Automatic vs. Explicit Memory Management: Settling the Performance Debate

Matthew Hertz and Emery D. Berger
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{hertz,emery}@cs.umass.edu

ABSTRACT

While garbage collection's software engineering benefits are indisputable, its performance impact remains controversial. Garbage collection proponents argue that its benefits outweigh its costs, but it is widely believed that garbage collection imposes an unacceptably high runtime and space performance penalty. This paper aims to settle this debate. We present the first empirical comparison of the performance costs of automatic versus explicit memory management in a garbage-collected language. Using a tracing and simulation based *oracular memory manager*, we execute unaltered Java programs as if they used explicit memory management. We examine the runtime, space consumption and virtual memory footprint of Java benchmarks across a range of general-purpose allocators and both copying and non-copying garbage collectors. We show that, at large heap sizes and under no memory pressure, the runtime performance of some garbage collection algorithms is competitive with the Lea memory allocator and occasionally *outperforms* it by up to 4%. However, our results confirm that garbage collection requires six times the physical memory to achieve this performance and suffers order-of-magnitude performance penalties when paging occurs.

1. Introduction

Automatic memory management, or garbage collection, provides significant software engineering benefits over explicit memory management. Garbage collection frees the programmer from the burden of memory management and improves modularity, while preventing accidental memory overwrites ("dangling pointers") and security violations [36, 45]. However, garbage collectors must perform more work than explicit memory managers, which rely on the programmer to indicate when to deallocate individual objects. Garbage collectors, on the other hand, must periodically identify objects reachable by pointer traversal, and reclaim those that are unreachable (the *garbage*).

Previous researchers have measured the runtime performance and space impact of *conservative*, non-moving garbage collection[1] in C and C++ programs [17, 48]. For these programs, comparing the performance of explicit memory management to that of a suitable garbage collector is a simple matter of linking with the garbage collection library.

Unfortunately, measuring the performance trade-off of garbage collection in languages designed for garbage collection is not so straightforward. One cannot simply replace garbage collection with an explicit memory manager. Since programs written in these languages never explicitly deallocate objects, the result would be a rapid exhaustion of available memory. While one can measure the costs of garbage collection itself [18, 25, 29, 42], because garbage collection alters application behavior both by visiting and reorganizing memory, it is impossible to subtract out its effects [9]. Extrapolating the results of previous studies is inadequate because garbage collected languages typically permit the use of precise, copying garbage collectors. These collectors, especially generational variants, consistently outperform conservative non-relocating garbage collectors [9, 12]. The result is that while the software engineering benefits of garbage collection are indisputable, in the absence of empirical data, its performance impact remains a matter of religious debate.

In this paper, we aim to settle this debate. We conduct what we believe to be the first direct comparison of garbage collection to explicit memory management in a garbage-collected language. Our key insight is the following. We use exact object reachability traces [27, 28] as an oracle to indicate when objects should be deallocated. By implementing this oracle inside a detailed architectural simulator, we can execute and precisely measure *unaltered* Java applications as if they were written using explicit memory management.

We use this *oracular memory management* framework to measure the impact of garbage collection versus explicit memory management on runtime performance, cache-level and page-level locality. We perform these measurements across a wide range of benchmarks, five garbage collectors (including copying and non-copying collectors), and two explicit memory managers (the Kingsley and Lea (GNU libc) allocators [34, 46]). Our framework allows us to use the actual C implementations of these explicit memory managers.

We find that at large heap sizes, the Appel-style generational collector [5] consistently provides runtime performance that is competitive with explicit memory management, performing on average just 4% slower and occasionally *outperforming* it by up to 4%. However, this runtime performance requires over six times as much physical memory as a program using explicit memory management. We also find that explicit memory management has substantially better page-level locality, generally requiring half or fewer pages to run with the same number of page faults.

[1] In particular, the Boehm-Demers-Weiser collector [13].

Submitted to OOPSLA 2004, Vancouver, BC, Canada.

We believe these results will have a significant impact both on the acceptance of garbage collection and on the direction of future memory management research. While we show that little opportunity remains to improve throughput when memory is plentiful, our results demonstrate a substantial performance gap both in terms of space consumption and page-level locality.

The remainder of this paper is organized as follows. We discuss related work in Section 2. We present our oracular memory management framework in detail in Section 3. In Section 4, we discuss our experimental methodology and we report performance and space results across different garbage collectors, and explicit memory managers in Section 5. We discuss planned future work in Section 6 and conclude in Section 7 with a discussion of the implications of these results.

2. Related Work

In this section, we first illustrate the conventional wisdom on this topic, and then discuss the key areas of related work: direct measurement of garbage collection and of explicit memory management, comparisons of conservative garbage collection to explicit memory management in C and C++, and compile-time garbage collection.

2.1 Conventional wisdom

The view that explicit memory management provides superior performance to garbage collection is widespread. The reasons that authors generally cite for this advantage are cache locality and runtime overhead (including unpredictable pause times), rather than space overhead or page-level locality. Linus Torvalds pungently asserts that garbage collection has especially deleterious effects on cache locality: "there just aren't all that many worse ways to [expletive deleted] up your cache behaviour than by using lots of allocations and lazy GC to manage your memory", adding that "GC sucks donkey brains through a straw from a performance standpoint" [41]. Dan Sugalski, the architect of the Perl 6 interpreter, states that garbage collection will be beaten "by a manual tracking system's best case performance", although he believes that GC is "normally more cache friendly" [39]. The Wikipedia entry "Comparison of Java to C Plus Plus" argues that a disadvantage of Java is that "automatic garbage collection and mandatory virtual members make Java performance unsuitable for some applications" [44]. In his book *Inside the Java Virtual Machine*, Venners states that "a potential disadvantage of a garbage-collected heap is that it adds an overhead that can affect program performance", and that "[garbage collection] will likely require more CPU time than would have been required if the program explicitly freed unnecessary memory" [43]. We show here that the conventional wisdom expressed above is largely inaccurate.

2.2 Garbage collection measurements

Researchers have long appreciated that garbage collection can account for a significant fraction of program execution, and there are several studies that attempt to measure its cost. Ungar [42] and Steele [25] observed garbage collection overheads in LISP accounting for around 30% of application runtime. Using trace-driven simulations of six SML/NJ benchmarks, Tarditi et al. concluded that generational garbage collection accounts for 19% to 46% of application runtime [18].

Recent research has focused on using garbage collection to *improve* application performance. Wilson, Lam and Moher exploit the relocation phase of garbage collection to reorder data to improve page-level locality [47]. Chilimbi and Larus use on-line reference behavior to guide the reorganization of objects for improved cache-level locality [15], and Adl-Tabatabai et al. use a combination of a hardware performance monitoring unit, static analysis, and garbage collection to inject prefetches into program code [2]. A comparison of explicit memory management with such strategies is beyond the scope of this paper, but our results suggest that these approaches will lead garbage collection to consistently outperform explicit memory management.

2.3 Explicit memory management measurements

Many researchers have examined the cost of explicit memory managers, starting with Knuth's simulation-based study [32]. Korn and Vo compare a number of different explicit memory management algorithms and find that buddy allocation was consistently the fastest and most space-efficient [33]. Zorn examines the impact of different memory managers and a conservative garbage collector on runtime performance, memory consumption, and reference locality [48]; we discuss this work in Section 2.4. Johnstone and Wilson measure the space consumed by several conventional explicit memory managers and find that they yielded nearly zero fragmentation, or wasted memory beyond that induced by object headers and alignment restrictions [31]. They also conclude that the Lea allocator is the best overall allocator in terms of the combination of speed and memory usage. Berger et al. measure the runtime and space consumption of a range of benchmarks when using the Lea and Kingsley allocators as well as their counterparts written in the Heap Layers framework [7], and subsequently measure the time spent in memory operations for a range of benchmarks [8]. They find that programs using general purpose memory managers can spend up to 13% of program runtime in memory operations. As we show here, this apparently lower cost does not necessarily translate to improved runtime performance when compared to copying garbage collection.

2.4 Comparisons with conservative garbage collection

Previous comparisons of garbage collection to explicit memory management have understandably taken place in the context of conservative, non-relocating garbage collectors and C and C++. In his thesis, Detlefs compared the performance of garbage collection to explicit memory management for three C++ programs [17, p.47]. He found that garbage collection generally resulted in poorer performance (from 2% to 28% overhead), but also that the garbage-collected version of cfront performed 10% faster than a version modified to use general-purpose memory allocation exclusively. However, the garbage collected version still ran 16% slower than the original version of cfront using its custom memory allocators.

Zorn performed a direct empirical comparison of the cost of conservative garbage collection to explicit memory management in the context of C programs [48]. He found that the runtime performance of code using conservative garbage collection (specifically, the Boehm-Demers-Weiser collector [13]) was occasionally faster than explicit memory allocation. Moreover, the memory consumed by the BDW collector was almost always higher than that consumed by explicit memory managers, ranging from 21% less to 228% more. While these prior studies examine conservative garbage collectors running within C and C++ programs, our research focuses on the performance of code written from the outset to use garbage collection. We are thus able to examine the impact of precise copying garbage collectors, which typically have higher throughput than non-relocating garbage collectors [9].

2.5 Compile-time garbage collection

The goal of *compile-time garbage collection* is to avoid garbage collection altogether. The idea is to employ static analyses to determine when it is safe to deallocate objects, and to insert explicit deallocation calls at the appropriate point in the program. This line

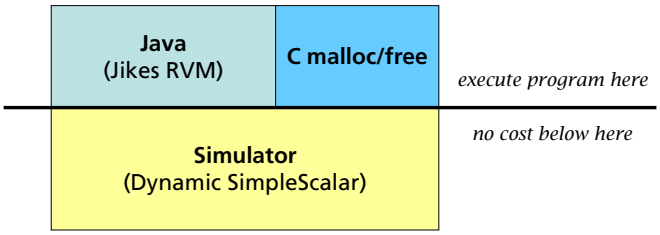


Figure 1: An overview of our oracular memory management framework. Using exact object reachability tracing and detailed architectural simulation allows us to quantify the performance of *unaltered* Java programs with automatic and explicit memory management.

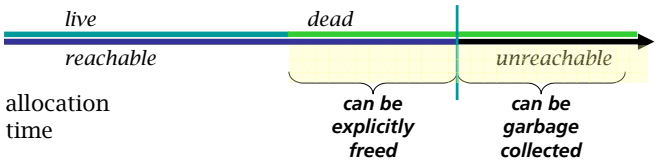


Figure 2: The lifetime of an object, showing when an explicit memory manager or an automatic memory manager might reclaim it. The oracular memory manager frees the object just before it becomes unreachable, which is the last possible time an explicit memory manager could free it.

of research dates back to Barth's description how a compiler could eliminate certain GC activities [6]. In addition to allowing us to evaluate the performance impact of garbage collection versus explicit memory management, our work represents a limit study for compile-time garbage collection. The oracular memory manager precisely simulates the impact of a perfect compile-time garbage collector, and so establishes the opportunity for performance improvement and space reduction available to compile-time garbage collection.

3. Oracular Memory Management

Figure 1 presents an overview of our oracular memory management framework. The framework consists of three parts: a lightly-modified Java virtual machine, exact object reachability traces, and a detailed architectural simulator.

3.1 The VM

For the Java virtual machine, we use the Jikes RVM, version 2.0.3, configured to target AIX and produce PowerPC code [3, 4]. Jikes is a widely-used research platform (formerly known as Jalapeño) that is written almost entirely in Java. A key advantage of Jikes is that it allows us to use a number of garbage collection algorithms, including those built using the University of Massachusetts Garbage Collection Toolkit (GCTk). These include classical algorithms such as semispace and an Appel-style generational collector [5], as well as recent garbage collectors such as Beltway [11] and Mark-Copy [35]. We also use the mark-sweep collector distributed with Jikes.

To provide predictable runs, we use the "Fast" configuration of Jikes. This configuration optimizes as much of the system as possible and compiles it into a prebuilt virtual machine. In addition, it uses the optimizing compiler on all code at runtime. While Java virtual machines normally use an adaptive system that only optimizes "hot" methods, previous work has shown that this skews results for short running programs [19]. To ensure reproducibility, we use deterministic thread switching, which switches threads based upon the number of methods executed.

3.2 The Oracle

To decide when to free objects without performing garbage collection, we need an oracle to inform us when these objects become unreachable. The oracle in this case is the exact object reachability that the Merlin algorithm provides [27, 28]. Merlin efficiently and precisely computes object reachability information by timestamping objects with the last time that they were known to be reachable. Our extended version of DSS generates traces that we subsequently use to compute exact object lifetimes with an offline implementation of the Merlin algorithm. For each object allocated, we compute exactly when (in allocation time) is last reachable. We run each benchmark twice, first to generate the trace we post-process and use as our oracle, and then to simulate explicit memory management.

Immediately before allocating a new object, the oracular memory manager frees any object that has just become unreachable, as Figure 2 shows. This policy is theoretically a worst-case approximation of explicit memory management, because the last possible moment that a program using explicit memory management can deallocate an object is just before it becomes unreachable. In practice, however, it is an excellent approximation. Programs often visit each object in a linked data structure before deallocating it. Shaham et al. measure the average impact of inserting `null` assignments in Java code, simulating nearly-perfect placement of "explicit deallocations" [37]. Their study includes five of the SPECjvm98 benchmarks we examine here. They report an average difference in space consumption of just 15% over deallocating objects when they become unreachable, showing that the policy of freeing objects when they become unreachable is close to the best achievable by a programmer.

3.3 The Simulator

Executing the oracle inside a Java virtual machine would distort program runtimes and space consumption. Object lifetime traces are quite large (several gigabytes), and the cost of processing these traces would dominate execution time. Therefore, we need a method of obtaining the information that the oracle provides at zero cost.

We accomplish this by executing the program on top of a simulator and placing the oracle code and processing inside the simulator. We use the Dynamic SimpleScalar detailed architectural simulator [30]. Dynamic SimpleScalar, or DSS, is an extension of the SimpleScalar superscalar architectural simulator [14] that permits the use of dynamically-generated code, a requirement for simulating the Jikes RVM. DSS executes the application and measures various performance characteristics including cycle count and cache miss rate. We branch into the simulator in order to execute the oracle with no measured cost, as we describe below.

3.4 Key Modifications

In order to create the oracular memory manager, we needed to make a number of significant changes to the Jikes RVM and Dynamic SimpleScalar. These modifications allow us to compute object reachability information, detect memory allocation operations, and insert explicit deallocation calls without distorting the simulated execution of the program.

Modifications to the Jikes RVM

We first need a number of modifications to compute object lifetimes using the Merlin algorithm offline. This algorithm requires that we capture all object allocations and intra-heap pointer updates as well as the current program roots at each object allocation. A key difficulty is determining when allocations occur and where objects are allocated. This determination cannot be done through simple tricks such as waiting for an allocation function to be executed, because the optimizing compiler inlines the allocation fast-path to improve system performance. Instead, we rely on minimal modification to the Jikes compiler.

The Jikes RVM has intermediate representations that differentiate between method calls within the VM and calls to the host operating system. While Jikes uses these parallel IR representations to minimize modifications needed to support OS calling conventions, we build upon this by adding a third set of nodes to represent calls to `malloc`. Our extension insures the compiler treats calls to `malloc` like any other function call, while also allowing us to emit a new, *illegal* opcode instead of the usual branch instruction. Because our new opcode is illegal, it cannot otherwise occur in any program. This provides us with a unique method to identify when new objects are allocated.

Replacing normal opcodes with illegal ones is at the heart of our approach for non-intrusively generating the needed heap traces. As with calls to `malloc`, we modify the Jikes RVM to replace intra-heap reference stores with other illegal instructions. These changes enable detecting object allocations and intra-heap pointer updates without inserting any trace-specific code. While this does require modifications to Java code within Jikes, their only effect is the substitution of our illegal opcodes for the normal PowerPC codes. Since we always use the simulator, our modified operations are included both for runs using the garbage collector and for those using the general-purpose allocators.

Modifications to DSS

The changes to Jikes described above required corresponding changes to DSS. These modifications generate the heap trace and insert `free` calls. When using illegal opcodes, detecting the heap pointer updates and object allocations is simple. We added code to the simulator to emit the size and address of newly allocated objects upon returning from a branch to `malloc` and to record the source and target of a pointer store instruction. Root references are also needed to accurately compute reachability information. We added to DSS code duplicating the Jikes RVM's register, stack and static scanning code, but working within the simulated memory structures. The simulator scans and emits root references into the heap trace without affecting the running Java program. In sum, these modifications allow us to generate the program heap trace without having any apparent effect on the benchmark.

The final change needed to Dynamic SimpleScalar is to insert calls to `free` when prompted by the oracle. We implement this by one last manipulation to our new opcode calling `malloc`. Before branching to the allocation function, we check if any objects must be freed. When jumping to `malloc`, our code caches the branch instruction return address. After allocating the new object with `malloc`, the simulator jumps to this return address for execution. By waiting for the simulator to execute this instruction, the address of the newly allocated object is discovered. When an object must be reclaimed before an allocation, we save the jump target and replace it with the address of `free`. The simulator also stashes the current function parameter and substitutes the previously saved address of the object to free. Finally, our branch is changed so that, once finished, execution returns to the `malloc` call instruction. When the branch is again reached, the simulator checks if more `free`s are needed. While frees are needed, the register and return values are again hijacked and the needed objects released. Once there are no objects to be reclaimed, the original register and branch target values are restored and allocation continues as normal.

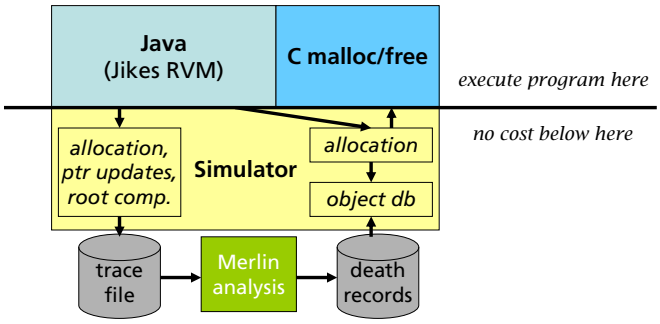


Figure 3: A detailed diagram of the oracular memory management framework. We first execute the Java program to measure it and to collect traces, which we process with the Merlin analysis to obtain exact object reachability information ("death records"). We then execute the program again, this time replacing allocation by calls to `malloc` and invoking `free` on objects when they become unreachable.

Garbage collectors	
Appel	variable-sized nursery using two generations [5]
Beltway	generalized copying GC framework [11]
Mark-Copy	space-efficient copying using two generations [35]
Mark-Sweep	non-relocating, non-copying single-generation
SemiSpace	two-space single-generation
Explicit memory managers	
Kingsley	segregated fits [46]
Lea	approximate best-fit with coalescing [34]

Table 1: Garbage collectors and explicit memory managers examined in this paper. All of the garbage collectors are copying collectors except Mark-Sweep.

4. Experimental Methodology

We compare the performance of 7 benchmarks across a variety of garbage collectors and allocators. Table 2 presents our benchmarks, which are primarily drawn from the SPECjvm98 benchmark suite [16]. We exclude two SPECjvm98 benchmarks, both of which are ray-tracing applications that cannot run in our framework due to limitations in the DSS floating point implementation. lpsixql is a persistent XML database system from the University of Colorado benchmark suite.

Table 1 lists the garbage collectors and explicit memory managers we analyze in this study. When using oracular memory management, we used the Kingsley [46] and Lea [34] allocators as the explicit memory managers.

The Kingsley allocator is a segregated fits allocator: all allocation requests are rounded up to the nearest size class. This rounding can lead to severe *internal fragmentation* (wasted space inside allocated objects), because in the worst case, it allocates twice as much memory as requested. Once an object is allocated for a given size, it can never be reused for another size: the allocator performs no

Benchmark statistics				
Benchmark	Total Bytes Alloc	Max. Bytes Live	Alloc/Live	URL
_201_compress	174,617,204	8,793,452	19.86	http://www.specbench.org/osg/jvm98
_202_jess	432,237,544	5,442,552	79.42	*ibid*
_209_db	151,308,296	10,677,544	14.17	*ibid*
_213_javac	521,763,424	11,802,216	44.21	*ibid*
_222_mpegaudio	102,667,248	4,382,192	23.43	*ibid*
_228_jack	480,765,676	5,327,612	90.24	*ibid*
ipsixql	128,955,804	4,485,088	28.75	http://systems.cs.colorado.edu/colorado_bench

Table 2: Memory usage statistics for our benchmark suite.

splitting (breaking large objects into smaller ones) or coalescing (combining adjacent free objects). This algorithm is well known to be among the fastest memory allocators although it is among the worst in terms of fragmentation [31].

We modified the Kinglsey allocator to make it suitable for use in the context of Java. In its original form, the Kingsley allocator uses power of two size classes. This choice results in catastrophic internal fragmentation for Java applications. We modified the allocator to include exact size classes for every multiple of four bytes up to 64 bytes, and then use powers of two for larger objects.

The Lea allocator is an approximate best-fit allocator that provides both high speed and low memory consumption. It forms the basis of the memory allocator included in the GNU C library [23]. The current version (2.7.2) is a hybrid allocator with different behavior based on object size. Small objects (less than 64 bytes) are allocated using exact-size quicklists (one linked list of freed objects for each multiple of 8 bytes). Requests for a medium-sized object (less than 128K) and certain other events trigger the Lea allocator to coalesce all of the objects in these quicklists in the hope that this reclaimed space can be reused for the medium-sized object. For medium-sized objects, the Lea allocator performs immediate coalescing and spliting and approximates best-fit. Large objects are allocated and freed using `mmap`. The Lea allocator is the best overall allocator (in terms of the combination of speed and memory usage) of which we are aware [31].

We compare the performance of explicit memory management to a number of copying garbage collectors: SemiSpace [20], an Appel-style generational collector [5], Beltway [11], and Mark-Copy [35]. We use the versions of these collectors as implemented in the GCTk memory management framework[2]. We also use one non-copying collector, the Mark-Sweep collector included with the Jikes distribution. Space does not permit a full description of each of these garbage collectors, but Table 1 sums up their key characteristics.

5. Experimental Results

We run each benchmark and garbage collector combination with heaps ranging from the smallest heap size needed by the collector to a heap six times larger. We base our memory configuration on the PowerPC G5 processor [1], and assume a 2 GHz clock. We use a 4K page size, as in Linux and Windows. When accounting for the cost of page faults, we assume an aggressive 5 millisecond page fault service time. Table 3 presents the architectural parameters we use in this study.

[2]GCTk has been superseded by the MMTk (formerly JMTk) toolkit [10], but at the time of this writing, it was not possible to use MMTk in our framework.

Memory hierarchy	
L1, I-cache	32K, 2-way associative, 3 cycle latency
L1, D-cache	64K, direct-mapped, 3 cycle latency
L2 (unified)	512K, 8-way associative, 11 cycle latency
	All caches have 128 byte lines
Main memory	
RAM	200 cycles (100ns)

Table 3: The architectural parameters used in this paper, based on the PowerPC G5 microprocessor.

5.1 Runtime

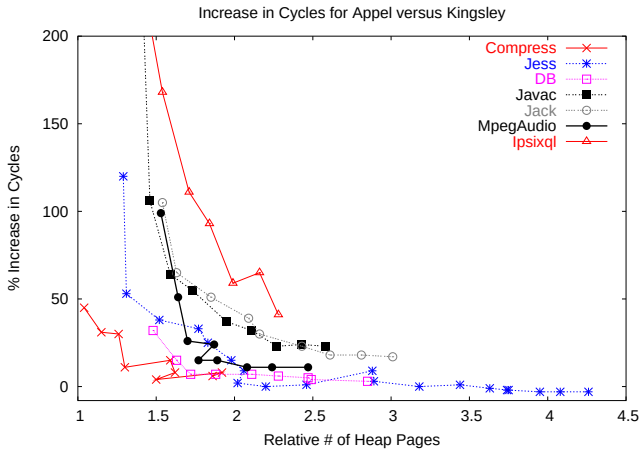
Figure 5(a) shows the relative performance of the *best* garbage collection strategy to explicit memory management. Assuming no memory pressure, garbage collection adds only 3% to the total execution time on average and improves the performance of jess by 4%. However, this figure assumes that one could select the fastest memory allocator and best collector configuration. Soman et al. present a dynamic selection mechanism for application-specific garbage collection that might be useful here [38].

At the largest heap sizes, garbage collection performance varies dramatically between collectors. Mark-Sweep is slightly faster then the Lea allocator on javac and performs as well as the Kingsley allocator on db. Mark-Sweep is also the fastest garbage collector on mpegaudio, which performs relatively little allocation. Appel runs jess faster then any other memory manager. Beltway is the fastest garbage collector for ipsixql and compress, while Mark-Copy is fastest on jack. While at the largest heap sizes the Appel and Mark-Copy collectors run ipsixql 5% faster than the Lea allocator, at all other heap sizes, all garbage collectors increase runtime by at least 9% over both explicit memory managers. When memory pressure is low and space is plentiful, explicit memory management rarely shows a clear improvement over garbage collection.

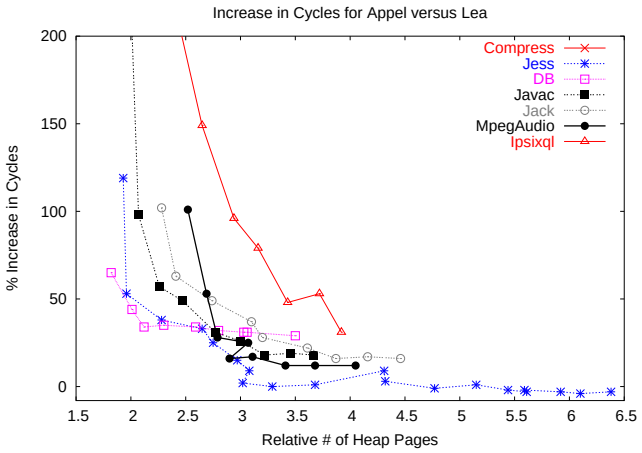
Figure 4 presents our runtime results across all garbage collectors and benchmarks. Each graph within this figure compares a particular garbage collector and explicit memory manager. Points in the graph represent the number of heap pages (the x-axis) and runtime (y-axis) for the garbage collection algorithm relative to the explicit memory manager. These graphs compactly summarize these results and present the time-space tradeoff involved in using garbage collection.

Figure 4 focuses on the Lea allocator[3]. As these graphs show, even at the largest heap sizes, the Lea allocator is faster on average then each GC algorithm. Figure 5(c) compares each collector's average performance versus the two memory allocators when con-

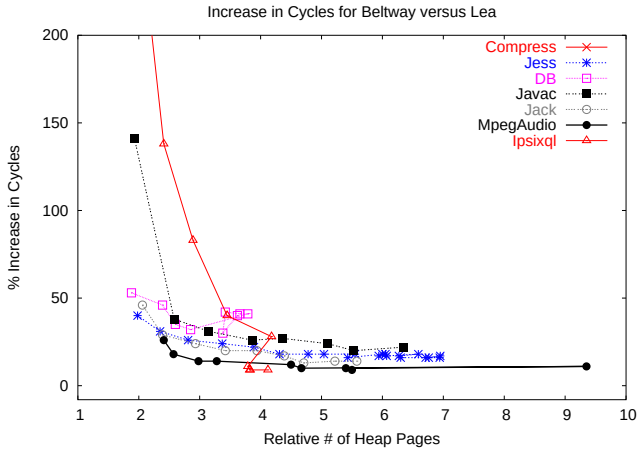
[3]compress triggers degenerate allocation behavior in the Lea allocator so we do not include it within these averages.



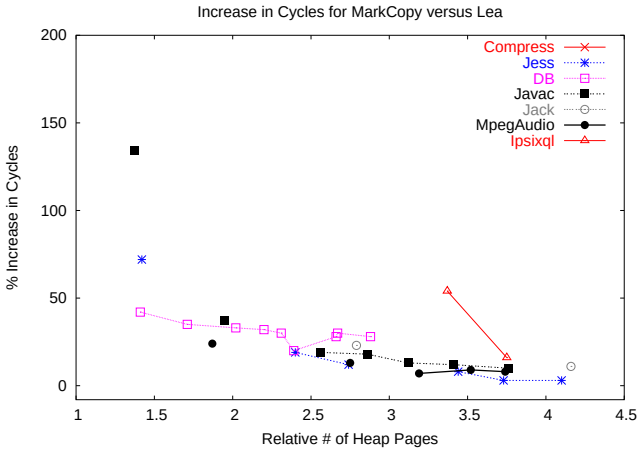
(a) Appel versus Kingsley



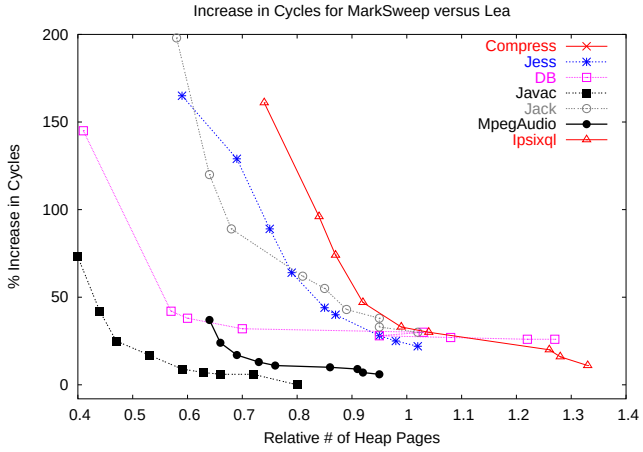
(b) Appel versus Lea



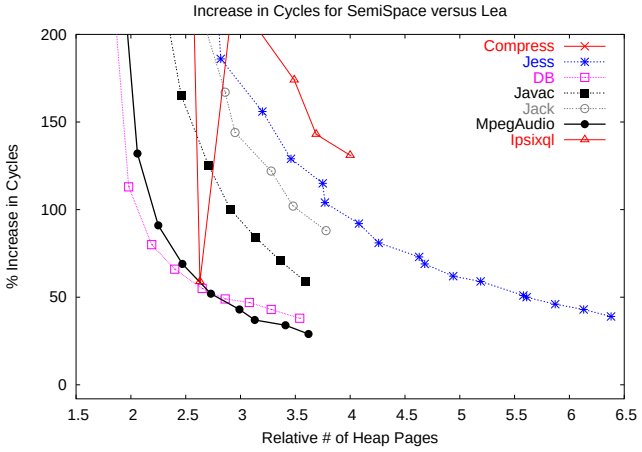
(c) Beltway versus Lea



(d) MarkCopy versus Lea

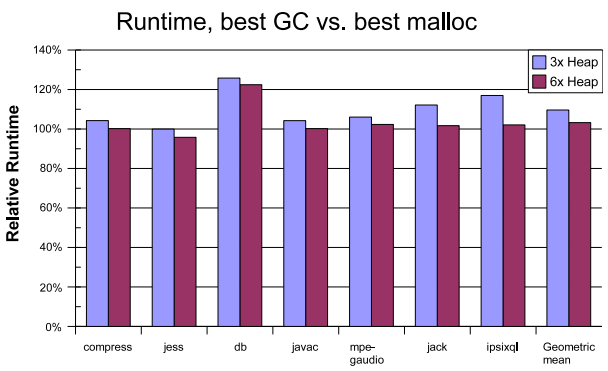


(e) MarkSweep versus Lea

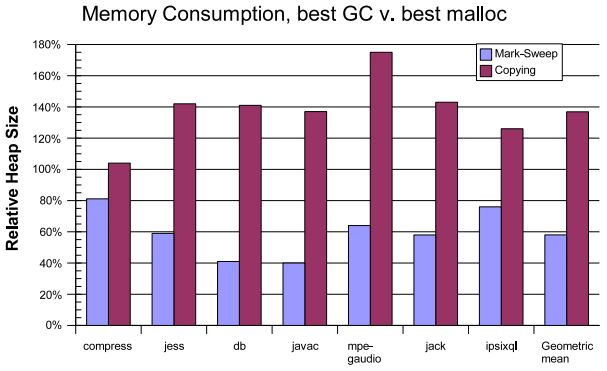


(f) SemiSpace versus Lea

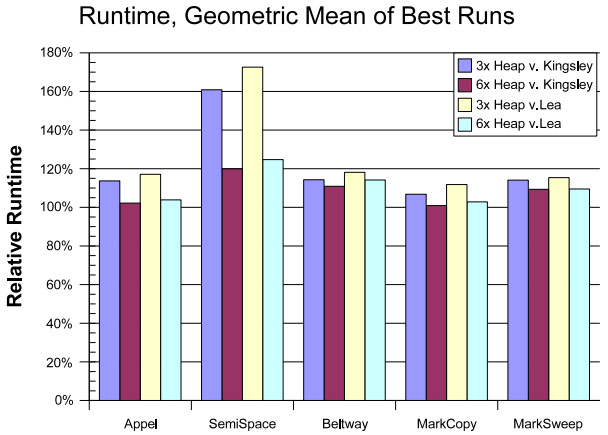
Figure 4: Runtime of garbage collectors versus the Lea allocator. For all but db **and** compress**, the runtime of the Lea and Kingsley allocators are similar, but where they differ, the Lea allocator is superior. On average, MarkCopy outperforms all other garbage collectors at the largest heap sizes, but the Appel collector actually improves over explicit memory manager performance by 4% on** jess**.**



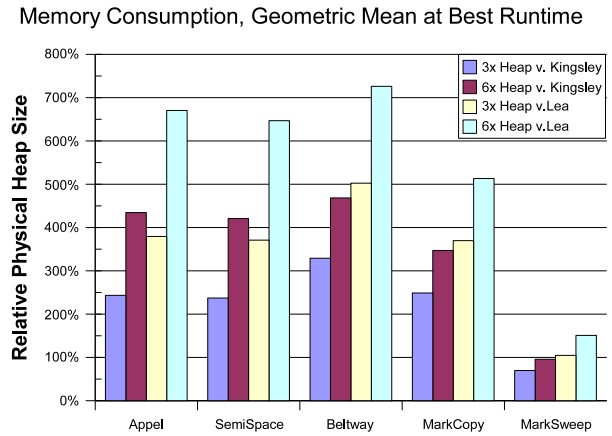
(a) Runtime performance of the best garbage collection versus the best explicit memory management.



(b) Memory consumption of the smallest heap for garbage collection versus the least space needed for explicit memory management.



(c) Average best runtime by garbage collector.



(d) Average relative heap size for the best runtime at each benchmark.

Figure 5: A summary of runtime and memory consumption results.

sidering only the collector's best run on each benchmarks. At the largest heap sizes, MarkCopy requires an average of 3% longer and Appel 4% longer than the Lea allocator to run our benchmarks. When we only consider heap sizes three times the collector's minimum required size, the results are quite different. MarkCopy now imposes an average runtime overhead of 9% over the Lea allocator, while Mark-Sweep and Appel add an average 10% and 17% runtime overhead, repectively. These results, however, are heavily influenced by db. This benchmark is very sensitive to cache performance and is the one benchmark where the Lea allocator shows better L1 D-cache locality then all the garbage collectors. Excluding db, MarkCopy is 1% faster than the Lea allocator at the largest heap size only 5% slower than the Lea allocator at the smaller heap sizes. Relative to the Kingsley explicit memory manager, the Appel collector runs only 2% slower and MarkCopy only 1% slower when heap space is most plentiful. SemiSpace's performance does help explain the concerns over garbage collection's runtime. Even at the largest heap size SemiSpace's runtime is within 10% of that needed for explicit memory manager for only compress and mpegaudio. On average, SemiSpace is over 20% slower than both allocators with the largest heaps heap six and imposes an over 60% runtime overhead with heaps three times the minimum size it needs. However, SemiSpace is rarely, if ever, used in a production system and its results should not be considered within any debate.

5.2 Memory consumption

While garbage collection execution time can compare favorably to that of the explicit memory managers, memory consumption is a less favorable comparison. Figure 5(b) shows the smallest garbage collected heap requires far less space then that of an explicit memory managers. While copying collection consumes more memory, this space is often not significantly greater. This does not consider the runtime increase required for garbage collection to maintain these smallest heaps. Figure 5(d) shows the average memory consumption with the heap using the best runtime for each collector.

Mark-Sweep is the only garbage collector that performs well while using less memory then either explicit memory manager. As Figure 5(d) shows, Mark-Sweep needs 4% less space then the Kingsley allocator and, when limited to the smaller heap sizes, only 5% more space then the Lea allocator. The Lea allocator allocates in 8-byte increments. Because Java allocates a large number of very small objects, this results in substantial internal fragmentation. More importantly, Mark-Sweep does not add the object headers that both explicit memory managers prefix to every allocation.

Every other garbage collector, when running at its fastest heap size, needs at least three times as much memory as even the space-inefficient Kingsley allocator. Because our method of freeing ob-

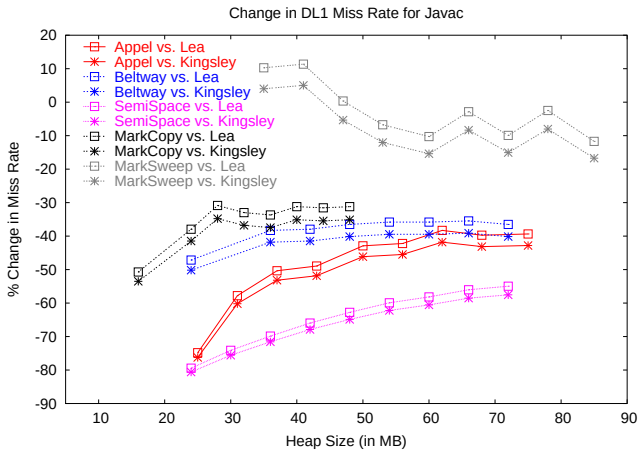


Figure 6: javac **L1 D-cache miss rates.**

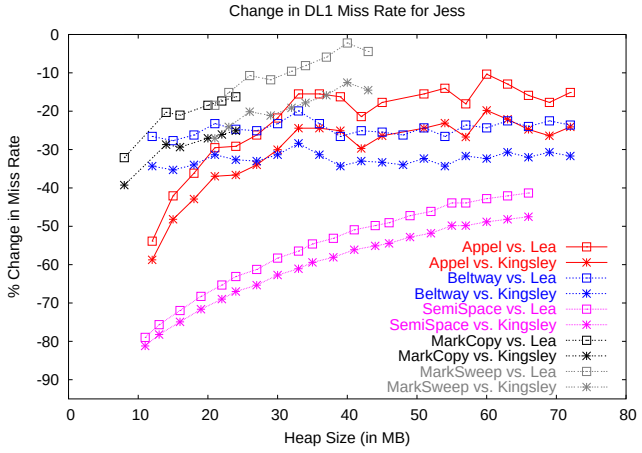


Figure 7: jess **L1 D-cache miss rates.**

jects also marks the earliest time a garbage collector can reclaim an object, some increase in memory consumption may not be surprising. However, the extent of this increase shows an area where garbage collection clearly needs improvement.

5.3 Cache performance

Figure 6 shows every run of javac using a copying collector had at least a 30% lower miss rate to the L1 D-cache then the Kingsley and Lea allocators. This improvement is highly correlated with copying collections, which place linked objects close to each other and generally reduce the space between live objects. Collectors copying larger portions of the heap, like Appel and SemiSpace, have the lowest miss rates, while Mark-Sweep has the highest rate. For all but Mark-Sweep, locality is best when garbage collection is most frequent, that is, at the smallest heap sizes.

Runs of jess show garbage collection improving the L1 D-cache rate even further. Figure 7 shows every run using garbage collection improving the cache locality. This figure also shows how the improvement to cache locality is tightly correlated to the fraction of the heap that can be moved with each collections. Only for runs of db does the Lea allocator consistently show a lower L1 D-cache miss rate. Runs of this benchmark have very little opportunity to

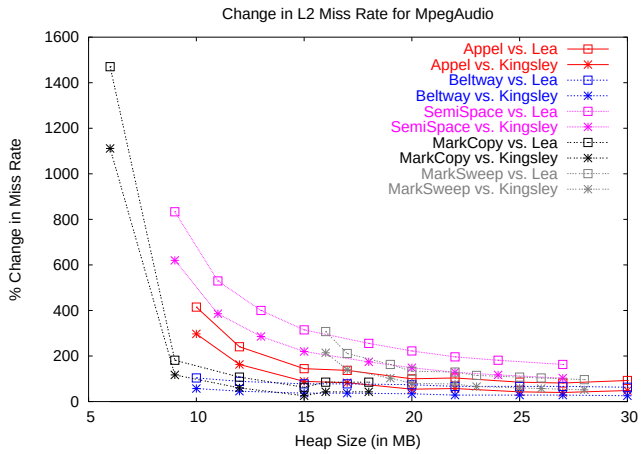


Figure 8: mpegaudio **L2 cache miss rates.**

collect garbage and thereby relocate objects. Beltway, SemiSpace and MarkSweep do improve the miss rate for the Kingsley allocator with db.

While garbage collection improves L1 cache locality, it hurts L2 cache locality. Figure 8 compares L2 cache locality for runs of mpegaudio, which is representative. Because the collector must examine a region far larger than the simulated L2 cache (512K), garbage collection increases the L2 miss rate. When garbage collections occur frequently, the L2 cache miss rate increases dramatically. Evidence for this is seen in the over 1500% increase for the smallest heap size of MarkCopy. As heap sizes increase, the L2 miss rates drop.

5.4 Page-level locality

We present the results of our page-level locality experiments in the form of miss curves. Assuming that the virtual memory manager observes an LRU discipline, these graphs show the number of page faults occurring for every possible number of pages allocated to the process.

Figure 9 compares the miss curves for both explicit memory manager versus an optimal garbage collector for each benchmark. The optimal garbage collector is one, given the benchmark and number of available memory pages, that selects the algorithm and heap size minimizing the number of page faults. Note that this heap size is often pessimal with respect to runtime. The x-intercept for these graphs corresponds to the smallest value for the maximum virtual memory footprint for each process. Every graph in Figure 9 shows the Lea and Kingsley allocators having smaller footprints then even this optimal garbage collector. Only in Figure 9(e) and Figure 9(f) are there garbage collectors with a footprint close to that of the explicit memory managers. These graphs also show that garbage collection has worse page locality for almost every experiment we performed. Only towards the left edge of the graphs of db (Figure 9(b)) and javac (Figure 9(c)) are there instances where garbage collection limits the number of page faults.

This analysis assumes that such an optimal garbage collector exists. Figure 10 shows the miss curves for the Appel collector running at a variety of heap sizes. As can be seen in this graph, the smallest footprint and number of page faults can often be seen in the smallest heap sizes. However, virtual memory footprint increases as heap sizes grow. Unfortunately, it is only at the largest heap sizes, and hence largest virtual memory footprints, that gar-

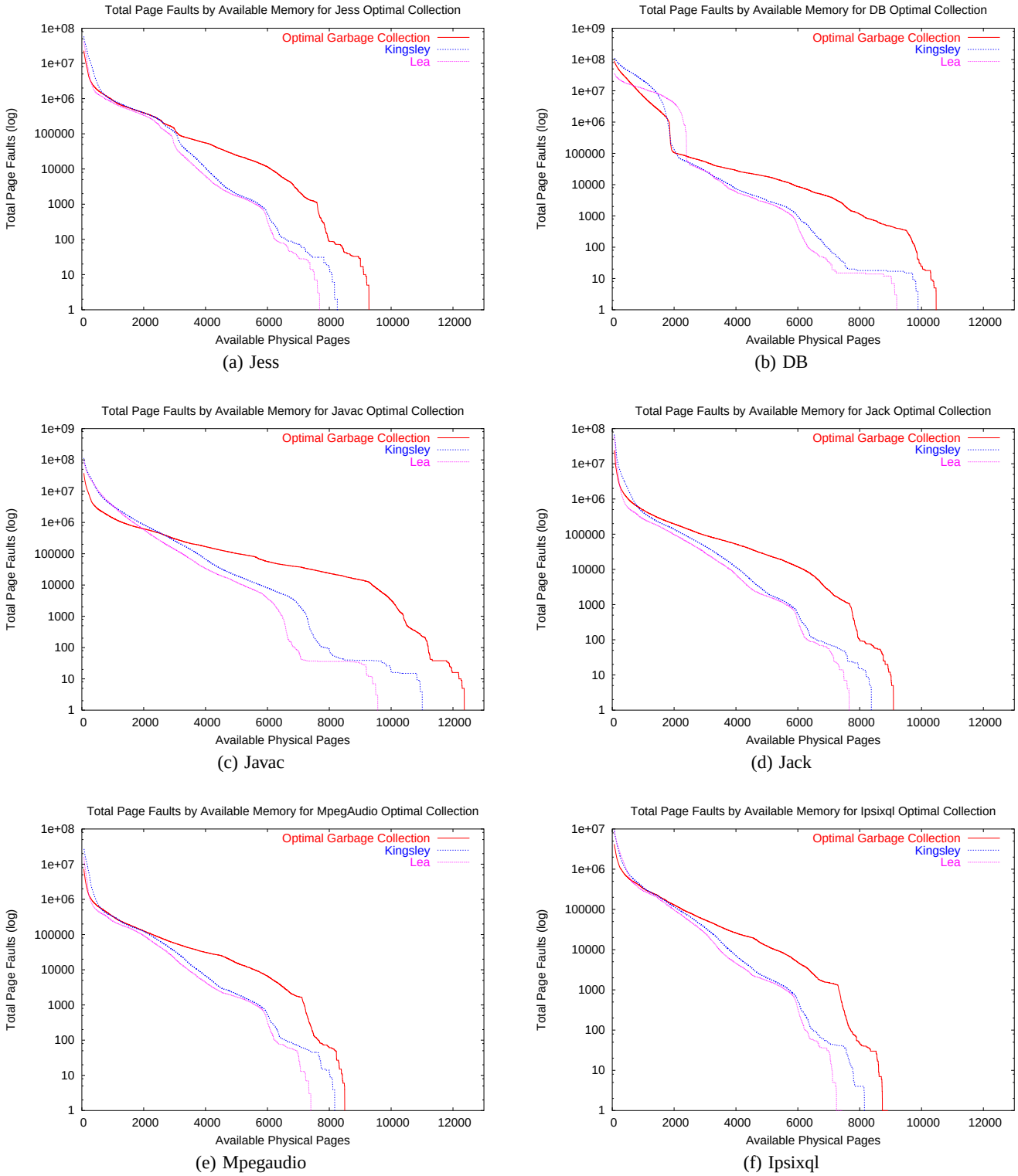


Figure 9: Page-level miss curves for six of the benchmarks (note that the y-axis is log-scale). Even if we select the optimal garbage collector and heap size, both explicit memory managers yield smaller footprints and fewer page faults at almost all reasonable physical memory sizes. Note that these "optimal" points are for the smallest heap sizes and so yield nearly pessimal throughput.

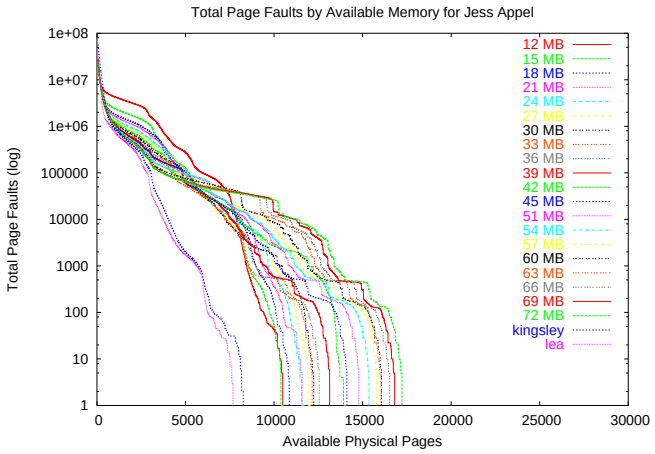


Figure 10: Page-level miss curves for the Appel collector running jess. **Larger heap sizes improve throughput but cause more page faults.**

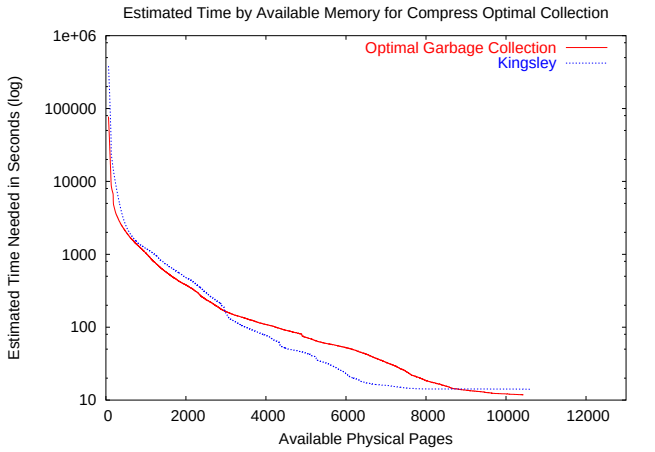


Figure 11: Execution time including paging costs for compress **(the y-axis is log-scale). Garbage collection suffers from paging problems much quicker.**

bage collection achieves runtime performance comparable to explicit memory management.

Figure 11 presents total execution times, including the cost of servicing page faults for compress. In this graph, the optimal garbage collector selects the garbage collector configuration that minimizes total execution time. There is little difference between the best possible GC configuration and the Kingsley allocator at the start. However, garbage collection performance quickly suffers from paging problems, and only again becomes comparable after paging slows execution by over an order of magnitude.

6. Future Work

While we have shown that garbage collectors can be competitive with and occasionally outperform explicit memory management, we have only addressed individual object management. Region-style custom memory allocators can dramatically improve the performance of applications using explicit memory management. [8, 26]. Regions are also increasingly popular as an alternative or complement to garbage collection [21, 22, 24, 40]. We plan to use our framework to examine the impact of the use of regions and a hybrid allocator, reaps [8], as compared to garbage collection.

In this study, we study two explicit memory allocators that place 8-byte object headers prior to each allocated object. As we discuss in Section 5.2, these headers lead to excessive space consumption compared to the Mark-Sweep collector. We plan to evaluate memory allocators that use bitmaps or BiBoP (big bag of pages) allocation, which we expect to yield substantially lower space consumption.

7. Conclusion

The controversy over garbage collection's performance impact has long overshadowed the software engineering benefits it provides. This paper introduces a tracing and simulation-based *oracular memory manager*. Using this framework, we execute a range of unaltered Java benchmarks using both garbage collection and explicit memory management. Comparing runtime, space consumption, and virtual memory footprints, we find that when space is plentiful, the runtime performance of garbage collection can be competitive with explicit memory management, and can even outperform it by up to 4%. We find that copying garbage collection can require six times the physical memory as the Lea or Kingsley allocators to provide comparable performance. We also show that garbage collection suffers orders-of-magnitude performance penalties when paging occurs. This first-ever comparison of explicit memory management and copying garbage collection shows where garbage collection must improve in the future. While we expect the incorporation of L3 caches to minimize the impact of garbage collection's poor L2 locality, the relative cost of disk latency continues to grow. Improving the space efficiency and page-level locality of garbage collection will thus become increasingly important.

8. Acknowledgments

Thanks to Eliot Moss, Chris Hoffmann, and the rest of the DSS team at UMass for their work on porting and debugging Dynamic SimpleScalar, and to Scott Kaplan, Kathryn McKinley, Yannis Smaragdakis, and the DaCapo group for helpful comments. Lastly, thanks to Linus Torvalds for inspiring us to settle this debate once and for all.

9. References

[1] Hardware — G5 Performance Programming. http://developer.apple.com/hardware/ve/g5.html, Dec. 2003.

[2] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Washington, DC, June 2004.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalepeño virtual machine. *IBM Systems Journal*, 39(1), Feb. 2000.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalepeño in Java. In *Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Languages, & Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 314–324, Denver, CO, Oct. 1999. ACM Press.

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

[6] J. M. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518, July 1977.

[7] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, Nov. 2002.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and reality: The performance impact of garbage collection. In *SIGMETRICS - Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems*, June 2004.

[10] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering*, page to appear, May 2004.

[11] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: getting around garbage collection gridlock. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 153–164. ACM, 2002.

[12] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA 2003 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, Oct. 2003.

[13] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, Sept. 1988.

[14] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Computer Sciences Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, Madison, WI, 1996.

[15] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the first international symposium on Memory management*, pages 37–48. ACM Press, 1998.

[16] S. P. E. Corporation. Specjvm98 documentation, Mar. 1999.

[17] D. L. Detlefs. Concurrent garbage collection for C++. In P. Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, 1991.

[18] A. Diwan, D. Tarditi, and E. Moss. Memory system performance of programs with intensive heap allocation. *ACM Trans. Comput. Syst.*, 13(3):244–273, Aug. 1995.

[19] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 38(11), pages 169–186, Oct. 2003.

[20] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, Nov. 1969.

[21] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 313 – 323, Montreal, Canada, June 1998.

[22] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 70 – 80, Snowbird, Utah, June 2001.

[23] W. Gloger. Dynamic memory allocator implementations in linux system libraries. http://www.dent.med.uni-muenchen.de/˜ wmglo/malloc-slides.html.

[24] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, Berlin, Germany, June 2002.

[25] J. Guy L Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, 1975.

[26] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. In *Software Practice & Experience*, volume 20(1), pages 5–12. Wiley, Jan. 1990.

[27] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, pages 140–151, Marina Del Ray, CA, June 2002.

[28] M. Hertz, N. Immerman, and J. E. B. Moss. Framework for analyzing garbage collection. In *Foundations of Information Technology in the Era of Network and Mobile Computing: IFIP 17th World Computer Congress - TC1 Stream (TCS 2002)*, pages 230–241, Montreal, Canada, Aug. 2002. Kluwer.

[29] M. W. Hicks, J. T. Moore, and S. M. Nettles. The measured cost of copying garbage collection mechanisms. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming*, pages 292–305. ACM, June 1997.

[30] X. Huang, J. E. B. Moss, K. S. Mckinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Feb. 2003.

[31] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *International Symposium on Memory Management*, Vancouver, B.C., Canada, 1998.

[32] D. E. Knuth. *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. Addison Wesley, Reading, MA, 1975. 2nd edition, 2nd printing.

[33] D. G. Korn and K.-P. Vo. In search of a better malloc. In *USENIX Conference Proceedings, Summer 1985*, pages 489–506, Portland, OR, 1985.

[34] D. Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 1998.

[35] N. Sachindran and J. E. B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming Systems, Languages and Applications*, Anaheim, CA, Oct. 2003.

[36] P. Savola. Lbnl traceroute heap corruption vulnerability. http://www.securityfocus.com/bid/1739.

[37] R. Shaham, E. . Kolodner, and M. Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Proceedings of the third international symposium on Memory management*, pages 64–75. ACM, 2002.

[38] S. Soman, C. Krintz, and D. F. Bacon. Dynamic Selection of Application-Specific Garbage Collectors. Technical Report 2004-09, Univ. of California, Santa Barbara, Jan. 2004. http://www.cs.ucsb.edu/čkrintz/abstracts/annotgc.html.

[39] D. Sugalski. Squawks of the parrot: What the heck is: Garbage collection. http://www.sidhe.org/ dan/blog/archives/000200.html, June 2003.

[40] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[41] L. Torvalds. Re: Faster compilation speed. http://gcc.gnu.org/ml/gcc/2002-08/msg00544.html, 2002.

[42] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, Apr. 1984. *ACM SIGPLAN Notices 19*, 5 (May 1984).

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

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

[45] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, Sept. 1992. Springer-Verlag.

[46] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 986, 1995.

[47] P. R. Wilson, M. S. Lam, and T. G. Moher. 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*, volume 26(6), June 1991.

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