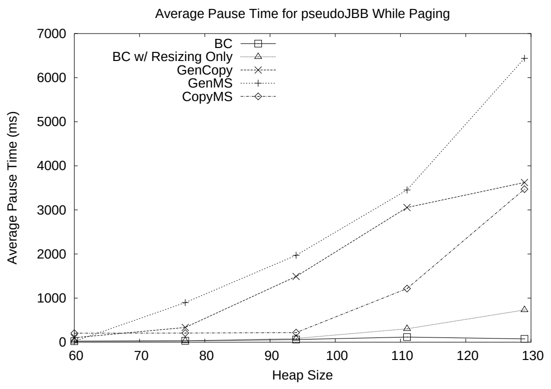
We evaluate the impact of memory pressure using three sets of experiments. We first measure the effect of *steady memory pressure*. Next we examine how *dynamically growing memory pressure*, such as that created by the creation of another process or a rapid increase in demand effects the collectors. Finally, we run *multiple JVMs simultaneously*. As discussed above, we use the pseudoJBB benchmark for all these experiments.

5.7.3.1 Steady Memory Pressure

To examine the effects of running under steady memory pressure, we run the first iteration of the benchmark with enough memory to allow all collectors to run without experiencing any memory pressure. We then use signalmem to decrease available memory to 60% the heap size and perform the timed benchmark iteration. Results of these experiments are shown in Figure 5.3. Note that we do not show results for MarkSweep or SemiSpace in these graphs, because runs with these collectors take orders of magnitude longer to complete.



(a) Execution time running pseudoJBB under steady memory pressure



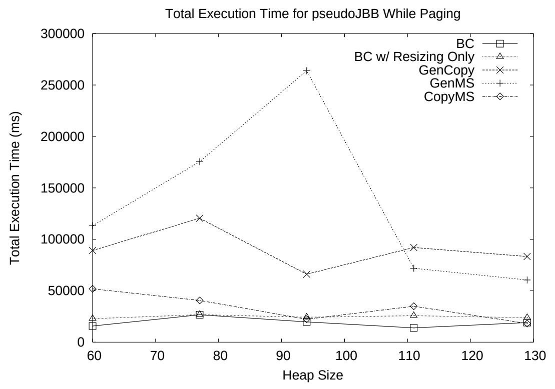
(b) Average GC pause time running pseudoJBB under steady memory pressure

Figure 5.3. Steady memory pressure (increasing from left to right), where available memory is sufficient to hold only 40% of the heap. As the heap becomes tighter, BC runs 7 to 8 times faster than GenMS and in less than half the time needed by CopyMS. Comparing BC and the BC variant that discards empty pages and shrinks the heap but cannot set or use bookmarks shows that bookmarking can yield higher throughputs and shorter pause times than simply resizing the heap.

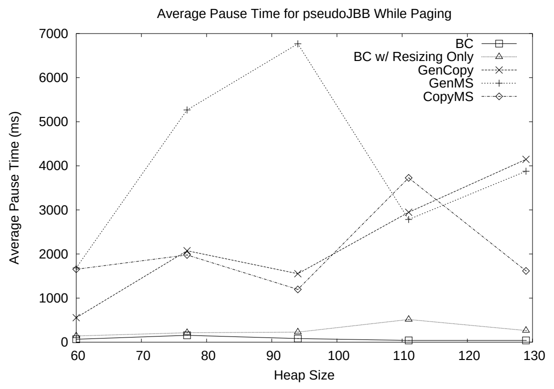
Figure 5.3 shows that under steady memory pressure, BC consistently outperforms all other generational collectors and most of the whole-heap collectors. While SemiSpace performs very poorly at the largest heap sizes, it does exhibit better throughput at the 80-95MB heap sizes. In this range of smaller heap sizes CopyMS outperforms BC also, but CopyMS ultimately runs twice as slow as BC at the 130MB heap size. At this larger heap size, GenMS has an average pause time of 3 seconds or 30 times longer of the bookmarking collector. To test CopyMS's behavior under greater memory pressure, we also measure the effect of removing memory equal to 70% of the heap size. While BC's performance remains largely unchanged under these more stressful conditions, CopyMS now takes over an hour to execute pseudoJBB. These results are shown in Figure 5.4.

5.7.3.2 Dynamic Memory Pressure

When evaluating the impact of a spike in memory pressure, we again run the first benchmark iteration with memory sufficient to run each of the collectors without memory pres-



(a) Execution time running pseudoJBB under steady memory pressure



(b) Average GC pause time running pseudoJBB under steady memory pressure

Figure 5.4. Steady memory pressure (increasing from left to right), where available memory is sufficient to hold only 30% of the heap. Unlike other collectors, BC's throughput and pause times stay relatively constant at all heap sizes.

sure. signalmem allocates 30MB at the start of the second iteration of pseudoJBB and then locks up an additional 1MB of available memory every 100ms until reaching the target memory size. The results of these experiments can be seen in Figure 5.5.

These figures again show that under memory pressure BC significantly outperforms all of the other collectors both in throughput and pause time. Note that, as with the previous paging experiments, we do not present SemiSpace and MarkSweep because of the dramatic time dilation it suffers.

Because the mark-sweep based collectors do not perform compaction, objects become spread out over a range of pages. After heap pages are evicted, collectors will need to visit these pages. This triggers a cascade of page faults and orders-of-magnitude increases in execution time. For instance, at the highest level of dynamic memory pressure in Figure 5.5 and Figure 5.6, GenMS's average garbage collection *pause* takes nearly 10 seconds — longer than the collector executes pseudoJBB when there is no memory pressure. Even when collections are relatively rare, spreading objects across a large number of pages can decrease throughput by increasing the number of mutator faults.

Compacting objects onto fewer pages can reduce faults, but both Figures 5.5 and 5.6 shows that the copying collectors also suffering from paging effects. For example, Figure 5.5(b) shows that increasing memory pressure causes an order-of-magnitude increase in GenCopy's execution time. Paging also increases GenCopy's average GC pause to several seconds, while BC's pause times remain largely unchanged.

The collector that is closest in execution time to BC in Figure 5.6 is CopyMS. While this collector performs well at low to moderate memory pressure, it performs far worse under both no memory pressure and severe memory pressure. This effect is due to pseudoJBB's allocation behavior. pseudoJBB initially allocates a few immortal objects and then allocates short-lived objects. While CopyMS reserves heap space into which it copies the survivors of a collection, little of this space is used. Between collections, LRU ordering causes nursery pages filled with dead object to be evicted. CopyMS's mark-sweep mature object space allows good heap utilization and few collections. But, while this heap organization delays paging, it cannot prevent it.

To tease apart the impact that various strategies have on paging, we also analyzed variants of the bookmarking collector and other generational collectors.

Bounded nurseries: Fixed-size nurseries can achieve the same throughput as variable-sized nurseries, but are believed to incur lower paging costs [67]. Figure 5.5(c) presents execution times for variants of the generational collectors using bounded (4MB) nurseries. The graph shows that bounded nursery variants of the other collectors can reduce the overall heap memory footprint, but perform as poorly as the variable-sized generational collectors once paging.

Bookmarking vs. resizing the heap: Tables 5.5 and 5.6 show the average count of pages BC discards, relinquishes, and reloads at each available memory size for both the 77MB and 94MB heap sizes. When memory pressure is low, BC's strategy of discarding empty pages is sufficient to avoid paging. As memory pressure increases, BC becomes unable to keep the entire heap in physical memory and must relinquish

BC with a 77MB Heap			
Available Memory	Discarded	Relinquished	Reloaded
212	0	0.0	0.0
163	4110.4	45.0	2.4
153	7529.4	0.0	0.0
143	11357.2	0.0	0.0
133	14025.4	0.8	0.2
123	17115.2	76.4	5.4
113	18619.6	129.6	4.6
103	2006.0	295.2	72.8
93	20273.4	338.4	70.6

Table 5.5. Average number of heap pages discarded, relinquished, and reloaded by BC running with a 77MB heap at different levels of available memory. As this table shows, BC's use of LRU ordering to select pages for eviction prevents the reloading of pages.

BC with a 94MB Heap			
Available Memory	Discarded	Relinquished	Reloaded
212	0	0	0
163	21004.4	133.0	20.2
153	21694.0	149.8	8.2
143	22571.4	180.0	19.4
133	24885.6	286.2	12.2
123	27085.6	231.4	18.4
113	30665.2	500.8	29.0
103	35582.5	648.8	229.0

Table 5.6. Average number of heap pages discarded, relinquished, and reloaded by BC running with a 94MB heap at different levels of available memory.

pages. When this happens, the variant of BC that only discards pages (shown as in these graphs as "BC w/resizing only") takes up to 10 times longer to run than when BC can both discard pages and set bookmarks. The results shown in Figures 5.3 and 5.5(a) demonstrate the importance of bookmarking in achieving high throughput and low pause times under moderate or severe memory pressure.

We next present *mutator utilization* curves [41]. Mutator utilization is the proportion of time that the mutator runs during a given time window. In other words, it is the amount of time spent running the application rather than collecting the heap. We adopt the methodology of Sachindran et al. and present *bounded mutator utilization*, or BMU [89]. The BMU for a given window size is the minimum mutator utilization for all windows of that size or greater.

Figure 5.7 shows BMU curves for the dynamic memory pressure experiments discussed above. While MarkSweep and both variants of BC perform very well with moderate memory pressure (runs ending with 143MB of available memory, Figure 5.7(b)), all of the other collectors exhibit poor mutator utilization. In particular, GenMS requires window sizes orders of magnitude larger than BC's running time before the mutator makes any progress.

Figure 5.7(c) shows that under severe memory pressure, the full bookmarking collector far outstrips all other collectors, with the mutator running almost 90% of the time. In fact, this figure shows that every other collector has at least one collection that takes longer than it takes BC to execute the entire program. The two next best collectors, the non-bookmarking variant of BC (labeled "BC w/resizing only") and CopyMS, need over 40 seconds to execute pseudoJBB half of which is spent collecting the heap. Interestingly, MarkSweep now exhibits the worst utilization, spending 7.5 *minutes* in garbage collection and another 2.5 minutes running the program.

5.7.3.3 Multiple JVMs

Finally, we examine a scenario with two JVMs executing simultaneously. For this experiment, we start two instances of Jikes RVM running pseudoJBB and measure the garbage collection pause times and total elapsed time from when the jobs are started until the later instance completes. These experiments cannot employ the "compile-and-reset" methodology because compiling the entire application generates too much paging traffic and we cannot "reset" the virtual memory manager's history. Instead, we employ the Replay compilation methodology [68, 89, 113] discussed in Section 4.1.1.1.

Figure 5.8 shows the results of executing two instances of pseudoJBB, each with a 77MB heap. While BC performs the best, total elapsed time can be misleading: for all of the other collectors, paging effectively serializes the benchmark runs. This serialization means that the first instance of pseudoJBB begins and runs to completion before the second instance is able to start. The average pause time numbers are thus more revealing. As available memory shrinks, bookmarking shows a gradual increase in average pause times. At the lowest amount of available memory, BC exhibits pause times averaging about 380ms, while average pause times for CopyMS, the next best collector, show a nearly eightfold increase.

5.8 Related Work

There are certain garbage collectors and garbage collection algorithms that are directly related to the bookmarking collector and we discuss them here.

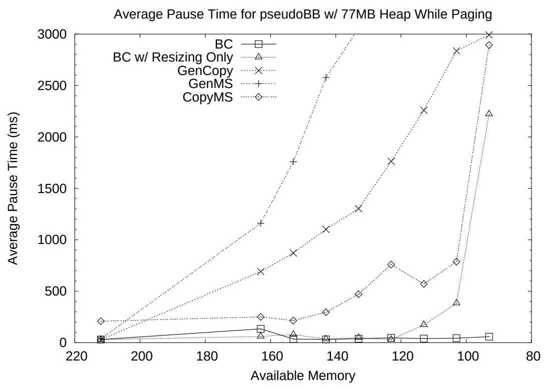
Like bookmarking collection, distributed garbage collection algorithms also treat certain references (those from remote systems) as *secondary roots*.[5] Distributed GC algorithms typically employ either reference counting or listing. Unlike bookmarking, both of these require substantial additional memory and updates on every pointer store, while BC only updates bookmarks when pages are evicted or made resident (relatively rare events).

Bookmarking collection's use of segregated size classes for compaction is similar to the organization used by Bacon et al.'s Metronome collector [15, 16]. Unlike the Metronome, BC uses segregated size classes for mature objects only. BC's copying pass is also quite different. The Metronome sorts pages by occupancy, forwards objects by marching linearly through the pages and continues until reaching a pre-determined size, forwarding pointers later. BC instead copies objects by first marking target pages and then forwarding objects as they are discovered in a Cheney scan. This approach obviates the need for further passes and immediately brings memory consumption to the minimum level.

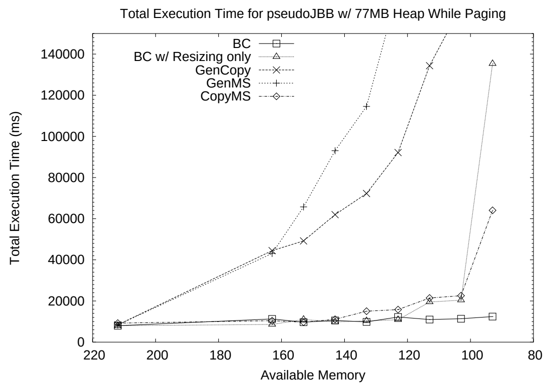
[5]See Abdullahi and Ringwood [7] and Plainfossé and Shapiro [85] for excellent recent surveys of this area.

5.9 Conclusion

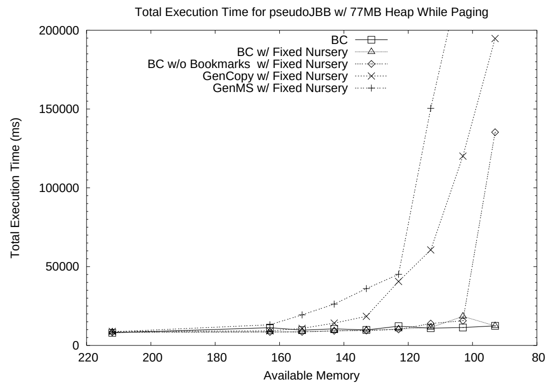
In this chapter, we present bookmarking collection, an algorithm that leverages cooperation between the garbage collector and virtual memory manager to eliminate nearly all paging caused by the garbage collector. When memory pressure is low, the bookmarking collector provides performance that generally matches or slightly exceeds that of the highest throughput collector we tested (GenMS). In the face of memory pressure, BC improves program performance by up to 5x over the next best garbage collector and reduces pause times by 45x. These results are even more dramatic when compared with GenMS. BC thus provides greater memory utilization and more robust performance than previous garbage collectors.



(a) Average GC pause times for runs of pseudoJBB

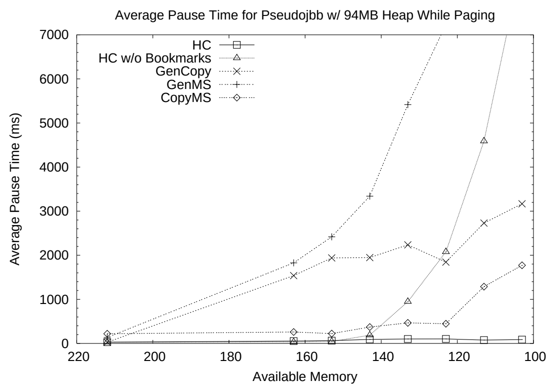


(b) Execution time running pseudoJBB using Appel-style nurseries

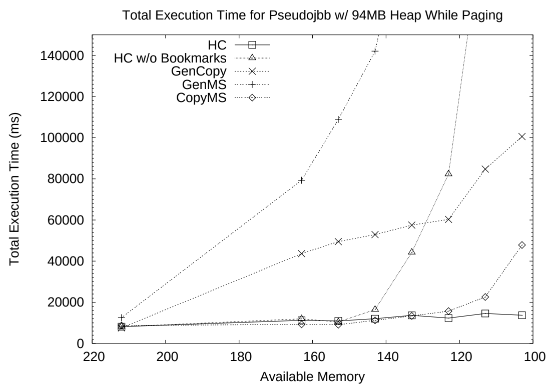


(c) Execution time running pseudoJBB using a *bounded* nursery size

Figure 5.5. Dynamic memory pressure (increasing from left to right), where the available memory shown is the amount ultimately left available. These results are for runs of pseudoJBB using a 77MB heap size. BC has a throughput up to 4x higher than the next best collector and up to 41x better than GenMS. BC's average GC pause time also remains unaffected as memory pressure was increased. While shrinking the heap improves both throughput and pause times, these graphs show that bookmarks are vital for good performance.

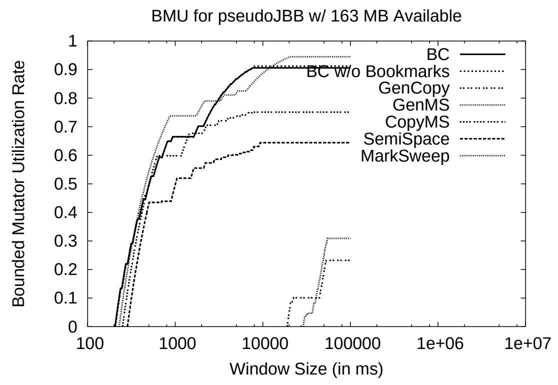


(a) Average GC pause times for runs of pseudoJBB

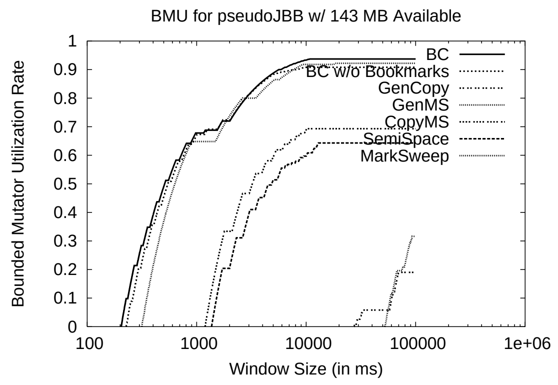


(b) Execution time running pseudoJBB using Appel-style nurseries

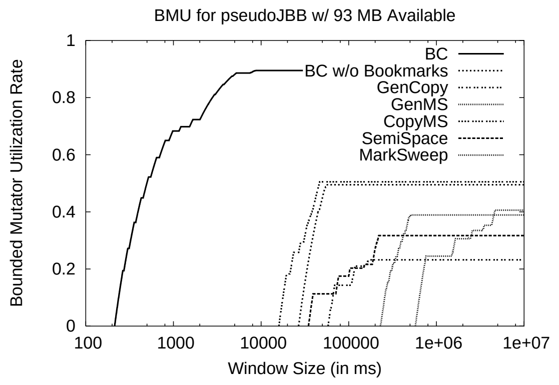
Figure 5.6. Dynamic memory pressure (increasing from left to right), where the available memory shown is the amount ultimately left available. These results are for runs of pseudoJBB using a 94MB heap size. BC has a throughput up to 2x higher than the next best collector and up to 29x better than GenMS.



(a) 163MB available: with little memory pressure, compaction is important for pause times

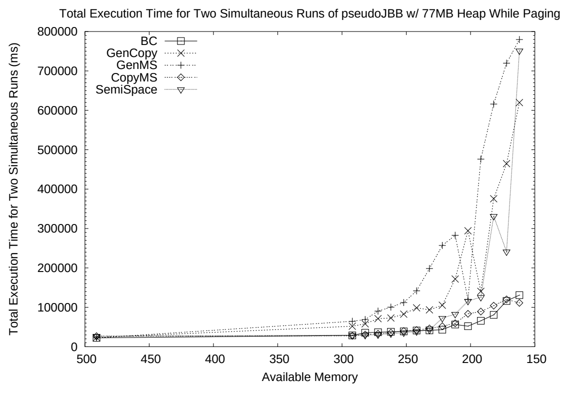


(b) 143MB available: BC is largely unaffected by moderate levels of memory pressure

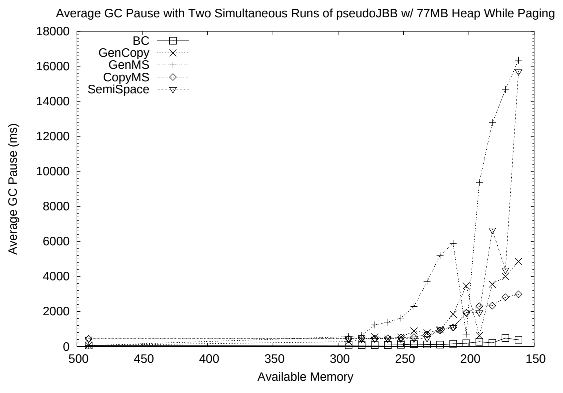


(c) 93MB available: under heavy memory pressure, bookmarks become increasingly important

Figure 5.7. Bounded mutator utilization curves (curves to the left and higher are better) for runs applying dynamic memory pressure. BC consistently provides high mutator utilization over time. As memory pressure increases (available memory shrinks), the importance of bookmarks in limiting garbage collection pause times becomes apparent.



(a) End-to-end execution time running two instances of pseudoJBB simultaneously



(b) Average GC pause time running two instances of pseudoJBB simultaneously

Figure 5.8. Execution time and pause times when running **two** simultaneous instances of pseudoJBB. The execution times can be somewhat misleading, because paging can effectively deactivates one of the instances for some collectors. Under heavy memory pressure, BC exhibits pause times around 7.5x lower than the next best collector.

CHAPTER 6
CONCLUSION

While garbage collection provides important software engineering benefits, such as reducing and eliminating memory leaks and dangling pointers, garbage collection's performance has been a hotly debated topic. This dissertation first demonstrates how to quantify garbage collection's performance. We use this performance analysis to improve garbage collection's paging performance, where our analysis shows its performance suffers most. This chapter summarizes the contributions of this thesis and discusses future directions to expand this work.

6.1 Contributions

This thesis begins by presenting the Merlin algorithm, which computes object reachability information in asymptotically optimal time. We show that the Merlin algorithm can reduce the time needed to generate program heap traces by orders-of-magnitude. We then discuss oracular memory management, which uses program heap traces to compare the performance of automatic and explicit memory managers. This thesis finds that, while the best garbage collector can match the top allocator's performance, garbage collection's good performance requires several times the physical memory to hold the heap. This thesis presents the bookmarking collector to reduce this performance brittleness. Our results show the cooperative bookmarking collector matches or exceeds the best collector's performance when not paging and improves both throughput and pause times substantially while paging.

At the outset, this thesis considers how well garbage collection performs currently and where opportunities for improvement remain. In answering this vital question, we highlight the brittleness of garbage collection's good performance and develop a new collector that makes this performance more robust.

6.2 Future Work

There are several directions in which researchers could expand this thesis in the future. We are interested in expanding this work to consider alternative strategies for selecting victim pages. While BC used pages' LRU order to select victim pages, it may be preferable to select, for example, pages that do not contain pointers and therefore could not create false garbage. By developing more sophisticated algorithms to select victim pages, we may be able to limit the number of garbage objects that remain bookmarked while not increasing the number of page faults substantially.

The metrics considered in our analysis of garbage collection performance are not exhaustive. We did not evaluate, for example, the often discussed question of what each memory manager's impact on cache locality is [44, 50, 60, 68, 77, 102, 112]. Another metric we plan on analyzing is to compare explicit memory managers' pause times and mutator utility (resulting from coalescing and reallocating memory) with those of garbage collection. Analyzing additional metrics such as these provides more data by which we can understand memory manager performance and a deeper analysis of the true costs associated with each memory manager.

This thesis considers allocating and freeing only individual objects. Custom allocation schemes that manage multiple objects, such as regions and reaps, can improve the memory performance of explicitly-managed applications dramatically [23, 61]. Expanding the oracular framework to evaluate these additional memory managers may suggest methods of improving garbage collection performance by grouping objects into pools which can be automatically reclaimed.

Another question this thesis raises is the effects other oracles would have on explicit memory management. One could investigate more moderate reclamation decisions, such as an oracle that combines trace-based and static analyses to find code points where the system can always reclaim objects. Moderate oracles provide more realistic simulations of what a programmer could accomplish and may suggest further memory manager improvements. By evaluating the potential improvements from using static analyses to explicitly free objects [43, 60], this work could also prevent researchers from considering compiler analyses that take longer than any possible improvement.

Finally while we limited our analysis to quantitative measures, garbage collection also has important qualitative effects. Another extension to this thesis would be to compare the software engineering impact of the various methods of memory management. Investigating how practitioners approach and write programs when using different memory managers could improve the methods educators use to teach novice programmers and enable language designers to select features that best match the desired memory manager.

APPENDIX
PROOF OF THE OPTIMALITY OF THE MERLIN ALGORITHM

While we discuss the design and execution of the Merlin algorithm in Chapter 3, we did not discuss or prove the algorithm's running time. We also left open the question of whether it would be possible to develop an algorithm with a faster asymptotic complexity. In this appendix, we first present a theoretical model of garbage collection that enables us to examine garbage collection abstractly. Using this theoretical foundation, we prove that the Merlin object reachability time algorithm requires time linearly proportional to the execution time of the program and, finally, show that the Merlin algorithm is asymptotically optimal.

Structures Used

It has long been understood that a program's heap memory can be viewed as a graph. Thus, our framework presents the analyses as graph theoretic problems. We first explain how our framework handles dynamically allocated objects and then propose a series of graphs that model the heap and capture the execution of the program and garbage collector.

The Heap and Program Roots

Our structures model objects dynamically allocated into the program's heap as the memory manager's reserving space in the heap for each field, and generating a map from the object's keys to the locations in memory in which they are stored. Memory for an object is typically contiguous and a field's mapping key is the offset from the start of the object to that field.

The program accesses objects in the heap through its *root set*. The framework models the root set as a mapping of keys (each representing a unique root with a referring type) to reference values. Unlike object mappings, the root set is neither bounded nor fixed. The program uses dynamically allocated objects only through the root set; objects not reachable from the root set cannot be used by the program and can be reclaimed by the system. For convenience, we do not consider oracular collectors or other garbage collection algorithms that rely upon external knowledge.

Heap State

The current state of the heap is defined by the objects and references within the heap and the program's root set. To allow for further analyses, we also include the set of objects that have been identified by the collector as unreachable. We represent the *heap state* (the current state of the heap) as a rooted, directed multi-graph. We call this multi-graph the *heap state multi-graph*, and express it as $H = (L, D, r, E)$.[1]

The set of vertices of H, $V = L \cup D$, includes a vertex for each object allocated in the heap, and a special vertex, called the root vertex and designated as r, representing the root set. The set D (for dead) contains the unreachable vertices, while L (for live) includes the remaining vertices ($L = V - D$, so $L \cap D = \emptyset$). Since roots are always reachable, r must be in L. Vertices represent only the existence of an object. This representation makes modeling garbage collectors easier by abstracting away implementation details such as objects' actual locations in the address space.

References to heap objects are represented at edges in the multi-graph. The edge multiset, E, contains an edge $(\langle v, n \rangle, o)$ if and only if object v at the field mapped to by key n contains a reference to object o. Notice that v may be r, in which case n is the key to a root that refers to o.

[1]Other elements could exist within the heap state multi-graph; we include only those elements used in this analysis.

Just as a garbage collector analyzes the heap using the root set and objects, this multi-graph can be analyzed for the relationships between the root vertex and other vertices.

Reachability

Given $H = (L, D, r, E)$, we say v refers to o (in H), written $refers_H(v, o)$, if and only if v has at least one field that refers to object o: $refers_H(v, o) = (\exists n)((\langle v, n\rangle, o) \in E)$.

Extending this definition, we say v *reaches* o (in H), written $reaches_H(v, o)$, if and only if v and o are equal or o is in the transitive closure of objects to which v refers: $reaches_H(v, o) = (v = o \vee (\exists p)(refers_H(v, p) \wedge reaches_H(p, o))$.

Given the importance of objects that the program can access from the root set, an object o is *reachable* (in H) if and only if r reaches o ($reachable_H(o) = reaches_H(r, o)$). Reachable objects may be used in the future by the program. Objects not reachable in the heap state (e.g., objects not in the transitive closure of the root set) cannot be accessed by the program and could not be reclaimed by the garbage collectors we consider at present.

A heap state multi-graph $H = (L, D, r, E)$ is *well-formed* if and only if all reachable objects are in L: $L \supseteq \{o | reachable_H(o)\}$. From here on we are concerned only with well-formed heap state multi-graphs.

Program Actions

The heap state and its corresponding graph are useful for analyzing snapshots of a program. But programs and their heaps are dynamic entities: the program mutates its heap as it runs. These changes include objects being dynamically allocated, fields of objects being updated, and objects being passed to and from functions. These changes occur throughout program execution and a dynamic memory manager must be capable of handling all of them.

Action Name	*Effect*	*Precondition*
Object Allocation	$L_{t+1} = L_t \cup \{o\}$; $E_{t+1} = E_t \cup \{(\langle r,n\rangle, o)\}$	$o \notin L_t \cup D_t \wedge$ $\neg(\exists o')((\langle r,n\rangle, o') \in E_t$
Root Creation	$E_{t+1} = E_t \cup \{(\langle r,n\rangle, o)\}$	$reachable_{H_t}(o) \wedge$ $\neg(\exists o')((\langle r,n\rangle, o') \in E_t)$
Root Deletion	$E_{t+1} = E_t - \{(\langle r,n\rangle, o)\}$	$(\langle r,n\rangle, o) \in E_t$
Heap Reference Creation	$E_{t+1} = E_t \cup \{(\langle v,n\rangle, o)\}$	$reachable_{H_t}(v) \wedge reachable_{H_t}(o) \wedge$ $\neg(\exists o')((\langle v,n\rangle, o') \in E_t)$
Heap Reference Deletion	$E_{t+1} = E_t - \{(\langle v,n\rangle, o)\}$	$reachable_{H_t}(v) \wedge (\langle v,n\rangle, o) \in E_t$
Program Termination	$E_{t+1} = E_t -$ $\{(\langle r,n\rangle, o) \mid (\langle r,n\rangle, o) \in E_t\}$	None

Table A.1. Definition of action a_t. Only changes from H_t to H_{t+1} are listed.

Interesting Program Actions

The framework is interested only in those actions that affect the heap. Though the causes of these mutations are language-specific, we group actions by their effect on the heap: object allocation, root creation, root deletion, field reference creation, field reference deletion, and program termination.[2]

We describe that the effect each action has on the heap state with respect to the heap state multi-graph at time t, $H_t = (L_t, D_t, r, E_t)$, and the multi-graph following the action, $H_{t+1} = (L_{t+1}, D_{t+1}, r, E_{t+1})$.[3] We additionally describe the preconditions necessary within H_t for the possible actions a_t. A precise mathematical definition of these effects and limitations can be found in Table A.1. The following paragraphs provide a simple description of each of the actions:

Object Allocation actions occur when an object is allocated in the heap. An object allocation action defines a new vertex to be added to the set of reachable vertices and the key value of the root that references the newly allocated vertex.

[2]Other actions could be included; these primitives are quite general and can be combined. We distinguish root and heap reference actions because many algorithms treat them differently. We specifically exclude GC behavior from program actions.

[3]Actions may also be considered functions producing H_{t+1} from H_t; we do this when it is convenient.

Root Creation actions occur when a root reference to an object is created. The root creation action defines a new edge to be added to the edge set from the root vertex to a reachable vertex.

Root Deletion actions remove an existing edge from the root vertex to a vertex in the set of reachable vertices. This action is equivalent to deleting a root reference or making a root reference null.

Heap Reference Creation actions occur when the program updates a heap object's unused field updated to reference another object. These actions add an edge to the edge set from the source vertex to the target vertex.

Heap Reference Deletion actions specify an edge in the edge set that is removed. Heap reference deletion actions occur whenever an object in the heap has a non-null field made null.

Program Termination actions occur when the program execution ends. Program termination may occur at any time and deletes the root set (removes any edge whose source is the root vertex).

In the framework, as in program execution, only existing references may be removed, each object field contains at most one reference, and only reachable objects may be involved in actions. Since the heap state multi-graph reflects the heap, the program actions mirror the changes in the heap. *By construction* programs cannot attempt actions whose preconditions are not met, so we need not consider the possibility. Further, it is easy to see that program actions preserve well-formedness of heap state multi-graphs (since they cannot remove vertices nor cause unreachable objects to become reachable).

Program History

Every program begins with the same heap. This *initial heap state* is represented by the multi-graph $H_0 = (L_0, D_0, r, E_0)$. This graph has only one vertex (the root vertex) and an empty edge multiset ($L_0 = \{r\}, D_O = \emptyset, E_0 = \emptyset$).

$$\begin{tikzcd} \overset{H_0}{(L_0, D_0, r, E_0)} \arrow[r, "a_1"] & \overset{H_1}{(L_1, D_1, r, E_1)} \arrow[r, "a_2"] & \ldots \arrow[r, "a_T"] & \overset{H_T}{(L_T, D_T, r, E_T)} \end{tikzcd}$$

Figure A.1. Illustration of the program history, showing how the heap-state multigraph mutates over the program execution

We also consider a program's heap state following program termination. *Final heap state* multi-graphs, designated $H_T = (L_T, D_T, r, E_T)$, have a vertex (in $L_T \cup D_T$) for each object allocated into the heap, plus the root vertex. Final heap state multi-graphs cannot contain edges from the root vertex, so only the root vertex is reachable in these multi-graphs.

The initial and final heap states do not contain much information about the program execution, but the entire run is necessary to analyze GC algorithms and optimizations. To record this information, the framework uses a *program history*. The program history begins with the initial heap state multi-graph, H_0, and the first program action, a_1. From this multi-graph and action, we build the successor heap state multi-graph, H_1, then add action, a_2, and so on. The program history continues up to the program termination action (a_T) and final heap state multi-graph, H_T. Figure A.1 illustrates a program history.

Since all programs start from the initial heap state, and each program action is deterministic, the actions alone are sufficient to recreate the program history. Since the program history works in a manner similar to heap traces, it is intuitive to use.

Null and Reachable Multi-Graphs

Using the program history, the framework can compute the *null heap state multi-graph* for each time step. The null heap state multi-graph at time t, $H_t^{\emptyset} = (L_t^{\emptyset}, D_t^{\emptyset}, r, E_t)$, is the heap state multi-graph where no objects have been determined to be unreachable (e.g., $D_t^{\emptyset} = \emptyset$).

Using the program history, the time when each object becomes unreachable can be determined. Given $H_t = (L_t, D_t, r, E_t)$, then the *reachable heap state multi-graph* is $H_t^* = Live(H_t) = (L_t^*, D_t^*, r, E_t)$. The reachable multi-graph specially defines L and D : $L_t^* = \{v \in L_t | reachable_{H_t}(v)\}$; $D_t^* = V_t - L_t^*$, that is, L_t^* is exactly the set of reachable vertices, and D_t^*

the remainder (the $*$ superscript is intended to suggest the optimal, i.e., smallest possible, L_t set.)

Given a heap state multi-graph H_t, we define the *reduced heap state multi-graph*, $H_t^R = Reduce(H_t)$, as the heap state multi-graph (L_t^R, D_t^R, r, E_t^R), where: $L_t^R = L_t$, $D_t^R = \emptyset$, $E_t^R = \{(\langle v, n \rangle, o) \in E_t | v \in L_t\}$. This reduction removes those vertices identified as unreachable, and any edges from these vertices, from the multi-graph. By removing edges and vertices that are known to be unnecessary, the reduced heap state multi-graph resembles the physical heap following GC.

Finally, given a heap state multi-graph H_t, we define the *reduced reachable heap state multi-graph*, H_t^-, as the heap state multi-graph $Reduce(Live(H_t))$.

The null, reachable, and reduced reachable heap state multi-graphs are similar: their reductions via $Reduce \circ Live$ are the same. Further, since program actions can manipulate only reachable objects, if we apply a_{t+1} to $H_t^\emptyset$, H_t^*, and H_t^-, we get analogous results, a fact we state precisely and prove in a moment. First we argue that if we have a well-formed heap state $H_t^\emptyset$ and corresponding action a_{t+1}, then a_{t+1} is legal for H_t^* and H_t^-.

Lemma 1 *If $H_t^\emptyset$ fulfills the preconditions of a_{t+1}, then so do H_t^* and H_t^-.*

Proof Assume H_t satisfies the preconditions of a_{t+1}. First, we note that all vertices reachable in $H_t^\emptyset$ are in L_t^* and thus in L_t^-, and that $L_t^- \cup D_t^- \subseteq L_t^* \cup D_t^* = L_t \cup D_t$. Thus if a_{t+1} is an object allocation, its precondition is satisfied in H_t^* and H_t^-. Reference creation and deletion preconditions are trivially satisfied because they mention only reachable objects and edges between reachable objects. ■

Now we state and prove a stronger relationship for program histories and their corresponding reachable and reduced reachable heap states:

Theorem 2 *If a_{t+1} takes H_t to H_{t+1}, then $a_{t+1} \circ Live$ takes H_t^* to H_{t+1}^* and $a_{t+1} \circ Live \circ Reduce$ takes H_t^- to H_{t+1}^-.*

Proof Assume $H_t = a_t(H_{t-1})$ (interpreting a_t as a function, and implying that H_{t-1} is well-formed and meets the preconditions of a_t). H^*_{t-1} differs from H_{t-1} only in the partitioning of the vertices into L and D sets, and both are well-formed. The same is true of $H_t = a_t(H_{t-1})$ and $a_t(H^*_{t-1})$. Since the edges are the same between these two graphs, their sets of reachable objects are the same, so applying *Live* to each of them gives the same result. Hence $Live(H_t) = Live(a_t(H^*_{t-1}))$. But $Live(H_t) = H^*_t$ by definition, proving the first part of the theorem.

The second part of the theorem follows if

$$Reduce(Live(a_t(H^*_{t-1}))) = Reduce(Live(a_t(Reduce(H^*_{t-1}))))$$

Let L_x and D_x be the L and D sets of H^*_{t-1}. Since *Reduce* does not affect L sets, and program actions do not add to D sets, the L sets of $a_t(H^*_{t-1})$ and $a_t(Reduce(H^*_{t-1}))$ are the same. Since a_t applies and preserves well-formedness, the L sets of $Live(a_t(H^*_{t-1}))$ and $Live(a_t(Reduce(H^*_{t-1})))$ are also the same, and consist of exactly those objects reachable in H^*_t. Finally, applying *Reduce* gives heap states with the same L set and an empty D set. Thus the L and D components of the two heap states are the same. Their r component is trivially the same (none of the functions changes r). Their E component is the same for reachable nodes, but after applying *Live* then *Reduce*, those are the *only* nodes in the graph.

■

Modeling Collector Behavior

We model garbage collector behavior by following each program action a_t with a garbage collector action g_t. Thus, we form H_t by first applying program action a_t to H_{t-1}, and then applying collector action g_t. A collector action potentially identifies some unreachable objects. In fact, we will equate g_t with the set of objects it identifies as unreachable, and define its effect on the heap state as mapping heap state $H = (L, D, r, E)$ to $(L - g_t, D \cup g_t, r, E)$,

$$\begin{tikzcd}
\overset{H_0^{\emptyset}}{(L_0^{\emptyset}, D_0^{\emptyset}, r, E_0^{\emptyset})} \arrow[r, "a_1, g_1^{\emptyset}"] \arrow[d, "Live()"'] & \overset{H_1^{\emptyset}}{(L_1^{\emptyset}, D_1^{\emptyset}, r, E_1^{\emptyset})} \arrow[r, "a_2, g_2^{\emptyset}"] \arrow[d, "Live()"'] & \cdots \arrow[r, "a_T, g_T^{\emptyset}"] & \overset{H_T^{\emptyset}}{(L_T^{\emptyset}, D_T^{\emptyset}, r, E_T^{\emptyset})} \arrow[d, "Live()"'] \\
\overset{H_0^{*}}{(L_0^{*}, D_0^{*}, r, E_0^{*})} \arrow[r, "a_1, g_1^{*}"] & \overset{H_1^{*}}{(L_1^{*}, D_1^{*}, r, E_1^{*})} \arrow[r, "a_2, g_2^{*}"] & \cdots \arrow[r, "a_T, g_T^{*}"] & \overset{H_T^{*}}{(L_T^{*}, D_T^{*}, r, E_T^{*})}
\end{tikzcd}$$

Figure A.2. Illustration of the Expanded Program History, including the null ($H^{\emptyset}$) and reachable (H^{*}) heap states.

with the precondition that $g_t \subseteq L - \{o \in L | reachable_H(o)\}$. When convenient, we will also use the notation g_t for the function that the collector action induces on heap states.

The simplest collector, which we call the *null collector*, never identifies any objects as unreachable. We write its actions as $g_t^{\emptyset}$; it induces the identity function on heap states.

The most "aggressive" collector, called a *comprehensive collector*, immediately identifies unreachable objects. We write it as g_t^{*} and the function it induces is complementary to *Live* (i.e., it identifies the unreachable objects).

Figure A.2 shows how the null ($H^{\emptyset}$) and reachable (H^{*}) heap state multi-graphs relate to null and comprehensive collector actions.

Real collectors are bounded (in what they reclaim) by the null and comprehensive collectors. Further, many collectors identify unreachable objects only occasionally. For example, they may allow a portion of the heap to fill, and identify unreachable objects in a batch only after the space is full. While we model only the end result (e.g., the set of objects identified as unreachable), in applying the framework it is easy to associate costs with collector action g_t and derive the effort taken by the collector at each time step.

Object Death Time Multi-Graph

We add to the framework the *object death time multi-graph.* This multi-graph differs from the others because it concerns only the efficiency of a collector. It is not related to the heap at any moment of the program history; rather it exists to prove the minimum information and work needed to determine the earliest time each object could be reclaimed. This multi-graph can also compare the efficiency of comprehensive collectors by analyzing the relative work needed to populate and analyze this multi-graph. Before describing the new graph, we discuss a concept upon which it relies: *final reference deletion time*.

An object's final reference deletion time is the last time at which the object has an incoming reference deleted (by a root or heap reference deletion action or at program termination). Because each object is allocated with a reference from the root set, and the program termination action removes any root references that exist, each object has a final reference deletion time. This time occurs between the object's allocation and program termination. We define the function f to map each vertex to its final reference deletion time. Given a vertex v, f is defined as: $f(v) = \max_{i<T}((\exists o,n)((\langle o,n\rangle, v) \in E_i \wedge (\langle o,n\rangle, v) \notin E_{i+1}))$. An object may have incoming references at its final reference deletion time, provided that the remaining incoming references are not deleted by a program action; in the name, "final" modifies "deletion", not "reference".

With this definition, we present our last graph structure. The object death time multi-graph is also a directed, rooted multi-graph, $F = (V, E_T, f)$, where f is the final reference deletion time function from above. The multi-graph's set of vertices V contains a vertex for each object allocated in the heap, i.e., $V = V_T - \{r\}$. The multi-graph's edge multiset, E_T, is the edge multiset of the final heap state.

As the name implies, the multi-graph determines the *death time* for each object. An object's death time is the time at which it becomes unreachable—the time the corresponding vertex would be included in a g^* action. As the following theorem shows, this time can be computed in the object death time multi-graph as each vertex's latest *reaching final reference deletion time*, the latest final reference deletion time among the vertices that reach each vertex.

Theorem 3 *The latest reaching final reference deletion time to a vertex in F is the time the corresponding object in the heap became unreachable.*

Proof Objects become unreachable only when references are removed; thus object death times occur only at actions that remove references. Since unreachable vertices cannot be involved in program actions, only final reference deletions cause objects to become unreachable: if an earlier reference deletion left an object unreachable, the object could not be involved in the later action. Therefore, object deaths occur only at final reference deletions.

Not all final reference deletions leave a vertex unreachable, however. From the definition of *reachable*, $\mathit{reachable}(v)$ is true when v is in the transitive closure of the root vertex, regardless of final reference deletion times. If v is the target of an edge from a reachable vertex, $\mathit{reachable}(v)$ is true. If v's final reference deletion time has passed, v becomes unreachable at the latest time that a referring vertex becomes unreachable. By this logic, each vertex becomes unreachable at the latest final reference deletion time for it *or any vertex that reaches it* (since the vertices reaching it may be reachable only because they are referred to by still other vertices). Thus, any object in the transitive closure set of an object at its final reference deletion time may become unreachable at that time. These transitive closure sets are defined by the final pointers—the object death time multi-graph's edge multiset.

Therefore, vertices become unreachable at the latest reaching final reference deletion time. As reachability in the heap state reflects reachability in the heap, this time is the corresponding object's death time. ∎

We note that this multi-graph is similar to H_T^* ($V = D_T^*$ and $E_T = E_T^*$) as one would expect, given its function. Using this multi-graph, we can compute object death times in asymptotically optimal time, as we show in the next section.

Merlin Algorithm Analysis

The Merlin algorithm presented in Chapter 3 computes last reachable times by performing a small amount of work at (some) program actions and delaying performing the more costly analyses. It computes each object's final reference deletion time in conjunction with program actions and analyzes the object death time multi-graph that it constructs with this information.

We now prove the asymptotic running time for the Merlin algorithm and then prove that this running time is optimal.

Merlin's Running Time

Computing which objects are reachable in the heap state is equivalent to solving the single-source/multi-sink reachability problem from the root vertex, which requires $O(L_t^R + E_t^R)$ time.

To compute object reachability times, Merlin builds and analyzes the object death time multi-graph as the program runs. Using the object death time multi-graph, finding when objects become unreachable requires comparing the reaching final reference deletion times for each vertex; this processing seems analogous to computing all of the transitive closures sets, requiring $O(V_T \cdot E_T)$ time [49, p. 766].

Solving the transitive closures determines when each object can be reclaimed, but requires more work than is needed. Object death times are the *latest* reaching final reference deletion times; to find death times, Merlin requires a single depth-first search from each vertex in reverse order of vertex final reference deletion times.

Lemma 4 *When computing object death times, every vertex and edge needs to be processed only once.*

Proof With the depth-first search, Merlin needs to process each vertex only once, because repeat visits to a vertex are computing equal or earlier reaching final reference deletion times. As only the latest time matters, processing these repeat visits will not change any object reachability times. Assume that by not processing subsequent visits, an incorrect last reachable time is computed for a vertex. For this to hold, a repeat visit must compute a later reaching final reference deletion time. But the depth-first search from the vertex with the later final reference deletion time must begin before, not after, the initial last reachable time was found. This contradiction invalidates the assumption; processing each vertex once correctly computes object reachability times. ■

With the framework, we can now prove Merlin's asymptotic running time.

Theorem 5 *Merlin requires $O(T)$ time to create the Object Death Time Multi-Graph and an additional $O(V_T + E_T)$ time to compute the last reachable times.*

Proof The Merlin algorithm takes constant time at each program action to build the object death time multi-graph. For T actions, this clearly takes $O(T)$ total time.

Given the object death time multi-graph, the time required to find all last reachable times is $O(V_T + E_T)$. The Merlin algorithm begins by sorting the vertices by their final reference deletion time. As these times must lie between 1 and T, we may use a radix sort to order the vertices; this sorting therefore taking $O(V_T)$ time. Since each vertex and edge needs to be processed only once, the depth-first searches also take $O(V_T + E_T)$ total time.[4] Therefore, the Merlin algorithm computes object reachability times in total time $O(V_T + E_T) + O(T) = O(T)$, since $O(V_T + E_T) \leq O(T)$ (there were only T actions, so one cannot have created more than $O(T)$ vertices or edges). ■

This leads to an additional theorem.

Theorem 6 *The Merlin algorithm computes object reachability times in asymptotically optimal time.*

Proof Each program action during a program's execution may define an action that is necessary in order to compute object reachability times (e.g., object allocations and program termination). Therefore any algorithm computing object reachability times requires at least $\Omega(T)$ time. Since the Merlin algorithm completes in this time, its $\Theta(V_T + E_T + T) = \Theta(T)$ time is optimal. To build the object death time multi-graph, the vertices, final pointers, and final reference deletion times are needed. To find the vertices, each object allocation action

[4]In practice, systems must first examine each field to determine if the contents are a reference or null.

must be handled. Since knowing which references are final pointers and which reference deletions are final is not computable until termination, all reference creations and deletions must be processed. This takes $\Omega(T)$ time.

Consider an object death time multi-graph where no two vertices share a final reference deletion time and an edge exists between two vertices if and only if the source vertex has an earlier final reference deletion time than the target vertex. As the reaching final reference deletion times to the vertices are the consecutive subsequences of final reference deletion times that begin with the earliest time, computing the object reachability times within this multi-graph requires completely ordering the last reachable times. This ordering requires as much time as sorting the vertices, which a radix sort accomplishes in $\Omega(V_T)$ time.

Without knowing the target of an edge, we cannot determine if the edge is on the path of the target's latest reaching final reference deletion time. If the edge is on this path, it must be processed; this cannot be determined, however, without examining the edge source and target. Therefore, every algorithm also requires $\Omega(E_T)$ time to compute last reachable times.

Combining the arguments of the previous three paragraphs, we conclude that $\Omega(T)$ time is required to build the object death multi-graph and $\Omega(V_T + E_T)$ time is needed to compute object reachability times. As the Merlin algorithm completes in this time, its $\Theta(V_T + E_T) + \Theta(T)$ time is optimal. ■

BIBLIOGRAPHY

[1] J2SE 1.5.0 documentation - garbage collector ergonomics. Available at `http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html`.

[2] Linux Programmer's Manual: SIGNAL(7). Linux man pages.

[3] Novell Documentation: NetWare 6 - Optimizing Garbage Collection. Available at `http://www.novell.com/documentation/index.html`.

[4] Technical white paper - BEA WebLogic JRockit: Java for the enterprise. Available at `http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf`.

[5] Hardware — G5 performance programming, Dec. 2003. Available at `http://developer.apple.com/hardware/ve/g5.html`.

[6] Technical note TN2087: PowerPC G5 performance primer, Sept. 2003. Available at `http://developer.apple.com/technote/tn/tn2087.html`.

[7] Abdullahi, Saleh E., and Ringwood, Graem A. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys 30*, 3 (Sept. 1998), 330–373.

[8] Adl-Tabatabai, Ali-Reza, Hudson, Richard L., Serrano, Mauricio J., and Subramoney, Sreenivas. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Washington, DC, June 2004), pp. 267–276.

[9] Agesen, Ole, Detlefs, David L., and Moss, J. Eliot B. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, QC, Canada, June 1998), ACM, pp. 269–279.

[10] Alonso, Rafael, and Appel, Andrew W. An advisor for flexible working sets. In *Proceedings of the 1990 Joint International Conference on Measurement and Modeling of Computer Systems* (Boulder, CO, May 1990), pp. 153–162.

[11] Alpern, Bowen, Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, Perry, Choi, J-D, Cocchi, Anthony, Fink, Stephen J., Grove, David, Hind, Michael, Hummel, S. F., Lieber, D., Litvinov, V., Mergen, Mark F., Ngo, Ton, Sarkar, Vivek, Serrano, M. J., Shepherd, Janice C., Smith, Stephen, Sreedhar, V. C., Srinivasan, H., and Whaley, J. The Jalepeño virtual machine. *IBM Systems Journal 39*, 1 (Feb. 2000).

[12] Alpern, Bowen, Attanasio, C. R., Barton, John J., Cocchi, Anthony, Hummel, Susan Flynn, Lieber, Derek, Ngo, Ton, Mergen, Mark, Shepherd, Janice C., and Smith, Stephen. Implementing Jalepeño in Java. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Denver, CO, Oct. 1999), pp. 314–324.

[13] Appel, Andrew W. Garbage collection can be faster than stack allocation. *Information Processing Letters 25*, 4 (1987), 275–279.

[14] Appel, Andrew W. Simple generational garbage collection and fast allocation. *Software: Practice and Experience 19*, 2 (1989), 171–183.

[15] Bacon, David F., Cheng, Perry, and Rajan, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, CA, June 2003), ACM, pp. 81–92.

[16] Bacon, David F., Cheng, Perry, and Rajan, V. T. A real-time garbage collecor with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, LA, Jan. 2003), ACM, pp. 295–298.

[17] Bacon, David F., and Rajan, V. T. Concurrent cycle collection in reference counted systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (Budapest, Hugary, June 2001), Jørgen Lindskov Knudsen, Ed., vol. 2072, Springer-Verlag, pp. 207–235.

[18] Baker, Henry G. List processing in real-time on a serial computer. *Communications of the ACM 21*, 4 (1978), 280–294.

[19] Beger, Emery D. The Hoard memory allocator. Available at `http://www.hoard.org`.

[20] Berger, Emery D. *Memory Management for High-Performance Applications*. PhD thesis, Univeristy of Texas-Austin, 2002.

[21] Berger, Emery D., McKinley, Kathryn S., Blumofe, Robert D., and Wilson, Paul R. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, Nov. 2000), ACM, pp. 117–128.

[22] Berger, Emery D., Zorn, Benjamin G., and McKinley, Kathryn S. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, UT, June 2001), ACM, pp. 114–124.

[23] Berger, Emery D., Zorn, Benjamin G., and McKinley, Kathryn S. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Seattle, WA, Nov. 2002), ACM, pp. 1–12.

[24] Bishop, Peter B. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, May 1977.

[25] Blackburn, Stephen M., June 2003. Personal communication.

[26] Blackburn, Stephen M., Cheng, Perry, and McKinley, Kathryn S. Myths and reality: The performance impact of garbage collection. In *Proceedings of the 2004 Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, June 2004), ACM, pp. 25–36.

[27] Blackburn, Stephen M., Cheng, Perry, and McKinley, Kathryn S. Oil and water? high performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering* (Edinburgh, Scotland, May 2004), IEEE Computer Society, pp. 137–146.

[28] Blackburn, Stephen M., Garner, Robin, McKinley, Kathryn S., Diwan, Amer, Guyer, Samuel Z., Hosking, Antony, Moss, J. Eliot B., and Stefanović, Darko. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (2006), ACM. To Appear.

[29] Blackburn, Stephen M., Jones, Richard, McKinley, Kathryn S., and Moss, J. Eliot B. Beltway: getting around garbage collection gridlock. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 2002), ACM, pp. 153–164.

[30] Blackburn, Stephen M., and McKinley, Kathryn S. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Anaheim, CA, Oct. 2003), ACM, pp. 344–358.

[31] Blackburn, Stephen M., Singhai, Sharad, Hertz, Matthew, McKinley, Kathryn S., and Moss, J. Eliot B. Pretenuring for Java. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Tampa, FL, Oct. 2001), ACM, pp. 342–352.

[32] Blanchet, Bruno. Escape analysis for object oriented languages. application to Java. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Denver, CO, Oct. 1999), ACM SIGPLAN Notices, ACM, pp. 20–33.

[33] Bobrow, Daniel G., and Murphy, Daniel L. Structure of a LISP system using two-level storage. *Communications of the ACM 10*, 3 (Mar. 1967), 155–159.

[34] Bobrow, Daniel G., and Murphy, Daniel L. A note on the efficiency of a LISP computation in a paged machine. *Communications of the ACM 11*, 8 (Aug. 1968), 558–560.

[35] Boehm, Hans-Juergen. Reducing garbage collector cache misses. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), ACM, pp. 59–64.

[36] Boehm, Hans-Juergen, and Weiser, Mark. Garbage collection in an uncooperative environment. *Software: Practice and Experience 18*, 9 (Sept. 1988), 807–820.

[37] Brecht, Tim, Arjomandi, Eshrat, Li, Chang, and Pham, Hang. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Tampa, FL, June 2001), pp. 353–366.

[38] Burger, Doug, Austin, Todd M., and Bennett, Steve. Evaluating future microprocessors: The SimpleScalar tool set. Computer Sciences Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, Madison, WI, 1996.

[39] Cahoon, Brendon, and McKinley, Kathryn S. Data flow analysis for software prefetching linked data structures in Java controller. In *2001 International Conference on Parallel Architechtures and Compilation Techniques* (Barcelona, Spain, Sept. 2001), pp. 280–291.

[40] Cheney, C. J. A non-recursive list compacting algorithm. *Communications of the ACM 13*, 11 (Nov. 1970), 677–678.

[41] Cheng, Perry, and Belloch, Guy. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, UT, June 2001), pp. 125–136.

[42] Cheng, Perry, Harper, Ron, and Lee, Peter. Generational stack collection and profile-driven pretenuring. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, QC, Canada, June 1998), pp. 162–173.

[43] Cherem, Sigmund, and Rugina, Radu. Compile-time deallocation of individual objects. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (June 2006), pp. 138–149.

[44] Chilimbi, Trishul M., and Larus, James R. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Vancouver, BC, Canada, 1998), pp. 37–48.

[45] Cohen, Jacques. Garbage collection of linked data structures. *Computing Surveys 13*, 3 (Sept. 1981), 341–367.

[46] Collins, George E. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (Dec. 1960), 655–657.

[47] Colvin, Greg, Dawes, Beman, and Adler, Darin. C++ Boost smart pointers, Oct. 2004. Available at `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`.

[48] Cooper, Eric, Nettles, Scott, and Subramanian, Indira. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference Record of the 1992 ACM Symposium on LISP and Functional Programming* (San Francisco, CA, June 1992), pp. 43–52.

[49] Cormen, Thomas H., Leiserson, Charles Eric, and Rivest, Ronald L. *Introduction to algorithms*. MIT Press, Cambridge, MA, 1990.

[50] Corry, Erik. Optimistic stack allocation for Java-like languages. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Ottawa, ON, Canada, June 2006), pp. 162–173.

[51] Courts, Robert. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM 31*, 9 (Sept. 1988).

[52] Denning, Peter J. The working set model for program behaviour. *Communications of the ACM 11*, 5 (May 1968), 323–333.

[53] Detlefs, David L. Concurrent garbage collection for C++. In *Topics in Advanced Language Implementation*, Peter Lee, Ed. MIT Press, 1991, pp. 101–134.

[54] Deutsch, L. Peter, and Bobrow, Daniel G. An efficient incremental automatic garbage collector. *Communications of the ACM 19*, 9 (Sept. 1976), 522–526.

[55] Diwan, Amer, Tarditi, David, and Moss, J. Eliot B. Memory system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems 13*, 3 (Aug. 1995), 244–273.

[56] Domani, Tamar, Kolodner, Elliot K., and Petrank, Erez. A generational on-the-fly garbage collector for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Vancouver, BC, Canada, June 2000), pp. 274–284.

[57] Eeckhout, Lieven, Georges, Andy, and Bosschere, Koen De. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Anaheim, CA, Oct. 2003), pp. 169–186.

[58] Fenichel, Robert R., and Yochelson, Jerome C. A LISP garbage collector for virtual memory computer systems. *Communications of the ACM 12*, 11 (Nov. 1969), 611–612.

[59] Fitzgerald, Robert P., Knoblock, Todd B., Ruf, Erik, Steensgaard, Bjarne, and Tarditi, David. Marmot: an optimizing compiler for Java. *Software: Practice and Experience 30*, 3 (2000), 199–232.

[60] Guyer, Samuel Z., McKinley, Kathryn S., and Frampton, Daniel. Free-me: A static analysis for automatic individual object reclamation. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, ON, Canada, June 2006), pp. 364–375.

[61] Hanson, D. R. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience 20*, 1 (Jan. 1990), 5–12.

[62] Hertz, Matthew, and Berger, Emery D. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (San Diego, CA, June 2005), pp. 316–326.

[63] Hertz, Matthew, Blackburn, Stephen M., Moss, J. Eliot B., McKinley, Kathryn S., and Stefanović, Darko. Error free garbage collection traces: How to cheat and not get caught. In *Proceedings of the 2002 Joint International Conference on Measurement and Modeling of Computer Systems* (Marina Del Rey, CA, June 2002), pp. 140–151.

[64] Hirzel, Martin, Diwan, Amer, and Henkel, Johannes. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems 24*, 6 (Nov. 2002), 593–624.

[65] Hirzel, Martin, Diwan, Amer, and Hertz, Matthew. Connectivity-based garbage collection. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Anaheim, CA, Oct. 2003), pp. 359–373.

[66] Hirzel, Martin, Diwan, Amer, and Hosking, Tony. On the usefulness of liveness for garbage collection and leak detection. In *Proceedings of the 19th European Conference on Object-Oriented Programming* (Budapest, Hungary, June 2001), pp. 593–624.

[67] Huang, Xianglong, Moss, J. Eliot B., McKinley, Kathryn S., Blackburn, Stephen M., and Burger, Doug. Dynamic SimpleScalar: Simulating Java virtual machines. Tech. Rep. TR-03-03, University of Texas at Austin, Feb. 2003.

[68] Huang, Xianlong, Blackburn, Stephen M., McKinley, Kathryn S., Moss, J. Eliot B., Wang, Zhenlin, and Cheng, Perry. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Vancouver, BC, Canada, Oct. 2004), pp. 69–80.

[69] Inoue, Hajime, Stefanović, Darko, and Forrest, Stephanie. Object lifetime prediction in Java. *IEEE Transactions on Computers 55*, 7 (2006), 880–892.

[70] Johnstone, Mark S., and Wilson, Paul R. The memory fragmentation problem: Solved? In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Vancouver, BC, Canada, Oct. 1998), pp. 26–36.

[71] Jones, Richard E. Personal communication.

[72] Jones, Richard E., and Lins, Rafael. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, UK, July 1996.

[73] Kaplan, Scott F. In-kernel RIG: Downloads. Available at `http://www.cs.amherst.edu/~sfkaplan/research/rig/download`.

[74] Kim, Kin-Soo, and Hsu, Yarsun. Memory system behavior of Java programs: Methodology and analysis. In *Proceedings of the 2002 Joint International Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, June 2000), pp. 264–274.

[75] Knuth, Donald E. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison Wesley, Reading, MA, 1975.

[76] Korn, David G., and Vo, Kiem-Phong. In search of a better malloc. In *USENIX Conference Proceedings, Summer 1985* (Portland, OR, 1985), pp. 489–506.

[77] Lam, Michael S., Wilson, Paul R., and Moher, Thomas G. Object type directed garbage collection to improve locality. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag.

[78] Lea, Doug. A memory allocator, 1998. Available at `http://g.oswego.edu/dl/html/malloc.html`.

[79] Lieberman, Henry, and Hewitt, Carl E. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6 (June 1983), 419–429.

[80] Lim, Tian F., Pardyak, Przemyslaw, and Bershad, Brian N. A memory-efficient real-time non-copying garbage collector. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Vancouver, BC, Canada, Oct. 1998), pp. 118–129.

[81] McNamee, Dylan, and Armstrong, Katherine. Extending the Mach external pager interface to accomodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop* (1990), pp. 17–29.

[82] Michael, Maged. Scalable lock-free dynamic memory allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Washington, DC, June 2003).

[83] Moon, David A. Garbage collection in a large LISP system. In *Conference Record of the 1994 ACM Symposium on LISP and Functional Programming* (Austin, TX, Aug. 1994), pp. 235–245.

[84] Paz, Harel, Bacon, David F., Koloder, Elliot K., Petrank, Erez, and Rajan, V. T. Efficient on-the-fly cycle collection. In *Proceedings of the 14th International Conference on Compiler Construction* (Edinburgh, Scotland, Apr. 2005), vol. 3443 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 156–171.

[85] Plainfossé, David, and Shapiro, Marc. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (Kinross, Scotland, Sept. 1995), vol. 986 of *Lecture Notes in Computer Science*, Springer-Verlag.

[86] Reppy, John H. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, Dec. 1993.

[87] Ross, D. T. The AED free storage package. *Communications of the ACM 10*, 8 (Aug. 1967), 481–492.

[88] Sachindran, Narendran, and Moss, J. Eliot B. Mark-Copy: Fast copying GC with less space overhead. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Anaheim, CA, Oct. 2003), pp. 326–343.

[89] Sachindran, Narendran, Moss, J. Eliot B., and Berger, Emery D. MC^2: High-performance garbage collection for memory-constrained environments. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications* (Vancouver, BC, Canada, Oct. 2004), pp. 81–98.

[90] Savola, Pekka. LBNL Traceroute heap corruption vulnerability. Available at `http://www.securityfocus.com/bid/1739`.

[91] Shaham, Ran, Kolodner, Elliot K., and Sagiv, Mooly. On the effectiveness of GC in Java. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), pp. 12–17.

[92] Shaham, Ran, Kolodner, Elliot K., and Sagiv, Mooly. Heap profiling for space-efficient Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, UT, June 2001), pp. 104–113.

[93] Shaham, Ran, Kolodner, Elliot K., and Sagiv, Mooly. Estimating the impact of heap liveness information on space consumption in Java. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Berlin, Germany, June 2002), pp. 64–75.

[94] Smaragdakis, Yannis, Kaplan, Scott F., and Wilson, Paul R. The EELRU adaptive replacement algorithm. *Performance Evaluation 53*, 2 (July 2003), 93–123.

[95] Sosnoski, Dennis M. Java performance programming, part 1: Smart object-management saves the day. *JavaWorld* (Nov. 1999).

[96] Standard Performance Evaluation Corporation. SPECjbb2000. Available at `http://www.spec.org/jbb2000/docs/userguide.html`.

[97] Standard Performance Evaluation Corporation. SPECjvm98 documentation, Mar. 1999.

[98] Steele, Guy L. Multiprocessing compactifying garbage collection. *Journal of the ACM 18*, 9 (Sept. 1975), 495–508.

[99] Stefanović, Darko, McKinley, Kathryn S., and Moss, J. Eliot B. On models for object lifetimes. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), pp. 137–142.

[100] Sugalski, Dan. Squawks of the parrot: What the heck is: Garbage collection, June 2003. Available at `http://www.sidhe.org/~dan/blog/archives/000200.html`.

[101] Tong, Guanshan, and O'Donnell, Michael J. Leveled garbage collection. *Journal of Functional and Logic Programming 2001*, 5 (May 2001), 1–22.

[102] Torvalds, Linus. Re: Faster compilation speed, 2002. Available at `http://gcc.gnu.org/ml/gcc/2002-08/msg00544.html`.

[103] Ungar, David M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Apr. 1984), pp. 157–167.

[104] van Riel, Rik. rmap VM patch for the Linux kernel. Available at `http://www.surriel.com/patches/`.

[105] Venners, Bill. *Inside the Java Virtual Machine*. McGraw-Hill Osborne Media, Jan. 2000.

[106] Vestal, S. C. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*. PhD thesis, University of Washington, Seattle, WA, Jan. 1987.

[107] Weizenbaum, J. Knotted list structures. *Communications of the ACM 5*, 3 (Mar. 1962), 161–165.

[108] Wheeler, D. A. SLOCcount. Available at `http://www.dwheeler.com/sloccount`.

[109] Wikipedia. Comparison of Java to C Plus Plus, 2004. Available at `http://en.wikipedia.org/wiki/Comparison_of_Java_to_Cplusplus`.

[110] Wilson, Paul R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), Yves Bekkers and Jacques Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag.

[111] Wilson, Paul R., Johnstone, Mark S., Neely, Mark, and Boles, D. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science 986* (1995).

[112] Wilson, Paul R., Lam, Monica S., and Moher, Thomas G. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, ON, Canada, June 1991), pp. 177–191.

[113] Yang, Ting, Hertz, Matthew, Berger, Emery D., Kaplan, Scott F., and Moss, J. Eliot B. Automatic heap sizing: Taking real memory into account. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management* (Vancouver, BC, Canada, Nov. 2004), pp. 61–72.

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

[115] Zorn, Benjamin, and Grunwald, Dirk. Evaluating models of memory allocation. Tech. Rep. CU-CS-603-92, University of Colorado at Boulder, Boulder, CO, July 1992.