Quantifying the Performance of Garbage Collection vs. Explicit Memory Management

Matthew Hertz *
Computer Science Department
Canisius College
Buffalo, NY 14208
matthew.hertz@canisius.edu

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

ABSTRACT

Garbage collection yields numerous software engineering benefits, but its quantitative impact on performance remains elusive. One can compare the cost of *conservative* garbage collection to explicit memory management in C/C++ programs by linking in an appropriate collector. This kind of direct comparison is not possible for languages designed for garbage collection (e.g., Java), because programs in these languages naturally do not contain calls to `free`. Thus, the actual gap between the time and space performance of explicit memory management and *precise*, copying garbage collection remains unknown.

We introduce a novel experimental methodology that lets us quantify the performance of precise garbage collection versus explicit memory management. Our system allows us to treat unaltered Java programs as if they used explicit memory management by relying on oracles to insert calls to `free`. These oracles are generated from profile information gathered in earlier application runs. By executing inside an architecturally-detailed simulator, this "oracular" memory manager eliminates the effects of consulting an oracle while measuring 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 conservatively frees objects just after they are last reachable. These oracles span the range of possible placement of explicit deallocation calls.

We compare explicit memory management to both copying and non-copying garbage collectors across a range of benchmarks using the oracular memory manager, and present real (non-simulated) runs that lend further validity to our results. These results quantify the time-space tradeoff of garbage collection: with five times as much memory, an Appel-style generational collector with a non-copying mature space matches the performance of reachability-based explicit memory management. With only three times as much memory, the collector runs on average 17% slower than explicit memory management. However, with only twice as much memory, garbage collection degrades performance by nearly 70%. When physical memory is scarce, paging causes garbage collection to run an order of magnitude slower than explicit memory management.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Dynamic storage management; D.3.4 [**Processors**]: Memory management (garbage collection)

General Terms

Experimentation, Measurement, Performance

Keywords

oracular memory management, garbage collection, explicit memory management, performance analysis, throughput, paging

1. Introduction

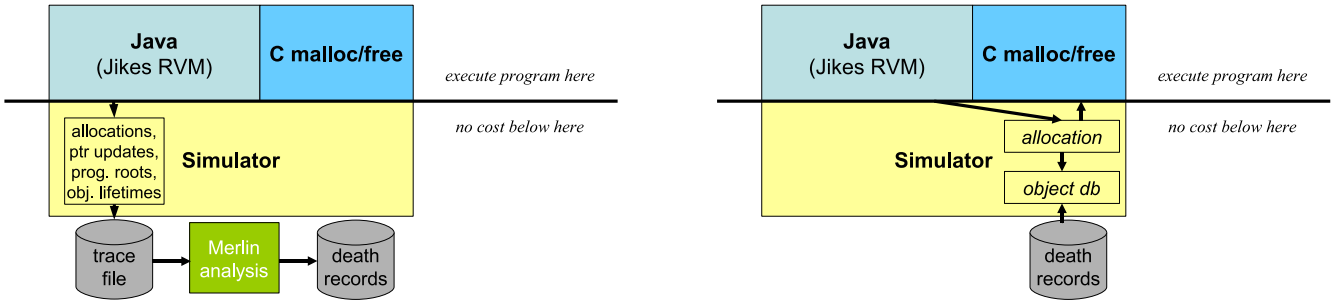
Garbage collection, or automatic memory management, provides significant software engineering benefits over explicit memory management. For example, garbage collection frees programmers from the burden of memory management, eliminates most memory leaks, and improves modularity, while preventing accidental memory overwrites ("dangling pointers") [50, 59]. Because of these advantages, garbage collection has been incorporated as a feature of a number of mainstream programming languages.

Garbage collection can improve the productivity of programmers [48], but its impact on performance is difficult to quantify. Previous researchers have measured the runtime performance and space impact of *conservative*, non-copying garbage collection in C and C++ programs [19, 62]. For these programs, comparing the performance of explicit memory management to conservative garbage collection is a matter of linking in a library like the Boehm-Demers-Weiser collector [14]. Unfortunately, measuring the performance trade-off in languages designed for garbage collection is not so straightforward. Because programs written in these languages do not explicitly deallocate objects, one cannot simply replace garbage collection with an explicit memory manager. Extrapolating the results of studies with conservative collectors is impossible because precise, relocating garbage collectors (suitable only for garbage-collected languages) consistently outperform conservative, non-relocating garbage collectors [10, 12].

It is possible to measure the costs of garbage collection activity (e.g., tracing and copying) [10, 20, 30, 36, 56] but it is impossible to subtract garbage collection's effect on mutator performance. Garbage collection alters application behavior both by visiting and reorganizing memory. It also degrades locality, especially when physical memory is scarce [61]. Subtracting the costs of garbage collection also ignores the improved locality that explicit memory managers can provide by immediately recycling just-freed memory [53, 55, 57, 58]. For all these reasons, the costs of precise,

*Work performed at the University of Massachusetts Amherst.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.



(a) Step one: collection and derivation of death records.

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

Figure 1: The oracular memory management framework.

copying garbage collection versus explicit memory management have never been quantified.

Contributions

In this paper, we conduct an empirical comparison of garbage collection to explicit memory management in Java. To enable this comparison, we develop an "oracular" memory manager. This memory manager relies on an oracle that indicates when the system should deallocate objects (i.e., by calling `free` on them). During a profiling run, the system gathers object lifetimes and generates a program heap trace that is later processed to generate the oracles. We use two different oracles that span the range of possible explicit deallocation calls. The *lifetime-based* oracle is the most aggressive: it uses object lifetimes to instruct the memory manager to `free` objects after their last use — the earliest time they can safely be freed. The *reachability-based* oracle is the most conservative: it reclaims objects at the last moment a program could call `free` (i.e., when they become unreachable). The reachability-based oracle relies on precise object reachability information obtained by processing the program heap traces with the Merlin algorithm [33, 34]. We discuss these two approaches in detail in Section 3.

We find that an on-line version of the reachability-based oracle interferes with mutator locality and increases runtime from 2%–33%. We eliminate this problem by executing the oracular memory manager inside an extended version of Dynamic SimpleScalar, an architecturally-detailed simulator [15, 39]. This approach allows us to measure the cost of Java execution and memory management operations while excluding disruptions caused by consulting the oracle. We believe this framework is of independent interest for studying memory management policies.

We use this framework to measure the impact of garbage collection versus explicit memory management on runtime performance, space consumption, and page-level locality. We perform these measurements across a range of benchmarks, garbage collectors (including copying and non-copying collectors), and explicit memory managers.

We find that GenMS, an Appel-style generational collector using a mark-sweep mature space, provides runtime performance that matches that provided by the best explicit memory manager when given five times as much memory, occasionally outperforming it by up to 9%. With three times as much memory, its performance lags so that it performs an average of 17% slower. Garbage collection performance degrades further at smaller heap sizes, ultimately running an average of 70% slower. Explicit memory management also exhibits better memory utilization and page-level locality, generally requiring half or fewer pages to run with the same number of page faults and running orders-of-magnitude faster when physical memory is scarce.

The remainder of this paper is organized as follows: Section 2 presents the oracular memory management framework in detail, and Section 3 discusses its implications and limitations. Sections 4 and 5 present experimental methodology and results comparing explicit memory management to a range of different garbage collectors. Section 6 addresses related work, Section 7 discusses future directions, and Section 8 concludes.

2. Oracular Memory Management

Figure 1 presents an overview of the oracular memory management framework. As Figure 1(a) shows, it first executes the Java program to calculate object lifetimes and generate the program heap trace. The system processes the program heap trace uses the Merlin algorithm to compute object reachability times and generate the reachability-based oracle. The lifetime-based oracle comes directly from the lifetimes computed during the profiling run. Using these oracles, the oracular memory manager executes the program as shown in Figure 1(b), allocating objects using calls to `malloc` and invoking `free` on objects when directed by the oracle (see Figure 2). Because trace generation takes place inside the simulator and oracle generation happens off-line, the system measures only the costs of allocation and deallocation.

Below we describe these steps in detail and discuss our solutions to the challenges of generating our oracles, detecting memory allocation operations, and inserting explicit deallocation calls without distorting program execution. We discuss the implications and limitations of our approach in Section 3.

2.1 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 [2, 3]. Jikes is a widely-used research platform written almost entirely in Java. A key advantage of Jikes and

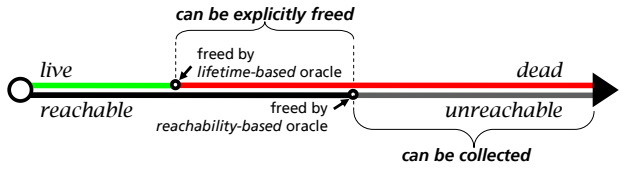


Figure 2: Object lifetime. The lifetime-based oracle frees objects after their last use (the earliest safe point), while the reachability-based oracle frees them after they become unreachable (the last possible moment an explicit memory manager could free them).

its accompanying Memory Management Toolkit (MMTk) is that it allows us to use a number of garbage collection algorithms [11]. The oracular memory manager executes inside Dynamic SimpleScalar (DSS) [39], an extension of the SimpleScalar superscalar architectural simulator [15] that permits the use of dynamically-generated code.

Repeatable Runs

Because the oracular memory manager uses allocation order to identify objects, we must ensure that the sequence of allocations is identical from run to run. We take a number of steps in Jikes RVM and the simulator to ensure repeatable runs. 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. Jikes RVM uses timer-based sampling at runtime to optimize methods once they reach a certain "hotness" level. To eliminate this considerable source of nondeterminism, we use a *pseudoadaptive* methodology [38, 49], which optimizes only "hot" methods, as determined from the mean of 5 runs. We also employ deterministic thread switching, which switches threads based upon the number of methods executed rather than at regular time intervals. Finally, we modify DSS to update the simulated OS time and register clocks deterministically. Rather than use cycle or instruction counts, which will change when we add calls to `free`, our modifications advance these clocks a fixed amount at each system call. These changes ensure that all runs are perfectly repeatable.

Tracing for the Liveness-Based Oracle

During a profiling run, the simulator calculates object lifetimes and generates program heap traces for later use. The simulator obtains per-object lifetime information by recording the location of every allocated object. At each memory access, the simulator then looks up the object being used and updates its latest lifetime (in allocation time). Unfortunately, this does not capture every use of an object. For example, testing for equality is a use of both argument objects, but Java's equality operation compares addresses and does not examine memory locations. To capture these object uses, we also mark all root-referenced objects as being in use. This extended definition potentially overestimates object lifetimes slightly, but eliminates the risk of freeing an object too early.

The system also preserves objects that we feel a programmer could not reasonably free. For instance, while our system can detect the last use of code and type information, these objects are not something that a developer would be able to deallocate in a real program. Similarly, our system will not free objects used to optimize class loading. These optimizations include objects mapping class member, string names, and type information in class files to their Jikes internal representation. Other objects that a programmer would not free and we therefore preserve enable lazy method compilation and reduce the time spent scanning jar files. At the end of the profiling run, the system preserves all of these objects and those to which these objects refer by extending their lifetime to the end of the program.

Tracing for the Reachability-Based Oracle

To generate the reachability-based oracle, we compute object reachability information efficiently and precisely using the Merlin algorithm [33, 34]. Our off-line implementation of the Merlin algorithm operates by analyzing a program heap trace. During this trace processing, the Merlin algorithm updates a timestamp associated with an object whenever it might become unreachable, e.g., when a pointer to it is overwritten. Once the entire trace has been processed, it generates "death records", a list ordered by allocation time that indicates which objects become unreachable at that time.

A key difficulty is capturing all the information required by the Merlin algorithm without affecting program execution. We need to execute the same code during step one (trace collection) as we do during step two (simulation). Otherwise, memory allocation and object lifetimes will be different, as the compiler will generate differently-sized chunks of code. However, there is a significant amount of information that the Merlin algorithm requires, including all object allocations, intra-heap pointer updates, and program roots whenever an object could be freed. It is not possible to obtain this information from the normal optimized code.

Replacing normal opcodes with illegal ones is at the heart of our approach for non-intrusively generating the needed heap traces. The new opcodes uniquely identify key events such as when new objects are allocated. When the simulator encounters such an opcode, it outputs a record into the trace. It then executes the illegal opcode exactly like its legal variant.

To enable the generation of these illegal opcodes, we extend Jikes RVM's compiler intermediate representations. Jikes RVM includes nodes in its IRs to differentiate between method calls within the VM and calls to the host system, minimizing the modifications needed to support different OS calling conventions. We build upon these by adding a set of nodes to represent calls to `malloc`. This extension allows the compiler to treat object allocations like any other function call, while emitting an illegal opcode instead of the usual branch instruction. We also modify Jikes RVM to replace intra-heap reference stores with illegal instructions. These opcodes allow us to detect events needed for heap tracing without inserting code that would distort instruction cache behavior.

2.2 Step Two: Simulating Explicit Memory Management

Before each allocation, the simulator consults the oracle to determine if any objects should be freed. When freeing an object, it saves the function parameter (the size request for `malloc`) and jumps to `free` instead, but sets the return address so that execution returns to the `malloc` call rather than the following instruction. The simulator repeats this cycle until there are no objects left to be reclaimed, and then allocation and program execution continues as normal. Both `malloc` and `free` are invoked via method calls. When these functions are implemented outside of the VM, they are called using the Jikes foreign function interface (`VM_SysCall`); we discuss the impact of this in Section 3.2.

2.3 Validation: Live Oracular Memory Management

In addition to the simulation-based framework described above, we implemented a "live" version of the oracular memory manager which uses the reachability-based oracle but actually runs on a real machine. Like the simulation-based oracular memory manager, the live oracle executes Java programs, but uses the actual instructions in place of the illegal ones. This live oracular memory manager uses the object lifetime information and a buffer recording where objects are allocated to fill a special "oracle buffer" containing the addresses of the objects to be freed. To determine the program's running time, we measure the total execution time and then subtract the time spent checking if objects should be freed and time spent refilling the buffer containing the addresses of the objects to free.

To measure the distortion introduced by the oracle, we compare the cost of running garbage collection as usual to running with a *null oracle*. The null oracle loads the buffers in the same way as the real oracular memory manager, but otherwise execution proceeds normally (it does not actually `free` any objects). We found that the distortion introduced is unacceptably large and erratic. For example, with the GenMS collector, the _228_jack benchmark with the null oracle reports a 12% to 33% increase in runtime versus

running with no oracle. By contrast, the null oracle slows the same collector down by at most 3% when running the _213_javac benchmark. Other collectors also show distortions from the null oracle, but without any obvious or predictable patterns. 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 lend credence to the simulation-based approach. As Figure 5 shows, the live version closely mirrors the trends of the reachability oracle simulation results.

3. Discussion

In the preceding sections, we have focused on the methodology we employ, which strives 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.

3.1 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 2.1, the liveness-based oracle preserves some objects beyond their last use. The liveness-based oracle also frees some objects that the reachability oracle does not. The number of objects involved is small: only pseudoJBB at 3.8% and _201_compress at 4.4% free more than 0.8% more objects. The liveness-based oracle makes these additional calls to `free` only for objects that do not become unreachable but which plausibly could be deallocated by a knowledgeable programmer.

Real program behavior is likely to fall between these two extremes. We would expect few programmers to reclaim objects immediately after their last use, and similarly, we would not expect them to wait until the very last point objects are reachable before freeing them. These two oracles thus bracket the range of explicit memory management options.

We show in Section 5.1 that the gap between these two oracles is small. Both oracles provide similar runtime performance, while the liveness oracle reduces heap footprints by at most 15% over the reachability oracle. These results generally coincide with previous studies of both C and Java programs. Hirzel et al. compare liveness to reachability on a benchmark suite including seven C applications [37]. For these, they find that when using an aggressive, interprocedural liveness analysis, they find a significant gap for two of their benchmarks, reducing average object lifetime by 11% for `gzip` (in allocation time) and 21% for `yacr2`); for the others, the gap remains below 2%. 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 [51]. They report an average difference in space consumption of 15% over deallocating objects when they become unreachable.

3.2 `malloc` Overhead

When using allocators implemented in C, the oracular memory manager invokes allocation and deallocation functions through the Jikes `VM_SysCall` foreign function call interface. While not free, these calls do not incur as much overhead as JNI invocations. Their total cost is just 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 functions defined in an external library (e.g., `libc.so`).

We also examine an allocator which implements `malloc` and `free` within the Jikes RVM. In this case, the oracular memory manager uses the normal Jikes RVM method call interface rather than the `VM_SysCall` interface. Because we still need to determine when an allocation occurs and, where appropriate, insert calls to `free`, we still cannot inline the allocation fast path. While this may prevent some potential optimizations, we are not aware of any explicitly-managed programming language that implements memory operations without function call overhead.

3.3 Multithreaded versus Single-threaded

In the experiments we present here, we assume a single-processor environment and disable atomic operations both for Jikes RVM and for the Lea allocator. In a multithreaded environment, most thread-safe memory allocators also require at least one atomic operation for every call to `malloc` and `free`: a test-and-set operation for lock-based allocators, or a compare-and-swap operation for non-blocking allocators [46]. These atomic operations are very costly on some architectures. For example, on the Pentium 4, the cost of the atomic `CMPXCHG` operation (compare-and-swap) is 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 [13].

However, the issue of multithreaded versus single-threaded environments is orthogonal to the comparison of garbage collectors and explicit memory managers, because explicit memory allocators can also avoid atomic operations for most memory operations. In particular, a recent version of Hoard [6] (version 3.2) maintains thread-local freelists, and generally uses atomic operations only when flushing or refilling them. Use of these thread-local freelists is cheap, normally through a register reserved for accessing thread-local variables. On architectures lacking such support, Hoard places the freelists at the start of each thread stack (aligned on 1MB boundaries), and accesses them by bitmasking a stack variable.

3.4 Smart Pointers

Explicit memory management can have other performance costs. For example, C++ programs might manage object ownership by using smart pointers. These templated classes transparently implement reference counting, which would add expense to every pointer update. For example, on the gc-bench benchmark, 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. [1]

However, smart pointers do not appear to be in widespread use. We searched for programs using the standard `auto_ptr` class or the Boost library's `shared_ptr` [16] on the open-source web site `sourceforge.net` and found only two large programs that use them. We attribute this lack of use both to their cost, since C++

[1] Richard Jones, personal communication.

Benchmark statistics			
Benchmark	Total Alloc	Max Reach	Alloc/Max
_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
ipsixql	214,494,468	8,996,136	23.84
pseudoJBB	277,407,804	32,831,740	8.45

Table 1: Memory usage statistics for our benchmark suite. Total allocation and maximum reachable are given in bytes. Alloc/max denotes the ratio of total allocation to maximum reachable, and is a measure of allocation-intensiveness.

Garbage collectors	
MarkSweep	non-relocating, non-copying single-generation
GenCopy	two generations with copying mature space
SemiSpace	two-space single-generation
GenMS	two generations with non-copying mature space
CopyMS	nursery with whole-heap collection
Allocators	
Lea	combined quicklists and approximate best-fit
MSExplicit	MMTk's MarkSweep with explicit freeing

Table 2: Memory managers examined in this paper. Section 4 presents a more detailed description of the allocators and collectors.

programmers tend to be particularly conscious of expensive operations, and to their inflexibility. For example, the same smart pointer class cannot be used to manage scalars and arrays, because C++ arrays require a different syntax for deletion (`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.

Nonetheless, some patterns of memory usage are inherently difficult to manage with `malloc` and `free`. For example, the allocation patterns of parsers makes managing individual objects an unacceptably-difficult burden. In these situations, C and C++ programmers often resort to custom memory allocators.

3.5 Custom Allocation

Many explicitly-managed programs use custom allocators rather than general-purpose allocators both to simplify and to accelerate memory management. In particular, Berger et al. show that region-style allocation is both useful for a variety of workloads and can be much faster than general-purpose allocation, but that they generally consume much more space than needed [8]. Exploring custom allocation policies like regions is beyond the scope of this paper.

3.6 Program Structure

The programs we examine here were written for a garbage-collected environment. Had they been written in a language with explicit memory management, they might have been written differently. Unfortunately, we do not see any way to quantify this effect. It would be possible (though onerous) to attempt to measure it by manually rewriting the benchmark applications to use explicit deallocation, but we would somehow have to factor out the impact of individual programmer style.

Despite the apparent difference in program structure that one might expect, we observe that it is common for Java programs to assign `null` to objects that are no longer in use. In this sense, programming in a garbage-collected environment is at least occasionally analogous to explicit memory management. In particular, explicit nulling of pointers resembles the use of `delete` in C++, which then can trigger a chain of class-specific object destructors.

4. Experimental Methodology

To quantify the performance of garbage collection versus explicit memory management, we compare the performance of eight benchmarks across a variety of garbage collectors. Table 1 presents our benchmarks. We include most of the SPECjvm98 benchmarks [18]. ipsixql is a persistent XML database system, and pseudoJBB is a fixed-workload variant of the SPECjbb benchmark [17]. pseudoJBB executes a fixed number of transactions (70,000), which simplifies performance comparisons.

For each benchmark, we run each garbage collector with heap sizes ranging from the smallest in which they complete to four times larger. For our simulated runs, we use the memory and processor configuration of a PowerPC G5 processor [1], and assume a 2 GHz clock. We use a 4K page size, as in Linux and Windows. Table 3 presents the exact architectural parameters.

Rather than relying on reported heap usage, we compare actual *heap footprints* by examining each run's maximum number of heap pages in use. Pages are "in use" only if they have been allocated from the kernel and touched. We do not include unused pages, such as allocated but untouched pages or pages that have been unmapped, since they are not assigned physical memory. Counting pages in use ensures the proper accounting of all memory usage, including metadata space, which is occasionally underreported.

For the oracular memory management experiments, we use both the Lea (GNU libc, "DLmalloc") allocator [44] and a variant of the MMTk MarkSweep collector. 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 [28]. The version used here (2.7.2) is a hybrid allocator with different behavior based on object size, although objects of different sizes may be adjacent in memory. Small objects (less than 64 bytes) are allocated using exact-size quicklists (one linked list of freed objects for each multiple of 8 bytes). The Lea allocator *coalesces* objects in these lists (combining adjacent free objects) in response to several conditions, such as requests for medium-sized objects. Medium objects are managed with immediate coalescing and splitting of the memory on the quicklists 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 [40].

While the Lea allocator is an excellent point of comparison, it differs significantly from the garbage collectors we examine here. Perhaps most importantly, it is written in C and not Java. In order to isolate the impact of explicit memory management, we added individual object freeing to MMTk's MarkSweep collector and large object manager ("Treadmill"). Each block of memory maintains its own stack of free slots and reuses the slot that has been most recently freed. This "allocator" is labelled as *MSExplicit* in the graphs.

Table 2 lists the garbage collectors we examine here, all of which are high-throughput "stop-the-world" collectors. These include a non-copying collector (MarkSweep [45]), two pure copying collectors (SemiSpace [25], and GenCopy [5]) and two hybrid collectors

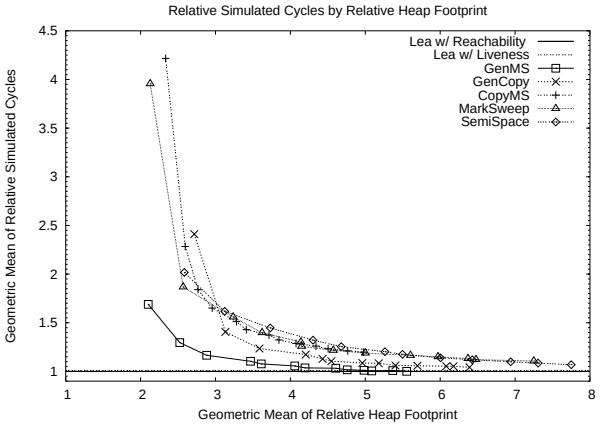


Figure 3: Geometric mean of garbage collector performance relative to the Lea allocator using the reachability oracle.

(GenMS and CopyMS). The generational collectors (the collector names starting with "Gen") use an Appel-style variable-sized nursery [5]: the nursery shrinks as survivors fill the heap. We use the versions of these collectors included with the MMTk memory management toolkit; the descriptions below are adapted from Blackburn et al. [11].

MarkSweep: MarkSweep organizes the heap into blocks divided into fixed-size chunks, which it manages with freelists. MarkSweep traces and marks the reachable objects, and lazily finds free slots during allocation.

SemiSpace: SemiSpace uses bump pointer allocation and has two copy spaces. It allocates into one, and when this space fills, it copies reachable objects into the other space and swaps them.

GenCopy: GenCopy uses bump pointer allocation. It is a classic Appel-style generational collector [5]. It allocates into a young (*nursery*) copy space and promotes survivors into an old SemiSpace. Its write barrier records pointers from old to nursery objects. GenCopy collects when the nursery is full, and reduces the nursery size by the size of the survivors. When the old space is full, it collects the entire heap.

GenMS: This hybrid generational collector is like GenCopy except that it uses a MarkSweep old space.

CopyMS: CopyMS is a non-generational collector (i.e., without write barriers) that uses bump pointer allocation to allocate into a copy space. When this space fills, CopyMS performs a whole-heap collection and copies survivors to a MarkSweep old space.

5. 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.

5.1 Runtime and Memory Consumption

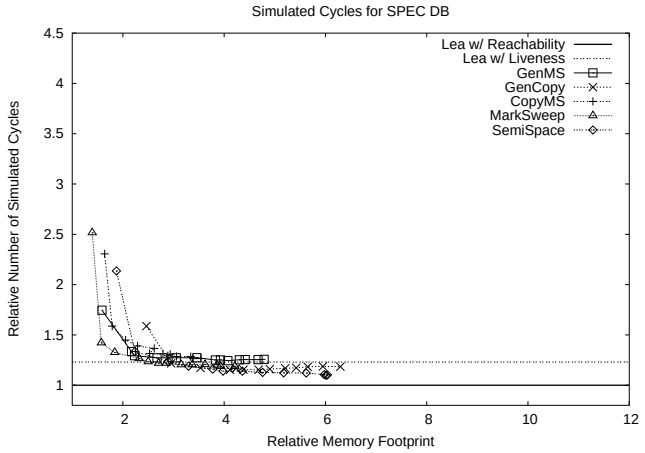
Figure 3 presents the geometric mean of garbage collection performance relative to the Lea allocator using the reachability oracle. We present runtime versus space results for individual benchmarks across all garbage collectors in Figure 4 . Each graph within this figure compares the garbage collectors and the Lea allocator using the reachability oracle. Points in the graph represent the heap footprint (the x-axis) and runtime (y-axis) for the garbage collection algorithm relative to the explicit memory manager. For readability, we do not include individual graphs for two of the benchmarks, _201_compress and _205_raytrace. Table 4 summarizes the results for the relative performance of GenMS, the best-performing garbage collector.

These graphs compactly summarize these results and present the time-space tradeoff involved in using garbage collection. Because we present pages actually touched rather than the requested heap size, they occasionally exhibit a "zig-zag" effect that can be surprising. As heap size increases, the number of heap pages normally also increases, but an increase in heap size can sometimes reduce the number of heap pages visited. For example, because of fragmentation or alignment restrictions, a larger heap size may cause objects to straddle two pages. This effect tends to be most pronounced for MarkSweep, which cannot reduce fragmentation by compacting the heap.

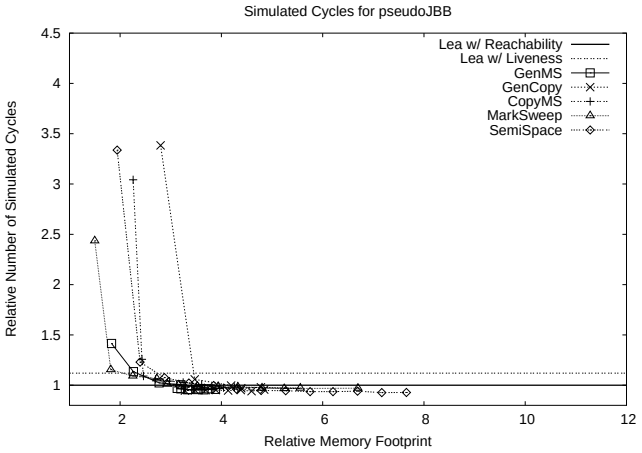
As the graphs show, the garbage collectors exhibit similar trends. Initially, in small heaps, the cost of frequent garbage collection dominates runtime. As heap sizes grow, the number of full-heap garbage collections correspondingly decreases. Eventually, total execution time asymptotically approaches a fixed value. For GenMS, this value is somewhat lower than the cost of explicit memory management. At its largest heap size, GenMS equals the performance of the Lea allocator. Its best relative performance on each benchmark ranges from 10% faster for ipsixql to 26% slower for _209_db, a benchmark that is unusually sensitive to locality effects.

The performance gap between the collectors is lowest for benchmarks with 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 size multiple becomes quite large (over 6x). The two generational collectors (GenMS and GenCopy) exhibit similar performance trends, although GenMS is normally faster. GenMS's MarkSweep mature space also makes it more space-efficient than GenCopy's mature space, which is managed by SemiSpace.

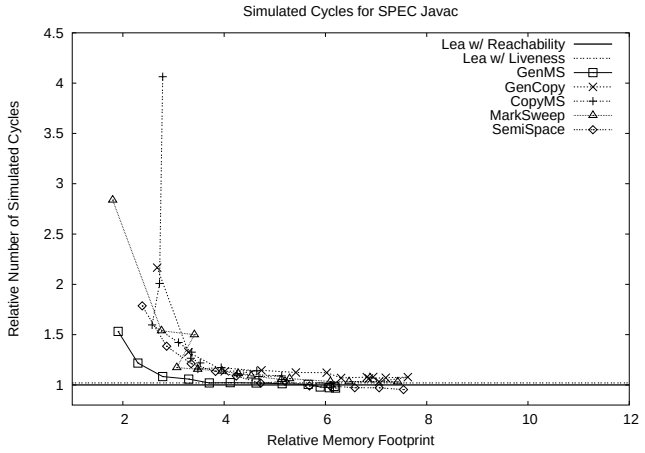
The shape of the garbage collection curves confirms analytical models that predict the performance of garbage collection to be inversely proportional to heap size [4; 41, p.35]. Note that the cost of explicit memory management does not depend on heap size, and is linear in the number of objects allocated. While this inverse proportionality relationship holds for MarkSweep and SemiSpace, we find that, on average, GenMS runs in time inversely proportional to the *square* of the heap size. In particular, the function execution time factor $= a/(b - \text{heap size factor}^2) + c$ characterizes the trend for GenMS, where the execution time factor is performance dilation with respect to Lea, and heap size factor is the multiple of the minimum required heap size. With the parameters $a = -0.246$, $b = 0.59$, and $c = 0.942$, the curve is an excellent fit. Visually, the curves are indistinguishable, and the rms (root mean square) error of the fit is just 0.0024, where 0 is a perfect fit. We find a similar result for GenCopy, whose rms error is just 0.0067 ($a = -0.297$, $b = 0.784$, $c = 1.031$). As far as we know, such inverse quadratic behavior has not previously been noted. We do not yet have an explanatory model, but conjecture that this behavior arises because the survival rate from nursery collections is also inversely proportional to heap size.



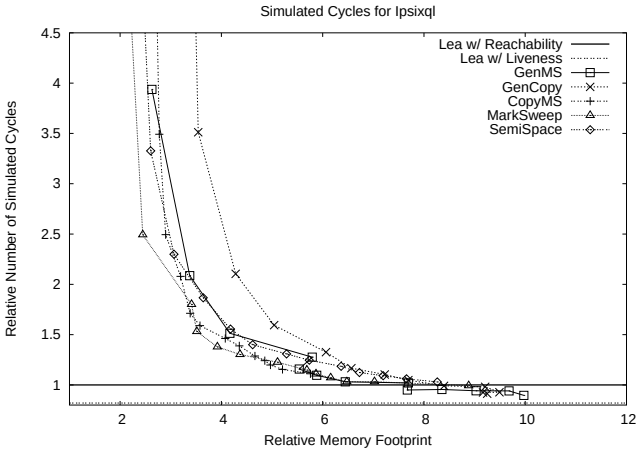
(a) _209_db (5.82)



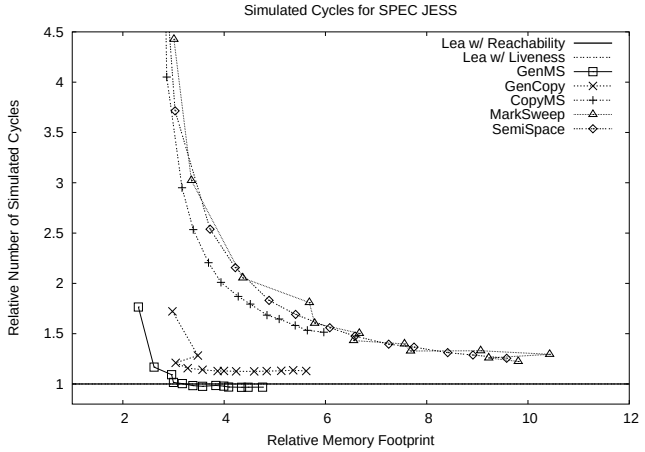
(b) pseudoJBB (8.45)



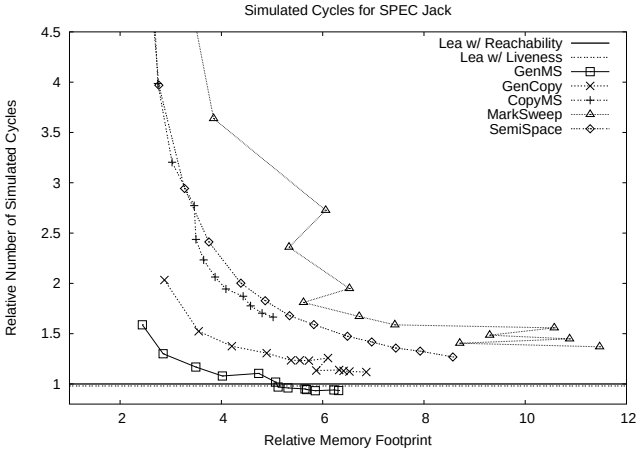
(c) _213_javac (16.27)



(d) ipsixql (23.84)



(e) _202_jess (36.02)



(f) _228_jack (39.63)

Figure 4: Runtime of garbage collectors versus the Lea allocator using the reachability oracle. The x-axis gives the maximum number of heap pages visited – the "zigzag" effect is primarily caused by fragmentation (see Section 5.1). The graphs are presented in increasing allocation intensity (alloc/max, given in parentheses); lower graphs are more allocation-intensive.

	Simulated PowerPC G5 system	Actual PowerPC G4 system
L1, I-cache	64K, direct-mapped, 3 cycle latency	32K, 8-way associative, 3 cycle latency
L1, D-cache	32K, 2-way associative, 3 cycle latency	32K, 8-way associative, 3 cycle latency
L2 (unified)	512K, 8-way associative, 11 cycle latency	256K, 8-way associative, 8 cycle latency
L3 (off-chip)	*N/A*	2048K, 8-way associative, 15 cycle latency
	all caches have 128 byte lines	*all caches have 32 byte lines*
RAM	270 cycles (135ns)	95 cycles (95ns)

Table 3: The memory timing parameters for the simulation-based and "live" experimental frameworks (see Section 2.3). The simulator is based upon a 2GHz PowerPC G5 microprocessor, while the actual system uses a 1GHz PowerPC G4 microprocessor.

	GenMS			
	vs. Lea w/ Reachability		vs. Lea w/ Liveness	
Heap size	**Footprint**	**Runtime**	**Footprint**	**Runtime**
1.00	210%	169%	253%	167%
1.25	252%	130%	304%	128%
1.50	288%	117%	347%	115%
1.75	347%	110%	417%	109%
2.00	361%	108%	435%	106%
2.25	406%	106%	488%	104%
2.50	419%	104%	505%	102%
2.75	461%	103%	554%	102%
3.00	476%	102%	573%	100%
3.25	498%	101%	600%	100%
3.50	509%	100%	612%	99%
3.75	537%	101%	646%	100%
4.00	555%	100%	668%	99%

Table 4: Geometric mean of memory footprints and runtimes for GenMS versus Lea. The heap sizes are multiples of the minimum amount required to run with GenMS.

	MSExplicit vs. GenMS			
	w/ Reachability		w/ Liveness	
Benchmark	**Footprint**	**Runtime**	**Footprint**	**Runtime**
_201_compress	162%	106%	251%	101%
_202_jess	154%	104%	165%	103%
_205_raytrace	131%	102%	147%	100%
_209_db	112%	118%	118%	96%
_213_javac	133%	95%	124%	93%
_228_jack	158%	103%	168%	105%
ipsixql	149%	100%	163%	97%
pseudoJBB	112%	106%	116%	87%
Geo. Mean	138%	104%	152%	98%

Table 5: Memory footprints and runtimes for MSExplicit versus Lea. In this table, we present results comparing results when run with similar oracles.

Finally, Table 5 compares the footprints and runtimes of MSExplicit (explicit memory management based on the MMTk MarkSweep implementation) and the Lea allocator when both use the same oracle. MSExplicit is substantially less memory-efficient than Lea, requiring between 38% and 52% more space. However, the results for runtime performance are similar. With the reachability oracle, MSExplicit runs an average of 4% slower than Lea; with the liveness-based oracle, it runs 2% faster. The worst-case for MSExplicit is for the locality-sensitive _209_db, where its segregated size classes cause it to run 18% slower when using the reachability oracle. On the other hand, it runs 5% faster than Lea for _213_javac with the reachability oracle, because this benchmark stresses raw allocation speed.

With the 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(f) is especially revealing: in this case, the runtime performance of MSExplicit is just 3% greater than that of Lea, but MarkSweep, using the same allocation infrastructure, runs from over 300% to 50% slower. These experiments demonstrate that the performance differences between explicit memory management and garbage collection are due to garbage collection itself and not to underlying differences in allocator infrastructure.

Comparing Simulation to the Live Oracle

We also compare the runtime performance of the various garbage collectors with the live oracle described in Section 2.3. 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 can be found in Table 3.

A comparison of the results of our live oracle experiments and simulations appears in Figure 5. These graphs compare the geometric means of executing all but three of the benchmarks. Because of the memory demands of the heap trace generation process and difficulties in duplicating time-based operating system calls, we are currently unable to run pseudoJBB, ipsixql, and _205_raytrace with the live oracle.

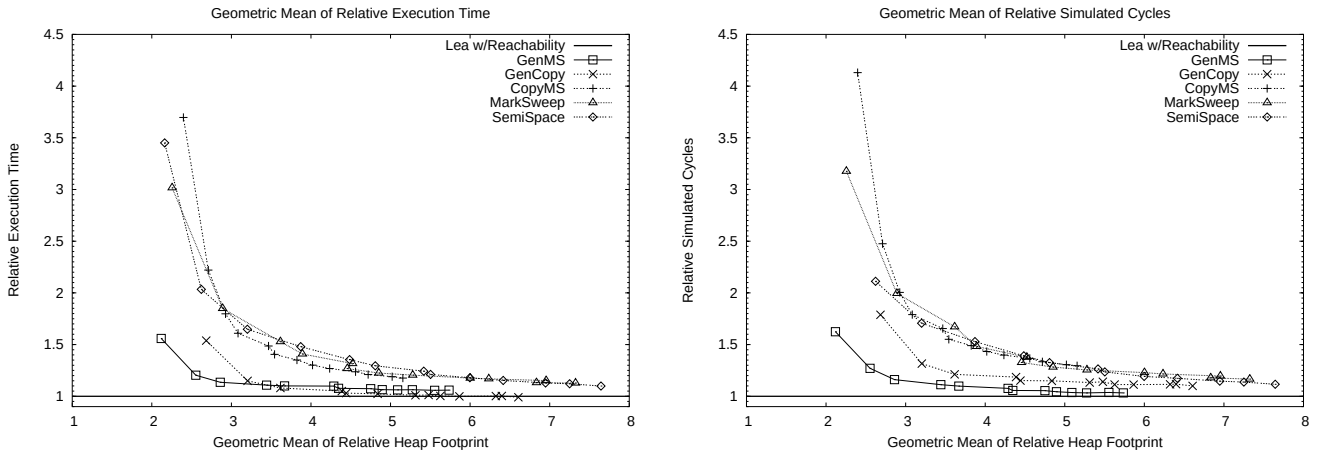
Despite their different environments, the live and simulated oracular memory managers achieve strikingly similar results. Differences between the graphs could be accounted for by the G4's L3 cache and smaller main memory latency compared to our simulator. While the null oracle adds too much noise to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

Comparing the Liveness and Reachability Oracles

We compare the effect of using the liveness and reachability-based oracles in Figure 6. This graph presents the average relative execution time and space consumption of allocators using both the liveness and reachability-based oracles. As usual, all values are normalized to the Lea allocator with 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 runtime graphs (Figures 3 and 4) also include a datapoint for the Lea allocator with the liveness oracle.

We find that the choice of oracle has little impact either on execution time. We expected the liveness-based oracle to improve performance by enhancing cache locality, since it recycles objects as soon as possible. However, this recycling has at best a mixed effect on runtime, degrading 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.



(a) Live oracular results (b) Simulated oracular results

Figure 5: Comparison of the "live" to the simulated oracular memory manager: geometric mean of execution time relative to Lea across identical sets of benchmarks.

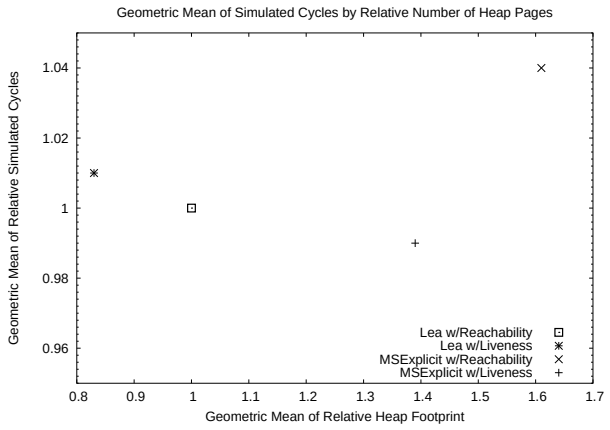


Figure 6: Geometric mean of explicit memory managers relative to the Lea allocator using the reachability oracle. While using the liveness oracle reduces the mean heap footprint substantially, the choice of oracle has little effect on mean execution time.

Figure 4(d) shows that ipsixql with the liveness-based oracle executes 18% faster than with the reachability oracle. This improvement is due to a halving of the L1 data cache miss rate. On the other hand, the liveness-based oracle significantly degrades cache locality in two other benchmarks, _209_db and pseudoJBB, causing them to execute 23% and 13% slower, respectively. While _209_db is notoriously susceptible to cache effects, the pseudoJBB result is surprising. In this case, using the lifetime-based oracle results in poor object placement, increasing the L2 cache miss rate by nearly 50%. However, these benchmarks are outliers. Figure 3 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 more pronounced impact on space consumption, reducing heap footprints by up to 15%. Using the liveness-based oracle reduces Lea's average heap footprint by 17% and MSExplicit's by 12%. While _201_compress's reliance on several large objects limits how much the liveness-based oracle can improve its heap footprint, all other benchmarks see their space consumption reduced by at least 10%.

5.2 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* [52, 61]. Assuming that the virtual memory manager observes an LRU discipline, these graphs show the time taken (y-axis) for different number of pages allocated to the process (the x-axis). Note that the y-axis is log-scale. We assume a fixed 5 millisecond page fault service time.

Figure 7 presents the total execution times for the Lea allocator, MSExplicit, and each garbage collector across all benchmarks. For each garbage collector, the fastest-performing heap size was selected.

These graphs show that, for reasonable ranges of available memory (but not enough to hold the entire application), both explicit memory managers substantially outperform all of the garbage collectors. For instance, pseudoJBB running with 63MB of available memory and the Lea allocator completes in 25 seconds. With the same amount of available memory and using GenMS, it takes more than ten times longer to complete (255 seconds). We see similar trends across the benchmark suite. The most pronounced case is _213_javac: at 36MB with the Lea allocator, total execution time is 14 seconds, while with GenMS, total execution time is 211 seconds, over a 15-fold increase.

The culprit here is garbage collection activity, which visits far more pages than the application itself [61]. As allocation intensity increases, the number of major garbage collections also increases. Since each garbage collection is likely to visit pages that have been evicted, the performance gap between the garbage collectors and explicit memory managers grows as the number of major collections increases.

6. Related Work

Previous comparisons of garbage collection to explicit memory management have generally taken place in the context of conservative, non-relocating garbage collectors and C and C++. In his thesis, Detlefs compares the performance of garbage collection to explicit memory management for three C++ programs [19, p.47]. He finds

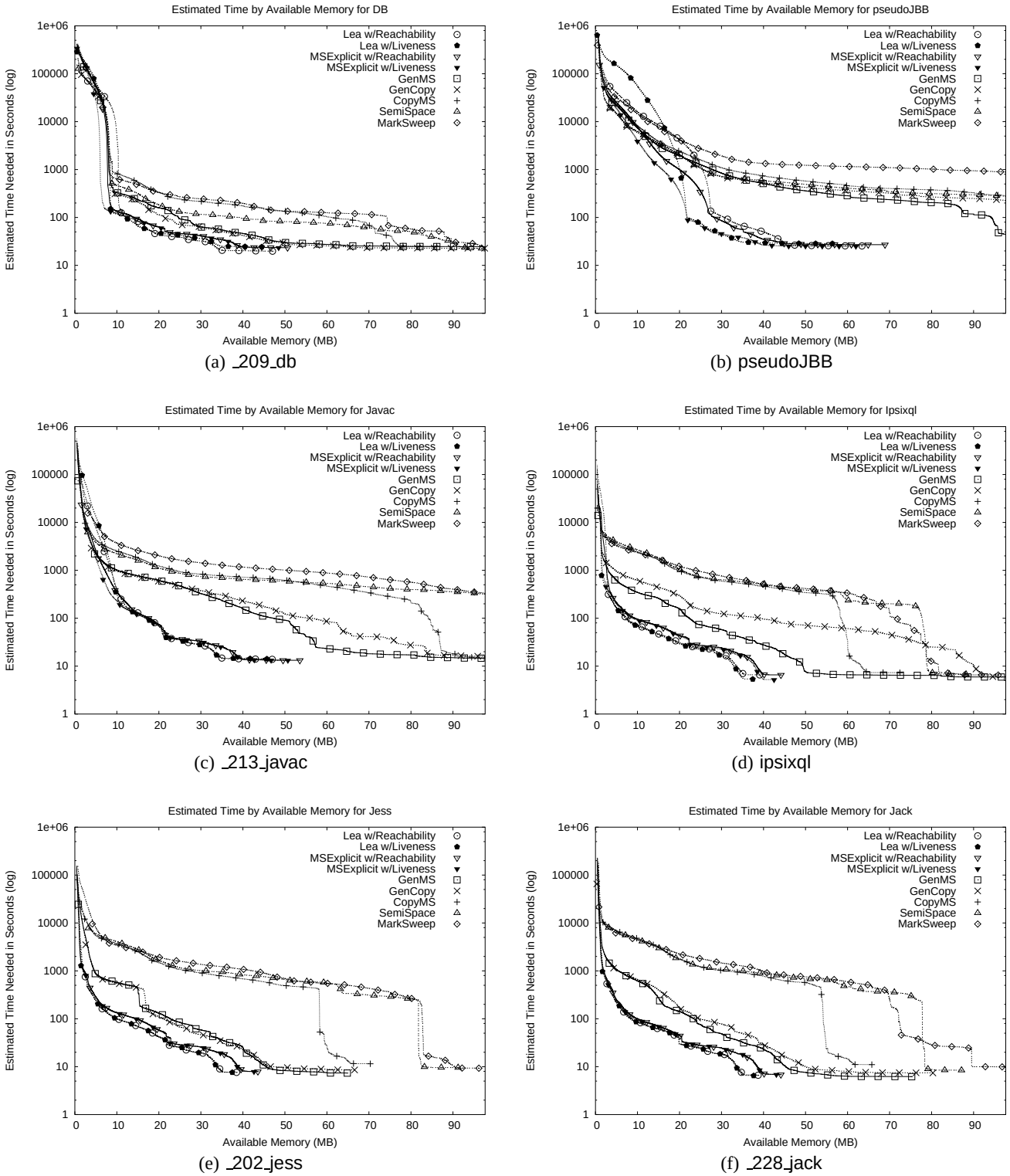


Figure 7: Estimated time for six of the benchmarks, including page fault service time (note that the y-axis is log-scale). The graphs are presented in increasing allocation intensity.

that garbage collection generally resulted in poorer performance (from 2% to 28% overhead), but also that the garbage-collected version of cfront performs 10% faster than a version modified to use general-purpose memory allocation exclusively. However, the garbage-collected version still runs 16% slower than the original version of cfront using its custom memory allocators. Zorn compares conservative garbage collection to explicit memory management in the context of C programs [62]. He finds that the Boehm-Demers-Weiser collector [14] is occasionally faster than explicit memory allocation, but that the memory consumed by the BDW collector is almost always higher than that consumed by explicit memory managers, ranging from 21% less to 228% more. Hicks et al. also find that programs written in Cyclone (a type-safe variant of C) and linked with the BDW collector can require much more memory than those using pure explicit memory management [35]. They also find that conservative garbage collection provides throughput equal to explicit memory management for most of their benchmarks, but that one benchmark showed a significant performance improvement with explicit memory management. While these studies examine conservative garbage collectors running within C and C++ programs, we focus on the performance of code written from the outset to use garbage collection.

Perhaps the closest work to that presented here is by Blackburn et al., who measure a similar range of garbage collectors and benchmarks in the Jikes RVM and the Memory Management Toolkit (MMTk) [10]. They conclude that generational garbage collection achieves locality benefits that make it faster than freelist-style allocation. To approximate explicit memory management, they measure the mutator time of execution with the MMTk mark-sweep garbage collector, and show that this exceeds the total execution time with generational collectors. This approach does not account for either cache pollution caused by garbage collection or the beneficial locality effects that explicit memory managers achieve by promptly recycling allocated objects. In addition, the MMTk mark-sweep collector segregates objects by size and thus disrupts allocation order. The Lea allocator we use here maintains segregated free lists but allows objects of different sizes to be adjacent in memory. Preserving allocation order is especially important for Java, since it approximates the locality effects of object inlining [21, 22, 23].

Numerous studies have sought to quantify the overhead of garbage collection and explicit memory management on application performance [7, 9, 40, 43, 62]. Steele observes garbage collection overheads in LISP accounting for around 30% of application runtime [30]. Ungar measures the cost of generational scavenging in Berkeley Smalltalk, and finds that it accounts for just 2.5% of CPU time [56]. However, this measurement excludes the impact of garbage collection on the memory system. Using trace-driven simulations of eight SML/NJ benchmarks, Diwan et al. conclude that generational garbage collection accounts for 19% to 46% of application runtime (measured as cycles per instruction) [20].

Using a uniform cost model for memory accesses, Appel presents an analysis that shows that given enough space, garbage collection can be faster than explicit memory management [4] (see Miller for a rebuttal of this claim with respect to stack allocation of activation frames [47]). He observes that the frequency of collections is inverse to the heap size, while the cost of collection is essentially constant (a function of the maximum reachable size). Increasing the size of the heap therefore reduces the cost of garbage collection. Wilson argues that this conclusion is unlikely to hold for modern machines because of their deep memory hierarchies [60]. Our results on such a system support Appel's analysis, although we find that an Appel-style collector runs inversely proportionally to the square of heap size.

7. Future Work

This paper addresses only individual object management, where all objects are allocated with `malloc` and freed with `free`. However, custom allocation schemes like regions can dramatically improve the performance of applications using explicit memory management [9, 31]. Regions are also increasingly popular as an alternative or complement to garbage collection [26, 27, 29, 54]. In future work, we plan to use our framework to examine the impact of the use of regions and a hybrid allocator, reaps [9], as compared to garbage collection.

The Lea allocator we use here places 8-byte object headers prior to each allocated object. These headers can increase space consumption and impair cache-level locality [24]. We plan to evaluate memory allocators like PHKmalloc [42] and Vam [24] that use BiBoP-style (big bag of pages) allocation and so avoid per-object headers. We intend to compare the virtual memory performance of explicit memory management with the bookmarking collector, which is specifically designed to avoid paging [32].

Finally, this paper examines only stop-the-world, non-incremental, non-concurrent garbage collectors. While these generally provide the highest throughput, they also exhibit the largest pause times. We would like to explore the effect on pause times of various garbage collectors relative to explicit memory managers, which also exhibit pauses. For example, the Lea allocator normally allocates objects in a few cycles, but occasionally empties and coalesces its quicklists. It also does a linear best-fit search for large objects. [2] To our knowledge, pauses caused by explicit memory management have never been measured or compared to garbage collection pauses.

8. Conclusion

This paper presents a tracing and simulation-based experimental methodology that executes unaltered Java programs as if they used explicit memory management. We use this framework to compare the time-space performance of a range of garbage collectors to explicit memory management with the Lea memory allocator. Comparing runtime, space consumption, and virtual memory footprints over a range of benchmarks, we show that the runtime performance of the best-performing garbage collector is competitive with explicit memory management when given enough memory. In particular, when garbage collection has five times as much memory as required, its runtime performance matches or slightly exceeds that of explicit memory management. However, garbage collection's performance degrades substantially when it must use smaller heaps. With three times as much memory, it runs 17% slower on average, and with twice as much memory, it runs 70% slower. Garbage collection also is more susceptible to paging when physical memory is scarce. In such conditions, all of the garbage collectors we examine here suffer order-of-magnitude performance penalties relative to explicit memory management.

We believe these results will be helpful both to practitioners and researchers. For practitioners, these results can guide their choice of using explicitly-managed languages like C and C++, or garbage-collected languages like Java or C#. As long as their applications will be running on systems equipped with more than three times as much RAM as required, then garbage collection is a reasonable choice. However, if these systems will have less RAM or if the application will be competing with other processes for physical memory, then practitioners should expect garbage collection to exact a substantial performance cost. This cost will be more pronounced for systems whose performance depends on maximizing the use of

[2] A newer version of the Lea allocator (2.8.2) uses tries to reduce the cost of this search.

memory, such as in-memory databases and search engines.

For researchers, these results should help guide their development of memory management algorithms. This study identifies garbage collection's key weaknesses as its poor performance in tight heaps and in settings where physical memory is scarce. On the other hand, in very large heaps, garbage collection is already competitive with or slightly better than explicit memory management.

9. Acknowledgments

Thanks to Steve Blackburn, Hans Boehm, Sam Guyer, Martin Hirzel, Richard Jones, Scott Kaplan, Doug Lea, Kathryn McKinley, Yannis Smaragdakis, Trevor Strohman, and to the anonymous reviewers for their helpful comments on drafts of this paper.

This material is based upon work supported by the National Science Foundation under Award CNS-0347339 and CISE Research Infrastructure Grant EIA-0303609. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

10. References

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

[2] 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.

[3] 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.

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

[5] A. W. Appel. Allocation without locking. *Software Practice and Experience*, 19(7), 1989. Short Communication.

[6] E. D. Beger. The hoard memory allocator. Available at `http://www.hoard.org`.

[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 *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, Nov. 2002. ACM Press.

[9] 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.

[10] 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.

[11] 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.

[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. Reducing garbage collector cache misses. In T. Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, Oct. 2000. ACM Press.

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

[15] 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.

[16] G. Colvin, B. Dawes, and D. Adler. C++ Boost Smart Pointers, Oct. 2004. Available at `http://www.boost.org/libs/smart_ptr/smart_ptr.htm`.

[17] S. P. E. Corporation. Specjbb2000. Available at `http://www.spec.org/jbb2000/docs/userguide.html`.

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

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

[20] 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.

[21] J. Dolby. Automatic inline allocation of objects. In *Proceedings of SIGPLAN'97 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Las Vegas, Nevada, June 1997. ACM Press.

[22] J. Dolby and Chien. An evaluation of object inline allocation techniques. In *OOPSLA'98 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Vancouver, Oct. 1998. ACM Press.

[23] J. Dolby and Chien. An automatic object inlining and its evaluation. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 345–357, Vancouver, June 2000. ACM Press.

[24] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the ACM SIGPLAN 2005 Workshop on Memory System Performance (MSP)*, Chicago, Illinois, June 2005.

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

[26] 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.

[27] 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.

[28] W. Gloger. Dynamic memory allocator implementations in linux system libraries. Available at `http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html`.

[29] 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.

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

[31] 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.

[32] M. Hertz and E. D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 143–153, June 2005.

[33] 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.

[34] 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.

[35] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In A. Diwan, editor, *ISMM'04 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Vancouver, Oct. 2004. ACM Press.

[36] 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.

[37] M. Hirzel, A. Diwan, and T. Hosking. On the usefulness of liveness for garbage collection and leak detection. In J. L. Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, volume 2072 of *Springer-Verlag*, Budapest, June 2001. Springer-Verlag.

[38] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *Proceeding of the ACM Conference on Object-Oriented Systems, Languages and Applications*, Vancouver, BC, Canada, Oct. 2004.

[39] 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.

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

[41] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996.

[42] P.-H. Kamp. Malloc(3) revisited. `http://phk.freebsd.dk/pubs/malloc.pdf`.

[43] 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.

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

[45] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[46] M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Washington, DC, June 2004. ACM Press.

[47] J. S. Miller and G. J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report AIM-1462, MIT AI Laboratory, Mar. 1994.

[48] G. Phipps. Comparing observed bug and productivity rates for java and c++. *Software Practice & Experience*, 29(4):345–358, 1999.

[49] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications*, Vancouver, BC, Canada, Oct. 2004.

[50] P. Savola. Lbnl traceroute heap corruption vulnerability. Available at `http://www.securityfocus.com/bid/1739`.

[51] R. Shaham, E. Kolodner, and M. Sagiv. Estimating the impact of liveness information on space consumption in Java. In D. Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 64–75, Berlin, June 2002. ACM Press.

[52] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, July 2003.

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

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

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

[56] 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).

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

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

[59] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings of*

International Workshop on Memory Management, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 Sept. 1992. Springer-Verlag.

[60] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 32–42, San Francisco, CA, June 1992. ACM Press.

[61] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the 2004 ACM SIGPLAN International Symposium on Memory Management*, Nov. 2004.

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