Generating Object Lifetime Traces with Merlin

MATTHEW HERTZ
University of Massachusetts, Amherst
STEPHEN M BLACKBURN
Australian National University
J ELIOT B MOSS
University of Massachusetts, Amherst
KATHRYN S McKINLEY
University of Texas at Austin
and
DARKO STEFANOVIĆ
University of New Mexico

Programmers are writing a rapidly growing number of programs in object-oriented languages, such as Java and C#, that require garbage collection. Garbage collection traces and simulation speed up research by enabling deeper understandings of object lifetime behavior and quick exploration and design of new garbage collection algorithms. When generating perfect traces, the *brute-force* method of computing object lifetimes requires a whole-heap garbage collection at every potential collection point in the program. Because this process is prohibitively expensive, researchers often use *granulated* traces by collecting only periodically, e.g., every 32K bytes of allocation.

We extend the state of the art for simulating garbage collection algorithms in two ways. First, we develop a systematic methodology for simulation studies of copying garbage collection and present results showing the effects of trace granularity on these simulations. We show that trace granularity often distorts simulated garbage collection results compared with perfect traces. Second, we present and measure the performance of a new algorithm called Merlin for computing object lifetimes. Merlin timestamps objects and later uses the timestamps of dead objects to reconstruct when they died. The Merlin algorithm piggybacks on garbage collections performed by the base system. Experimental results show that Merlin can generate traces over two orders of magnitude faster than the brute-force method which collects after every object allocation. We also use Merlin to produce visualizations of heap behavior that expose new object lifetime behaviors.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection)*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: Garbage collection, trace design, object lifetime analysis, trace generation

This work is supported by NSF grants CCR-0219587, ITR CCR-0085792, EIA-0218262, EIA-9726401, EIA-030609, EIA-0238027, EIA-0324845, DARPA F33615-03-C-4106, Hewlett-Packard gift 88425.1, Microsoft Research, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

1. INTRODUCTION

While languages such as LISP and Smalltalk have always used garbage collection (GC), the dramatic increase in the number of programs written in Java, C#, and other modern languages has prompted a corresponding surge in GC research. A number of these studies use garbage collection traces and simulations to examine the effectiveness of new GC algorithms [Hirzel et al. 2003; Stefanović et al. 1999; Zorn 1989]. Other research uses traces to tune garbage collection via profile feedback [Blackburn et al. 2001; Cheng et al. 1998; Shaham et al. 2000; Ungar and Jackson 1992]. A perfect trace for garbage collection includes the birth and death time of all objects, measured in bytes allocated. (The memory management community uses memory rather than operations to measure lifetime.) Computing perfect lifetimes can be a very time-consuming process. For instance, a *tracing* collector must determine all the reachable objects in the heap at every allocation by computing reachability from the stacks, global variables, and local variables:

$$\sum_{i=1}^{n} |\text{live objects}| \text{ at } a_i$$

where n is the number of objects the program allocates, and a_i is an allocation event. This cost is prohibitive even for modest programs that allocate on the order of 100MB and have an average live size on the order of 10MB, such as the widely used SPECjvm98 benchmarks [SPECjvm98 1998]. On current processors, many of these programs execute in under a minute, but brute-force trace generation takes over 3 months. Costs are similar even for a reference counting collector because it also requires a form of tracing to handle cycles. While future technology advances will reduce this time, these same trends inspire programmers to use larger data sets.

To avoid this cost, previous research often uses *granulated* traces which estimate object lifetimes periodically (e.g., after every k bytes of allocation). However, researchers have not studied the effects of granularity on the accuracy of garbage collection simulations or measures computed from them. While Zorn and Grunwald [1992] examined better methods of approximating traces, no one has studied what effects these approximations have either. In this work, we run simulations using granulated traces on a variety of copying garbage collection algorithms and metrics for evaluating them. The results demonstrate that granulated traces can produce significantly different results and thus that conclusions drawn from research based on simulations of granulated traces may be problematic.

We introduce the Merlin object lifetime algorithm which efficiently computes object lifetimes. The Merlin algorithm timestamps live objects when they lose an incoming reference and later uses the timestamps to reconstruct the time at which the object became unreachable. By using timestamps rather than tracing to identify the time of death, the new algorithm does not require frequent collections nor does it require whole-heap collections. Rather, it makes use of those collections that the system (virtual machine) normally performs to identify *which* objects have died, and then uses the timestamps to identify *when* they died. Ordering the dead objects from the latest timestamp to the earliest, the algorithm works from the current collection time backwards. Merlin thus only processes each object once to compute its death time after it knows that the object is dead. Merlin's execution time is proportional to the total allocations plus the number of times each object loses an incoming reference, m.

$$\sum_{i=1}^{n} |\text{object allocated}| \text{ at } a_i + \sum_{j=1}^{m} r_j$$

Experimental results on SPECjvm98 and other programs show that in practice the Merlin algorithm can improve performance by more than a factor of 800 over brute-force tracing, though it is 70 to 300 times slower than an untraced program. Merlin thus makes producing perfect traces much more attractive.

This paper extends our prior work [Hertz et al. 2002a] which introduced the Merlin algorithm with (1) a better description of the Merlin algorithm, (2) qualitative as well as quantitative analysis of the effects of trace granulation, (3) a more detailed performance analysis, (4) an algorithm that uses Merlin to generate granulated traces, and (5) results and analysis of Merlin heap lifetime visualizations.

As a demonstration of the usefulness of perfect traces, we present heap lifetime visualizations. Stefanović [1999] used brute-force traces to produce similar visualizations for a set of small programs to explore garbage collection performance. By reducing the time to generate traces, we examine here much larger and more realistic programs. These graphs reveal lifetime behaviors and trends that enhance the understanding of object lifetimes and design of garbage collection algorithms, and we offer some analysis here.

The remainder of the paper analyzes the effects of trace granularity on garbage collection simulation fidelity for a number of collectors, introduces the Merlin trace generation algorithm, and describes additional uses of lifetime traces. Section 2 gives some background on garbage collection, lifetime traces, and trace granularity. Section 3 describes our experimental methodology for analyzing the effects of trace granularity. Section 4 and 5 present and discuss the results of our granularity analysis. Section 6 introduces the Merlin trace generation algorithm and describes how it improves on the previous approaches. Section 7 presents and analyzes results from the new algorithm. Section 8 presents additional uses of perfect lifetime traces. Finally, Section 9 presents related work and Section 11 summarizes this study.

2. BACKGROUND

This section explains three background concepts: *garbage collection (GC)*, *garbage collection traces and their use in simulations*, and *garbage collection trace granularity*.

2.1 Garbage Collection

Garbage collection automates the reclamation of heap objects that are no longer needed [Jones and Lins 1996]. While a wide variety of systems use garbage collectors, we assume a system that uses an implicit-free environment, i.e., a system that defines an explicit `new` command for object allocation, but not a `free` command. Since garbage collectors cannot know which objects the program will use in the future without additional information, collectors *conservatively* approximate liveness with reachability; all reachable objects are assumed live, and all unreachable objects may be reclaimed since it is not possible for the program to access them again.[1] To determine reachability, a collection begins at a program's roots. The *roots* contain all the pointers from outside of the heap into the heap,

[1]Systems with finalization must maintain pointers to these objects until they perform the finalization operations, at which point the collector can reclaim them.

such as the program stacks, static (global) variables, and local variables in the current procedure. The collector then finds the live objects by finding all objects in the transitive closure over the points-to (reachability) relationship.

Whole-heap collectors compute the reachability of every object and remove all unreachable objects on every collection. Many collectors (e.g., generational collectors [Lieberman and Hewitt 1983; Ungar 1984]) often collect part of the heap, limiting the work at a collection. Because the collector reclaims only unreachable objects, it must conservatively assume that the regions of the heap not examined contain only live objects. If objects in the unexamined region point to objects in the examined region, the target objects must remain in the heap. Collectors typically use *write barriers* to find pointers into the collected region. A write barrier is code executed by the system in conjunction with each pointer store operation. A write barrier typically tests if the pointer target is in a region that will be collected before the region containing the pointer source, and records such pointers in some data structure.

2.2 Copying Garbage Collection Algorithms

We use four copying GC algorithms for evaluating trace granularity: a semi-space collector, a fixed-size nursery generational collector [Lieberman and Hewitt 1983; Ungar 1984], a variable-sized nursery generational collector [Appel 1989], and an Older-First collector [Stefanović et al. 1999; Stefanović et al. 2002]. We briefly describe each of these here and refer the reader to previous work for more details [Jones and Lins 1996].

A semi-space collector (SS) allocates into *From* space using a bump pointer. A bump pointer defines a boundary between allocated and free memory within a larger contiguous region. It allows simple and efficient allocation by incrementing the pointer by the size of the allocated object. When SS runs out of space, it collects this entire space by finding all reachable objects and copying them into a second space, called *To* space. The collector then reverses *From* and *To* space and continues allocating. Since all objects in *From* space may be live, it must conservatively reserve half the total heap for the *To* space, as do the generational collectors that generalize this collector.

A fixed-size nursery (FG) two-generation collector divides the *From* space of the heap into a nursery and an older generation. It allocates into the nursery. When the nursery is full, it collects the nursery and copies the live objects into the older generation. It repeats this process until the older generation is also full. It then collects the nursery together with the older generation and copies survivors into the *To* space of the older generation.

A variable-size nursery two-generation collector (VG) also divides the *From* space into a nursery and an older generation, but does not fix the boundary between them. In the steady state, the nursery is some fraction of *From* space. When the nursery is full, VG copies live objects into the older fraction. The new nursery size is reduced by the size of the survivors. When the nursery becomes too small, VG collects all of *From* space. The obvious generalization of these variants to n generations apply.

The Older-First collector (OF) organizes the heap in order of object age. It collects a fixed-size window that slides through the heap from older to younger objects. When the heap is full in the steady state, OF collects the window, returns the free space to the nursery, compacts the survivors, and then positions the window for the next collection at objects just younger than those that survived. If the window bumps into the allocation point, OF resets the window to the oldest end of the heap. OF need only reserve space the size of one window for collection (as opposed to half the heap for the other algorithms).

2.3 Garbage Collection Traces and Simulations

Given the typical difficulty of implementing a known garbage collector, implementing and debugging *new* garbage collection algorithms and optimizations can be a daunting process. Especially when a collector is designed to take advantage of new or unavailable hardware (e.g., a 64-bit address space [Stefanović 1999]) or compiler optimizations (e.g., [Hirzel et al. 2003]), researchers have often used simulators to enable rapid prototyping and evaluation before investing in a full implementation. By loosening restrictions on the knowledge available to a collector and what a GC algorithm may do, simulators are also useful for oracle-driven limit studies [Hertz and Berger 2004; Stefanović 1999]. A final value of simulators is their ability to support evaluations of a single implementation of a garbage collector with input from any number of different programming languages or virtual machines. As a portion of our study is an examination of simulator fidelity, here we provide the reader a basic description of GC simulators and the traces that drive them.

A *garbage collection trace* is a chronological record of every object allocation, heap pointer update, and object death (object becoming unreachable) over the execution of a program. Following common practice, traces measure time in bytes of allocation and not number of operations. Each event includes the information that a memory manager needs for its processing. Processing *object allocation* records requires an identifier for the new object and the object's size; *pointer update* records include the object and field being updated and the new value of the pointer; *object death* records indicate which object became unreachable. These events constitute the minimum amount of information that GC simulations need. Depending on the algorithm and detail of simulation, other events, such as procedure entry and exit, field reads, or root pointer enumeration may also be necessary and/or useful.

Simulators then apply one or more GC algorithms and optimizations to a given program trace. The trace must contain all the information that a garbage collection algorithm would actually use in a live execution and all of the events upon which the collector may be required to act, independent of any specific GC implementation. Traces do not record all aspects of program execution, but only those which are needed to recreate collector performance accurately. While even single-threaded garbage collection may not be deterministic, simulations return deterministic results since the trace file is fixed. With representative trace files, researchers can rely upon these results, making simulation attractive and accurate traces critical.

GC trace generators must be integrated into the memory manager of the interpreter or virtual machine in which the program runs. If the program is compiled into a stand-alone executable, the compiler back end must generate trace generation code in addition to the ordinary memory management code at each object allocation point and pointer update. The generator can log pointer updates by instrumenting pointer store operations; this instrumentation is particularly easy if the language and GC implementation use write barriers, since the generator can simply piggyback its instrumentation onto existing code.

The common brute-force method of computing object lifetimes determines reachability by performing a whole-heap GC after every allocation. The brute-force method incurs the expense of collecting the *entire* heap prior to allocating *each* object. In current technology, brute-force accurate trace generation for a small micro-benchmark at all allocation points takes days; traces of simple single-threaded programs from SPECjvm98 can require several months.

Even though objects may die between allocations, the memory management literature uses bytes of allocation to measure object lifetimes. Many GC algorithms only trigger collection when they need additional space in the heap, i.e., immediately before allocating a new object, and thus this measurement is fully accurate. GC algorithms such as deferred reference counting [Deutsch and Bobrow 1976; Blackburn and McKinley 2003] can initiate collections at other GC safe points as well, such as a procedure call or return. A GC *safe point* requires that the garbage collector correctly enumerate all root pointers. Although we do not consider these additional points here, such a trace would include markers for all such points, and a brute-force trace generator would perform additional reachability analyses at all these points as well.

2.4 Garbage Collection Trace Granularity

To reduce the prohibitive cost of brute-force trace generation, previous work often performs object lifetime analysis only periodically, e.g., after every k bytes of allocation. It also guarantees the trace to be accurate only at those specific points; the rest of the time the trace may overestimate the set of live objects. For correctness, any simulation must assume that objects become unreachable only at the accurate points. The *granularity* of a trace is the period between these moments of accurate death knowledge.

3. EFFECTS OF TRACE GRANULARITY

This section evaluates the effects of trace granularity on simulation accuracy using copying garbage collectors as the set of client algorithms. We first describe our simulator and programs. To our knowledge, all previous GC simulation work (including our own) neglected to consider precisely the question of information accuracy at different points in a trace with a given granularity. We explore a variety of methods for handling granularity in simulation. We find that although some methods yield better results than others, all methods introduce inaccuracies into GC algorithm simulations, even with relatively modest trace granularity.

3.1 Simulator Suite

For our trace granularity experiments, we used *gc-sim*, a GC simulator suite from the University of Massachusetts with front-ends for Smalltalk and Java traces. In our simulator, we implemented four different GC algorithms: SS, FG, VG, and OF, as described in Section 2.2. The first three collectors are in widespread use. For each collector, we use a number of fixed heap sizes to explore the inherent space-time trade off in garbage collection. We simulate eight different *From* space sizes, from 1.25 to 3 times the maximum size of the live objects within the heap, at 0.25 increments. For FG and VG we simulated each heap size with five different nursery sizes, and for OF, five window sizes. These latter parameters ranged from $\frac{1}{6}$ to $\frac{5}{6}$ of *From* space, in $\frac{1}{6}$ increments.

3.2 Granularity Schemes

We designed and implemented four different schemes to handle trace granularity. Each of these schemes is independent of the simulated GC algorithm. By affecting *when* the collections occur, they explore the limits of trace granularity.

3.2.1 *Unsynchronized.* When we began this research, our simulator used this naive approach to handling trace granularity: it did nothing. We call this method *Unsynchronized*. Unsynchronized simulations allow a GC to occur at any time in the trace; simulated

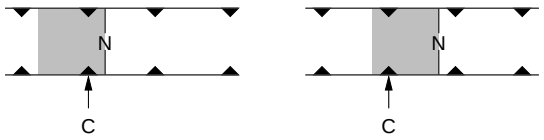
collections occur at the natural collection points for the garbage collection algorithm (such as when the heap or nursery is full). This scheme allows the simulator to run the algorithm as it is designed and does not consider trace granularity when determining when to collect. Unsynchronized simulations may treat an object as reachable because the object death record was not yet reached in the trace, even though the object is unreachable. However, they allow a GC algorithm to perform collections at their natural points, unconstrained by the granularity of the input trace.

3.2.2 *Synchronized Schemes.* Three other schemes, which we call *Synchronized*, simulate collections only at those points in the trace with accurate knowledge of unreachable objects. The schemes check if a GC is needed, or will be needed soon, only at the accurate points and simulate a collection only at these points. Figure 1 shows how each of the Synchronized schemes makes collection decisions. In each of these figures, the solid line labeled N is the natural collection point for the algorithm. The triangles denote points with perfect knowledge and the shaded region indicates one granule of the trace. Each scheme performs the collection at the point in the trace with perfect knowledge within the shaded region. This point is shown by the arrow labeled C.

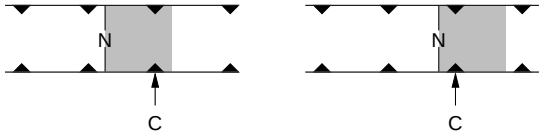
SyncEarly. The first scheme we call *SyncEarly*. Figure 1(a) shows how SyncEarly decides when to collect. If, at a point with perfect knowledge, the simulator determines that the natural collection point will be reached within the following granule of the trace, SyncEarly forces a collection. SyncEarly always performs a collection *at or before* the natural point is reached. Even assuming there are no effects from trace granularity, SyncEarly simulations may still perform extra garbage collections, e.g., when the last natural collection point occurs between the end of the trace and what would be the next point with perfect knowledge. But SyncEarly ensures that the simulated heap will never grow beyond the bounds it is given.

SyncLate. The second scheme is *SyncLate*. Figure 1(b) shows how SyncLate decides when to collect. At a point with perfect knowledge, if SyncLate computes that the natural collection point occurred within the preceding granule of the trace, SyncLate invokes a garbage collection. SyncLate collects *at or after* the natural point is reached. SyncLate simulations may collect too few times, e.g., when the last natural collection point occurs between the last point with perfect knowledge and the end of the trace. SyncLate allows the heap and/or nursery to grow beyond their nominal bounds between points with perfect knowledge, but enforces the bounds whenever a collection is completed.

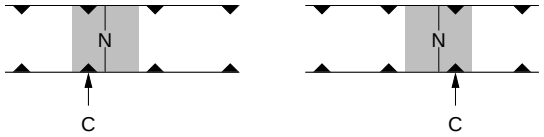
SyncMid. The last Synchronized scheme is *SyncMid*. Figure 1(c) shows how SyncMid decides when to collect. SyncMid forces a GC invocation at a point with perfect knowledge if a natural collection point is within half of a granule in the past or future. SyncMid requires a collection at the point with perfect knowledge *closest* to the natural collection point. SyncMid simulations try to balance the times they invoke collections too early and too late to achieve results close to the average. SyncMid simulations may, like SyncEarly, perform more or may, like SyncLate, perform fewer garbage collections. Between points with perfect knowledge, SyncMid simulations may also require the heap and/or nursery to grow beyond their nominal bounds. However, heap bounds are enforced immediately following a collection.



(a) SyncEarly



(b) SyncLate



(c) SyncMid

Fig. 1. These figures show points with perfect knowledge (triangles) and the natural collection point (N) (where the collector runs out of space). The shaded region highlights a granule-sized region of the trace and contains the collection point (C) where the Synchronization scheme will actually simulate a collection.

4. TRACE GRANULARITY RESULTS

Using our simulator suite, we performed a number of experiments to determine if trace granularity affects garbage collection simulations. We examined the performance of each combination of collector and trace granularity scheme described above on a variety of Java and Smalltalk benchmarks across several commonly used GC metrics. Our results show that even small trace granularities produce differences in simulator results and that algorithm choice could help limit, but not eliminate, this problem. The remainder of this section describes in more detail the metrics we considered, the experiments we performed, and presents an overview of these results.

4.1 GC Simulation Metrics

Each GC simulation measures the following: the number of simulated collections, the *mark*/*cons* ratio, the number of *write-barrier stores*, and the *space-time product*. For a given trace, these metrics are deterministic.

The mark/cons ratio is the number of bytes that the collector copied (*marked*) divided by the number of bytes allocated (*cons'ed*, in LISP terminology). The ratio approximates the amount of work done by a collector. Higher mark/cons ratios suggest an algorithm will need more time, because it must process and copy more objects and more bytes.

Another metric we report is the number of write-barrier stores during a program run. Since many garbage collectors do not collect the entire heap, they use a write barrier to find pointers between collection regions (as discussed in Section 2.1). The write barrier instruments pointer store operations to determine if the pointer crosses from one collec-

tion region to another. Depending on the GC algorithm, pointers crossing particular region boundaries in particular directions must be recorded ("remembered") so that they can subsequently be examined at GC time; these stores are called *write-barrier stores*. The number of pointer stores, and the cost to instrument each of these, does not vary in a program run, but the number of write-barrier stores varies between GC algorithms at run time and affects their performance.

We measure the space-time product, computing the sum of the number of bytes used by objects within the heap at each allocation point multiplied by the size of the allocation, i.e., the integral of the number of bytes used by objects within the heap with respect to bytes of allocation (time). Since the number of bytes allocated does not vary between algorithms, this metric captures how well an algorithm manages issues such as fragmentation throughout the program execution.

None of these metrics is necessarily sufficient in itself to determine how well an algorithm performs. Algorithms can perform better in one or more of the metrics at the expense of another. The importance of considering the totality of the data can be seen in models developed that combine the data to determine the total time each algorithm needs [Stefanović et al. 1999].

4.2 GC Traces

We used 15 GC traces in this study. Nine of the traces are from the Jikes RVM [Alpern et al. 1999; Alpern et al. 2000], a compiler and run-time system for Java in which we implemented our trace generator. Because it is written in Java, these traces include heap allocations from both the application and the Jikes RVM. The nine Java traces are: bloat-bloat (Bloat [Nystrom 1998] using its own source code as input), two different configurations of Olden health (5 256 and 4 512) [Cahoon and McKinley 2001], and compress, jess, raytrace, db, javac, and jack from SPECjvm98 [SPECjvm98 1998]. We also have six GC traces that we generated previously using the University of Massachusetts Smalltalk Virtual Machine. The Smalltalk traces are: lambda-fact5, lambda-fact6, tomcatv, heapsim, tree-replace-random, and tree-replace-binary [Hosking et al. 1992; Stefanović et al. 1999]. More information about the programs appears in Table I. These programs are widely used in the garbage collection literature.

We implemented a filter that accepts a perfect trace and target value, and outputs the trace with the targeted level of granularity. From our perfectly accurate traces for each of the programs we generated 7 granulated versions of each trace with trace granularities ranging from 1KB to 64KB. To examine the effects of very large trace granularity, we use granularities of 512KB, 1024KB and 2048KB. We selected the minimum 1KB granularity to be smaller than most prior traced-based research but still large enough to provide some savings in trace generation time. Table II shows an example of the simulator output where $|GC|$ is the number of collections, *xcopy b* is the number of excess copied bytes (unreachable bytes copied), and *mut. i/s* is the number of write-barrier stores that occur during program execution.

4.3 Analysis

We began our experiments by simulating all combinations of benchmark, trace granularity, granularity scheme, GC algorithm, and *From* space and nursery (window) size, recording the four metrics from above for each combination. This provided us with 600 OF, VG, and FG simulation runs and 120 SS simulation runs for each combination of trace granularity

Table I. Traces used in the experiment. Sizes are expressed in bytes.

Program	Description	Max. Live	Bytes Alloc	Objs Alloc
bloat-bloat	Bytecode-Level Optimization and Analysis Tool 98 using its own source code as input	3 207 176	164 094 868	3 653 255
Olden Health (5 256)	Columbian health market simulator from the Olden benchmarks, recoded in Java	2 337 284	14 953 944	662 395
(4 512)	A smaller run of Olden health	1 650 444	9 230 756	353 094
SPEC _201_compress	Compresses and decompresses 20MB of data using the Lempel-Ziv method	8 144 188	120 057 332	138 931
SPEC _202_jess	Expert shell system using NASA CLIPS	3 792 856	321 981 032	8 575 988
SPEC _205_raytrace	Raytraces a scene into a memory buffer	5 733 464	154 028 396	6 552 000
SPEC _209_db	Performs series of database functions on a memory resident database	10 047 216	85 169 104	3 314 278
SPEC _213_javac	Sun's JDK 1.0.4 compiler	11 742 640	274 573 404	8 096 562
SPEC _228_jack	Generates a parser for Java programs	3 813 624	322 274 664	8 107 004
lambda-fact5	Untyped lambda calculus interpreter evaluating 5! in the standard Church numerals encoding	25 180	1 111 760	53 580
lambda-fact6	Untyped lambda calculus interpreter evaluating 6! in the standard Church numerals encoding	54 700	4 864 988	241 864
tomcatv	Vectorized mesh generator	126 096	42 085 496	3 385 900
heapsim	Simulates a garbage collected heap	549 504	9 949 848	764 465
tree-replace-random	Builds a binary tree then replaces random subtrees at a fixed height with newly built subtrees	49 052	2 341 388	121 588
tree-replace-binary	Builds a binary tree then replaces random subtrees with newly built subtrees	39 148	818 080	34 729

Table II. Simulator output from a fixed-sized nursery (FN) simulation of Health (4, 512). The top lines are the metrics after six collections, when the differences first become obvious; the bottom lines are the final results of the simulation.

|GC|	alloc b	copy b	xcopy b	garbage b	mark/con	xcopy/copy	mut. i/s
6	5 221 236	1 098 480	268 088	3 770 048	0.210 387	0.244 054	14 243
10	9 230 756	1 552 152	284 404	6 622 732	0.168 150	0.183 232	40 675

(a) Perfect Trace

|GC|	alloc b	copy b	xcopy b	garbage b	mark/con	xcopy/copy	mut. i/s
6	4 787 328	1 443 608	355 768	2 824 328	0.301 548	0.246 444	11 644
11	9 230 756	2 007 252	375 464	6 392 528	0.217 453	0.187 054	41 949

(b) SyncMid With 1KB Granularity

and granularity scheme. From this large population of data we perform qualitative and statistical analyses of the results.

We exclude the following sets of simulations that do not exercise the memory system well, and/or yield incomplete information. We remove simulations with fewer than 10 garbage collections. We remove simulations where the trace granularity equaled 50% or more of the simulated *From* space size, since trace granularity would obviously impact these results. We also excluded simulations where either the simulation using the perfect trace or granulated trace could not run within the given heap size. For example, the heap size was too small to accommodate imperfect collection due to late or early synchronization.

For our statistical study, the number of experiments remaining at the 1KB granularity was about 90 for SS, 200 for VG, 250 for FG, and 425 for OF. The number of valid simulations does not vary by more than 2%–3% until the 32KB granularity. At the 32KB granularity, there are 20% fewer valid simulations. The numbers continue to drop as the granularity increases; by the 2048KB granularity there are fewer than half the number of usable simulations as at the smallest granularity.

We analyze the data as follows to reveal if trace granularity affects GC simulations and if it does, at what granularities do differences appear. To aggregate the data, we normalize the granulated trace simulation results to the results of an identically configured simulation using a perfect trace. We use the logarithm of this ratio so that values twice as large and half as much average to 1. To provide a qualitative analysis of the effects of trace granularity, we compared the normalized simulator result of each metrics versus the granularity of the trace being simulated. We found that expressing the trace granularity by different methods helps show different causes of these errors. The three graphs in Figure 2 all show normalized mark/cons values for SyncMid with VN, but using three different methods of expressing the trace granularity.

Figure 2(a) plots the relative mark/cons ratio as a function of the trace granularity. This graph reveals that normalized simulator results for this metric range from 1.6 to 0.5 at even the smallest granularity with the spread increasing at larger granularities. From this graph, however, it is difficult to determine how much of this behavior is due to the relative size of the trace. Figure 2(b) shows the same results as a function of trace granularity relative to the maximum live size of the trace. It separates the data, and shows the range of errors for the mark/cons ratio that can occur at a single heap size. Figure 2(c) expresses the trace granularity relative to the size of the *simulated* heap and is a better predictor of error, but still does not place a tight bound on the deviations. Figure 3 plots relative trace granularity for VN using SyncEarly, SyncLate, and Unsynchronized. In comparison to Figure 2(c), SyncMid is as good as or better than the other granulation schemes.

While these results are helpful for understanding when errors occur, statistical analysis is needed to determine (1) if measures of trace granularity are simulation-dependent, (2) if there exists some granularity size that could yield acceptable error at trace generation time; and (3) if even when relative trace granularity is quite small, we will continue to see a sizable error in simulated results.

For a more definitive answer as to whether trace granularity affects GC simulations, we performed two-tailed t-tests on the aggregated results for all metrics. A two-tailed t-test determines if the difference between the actual mean (e.g., the result from the granulated trace) and expected mean (e.g., the result if trace granularity had no effect on the simulator results) is the result of natural variance in the data or the effects of trace granulation are larger than can be explained by normal variance. Following convention, we consid-

Table III. Smallest granularity (in KB) at which each metric becomes significantly different, by simulation method and collector. Differences were tested using a two-tailed t-test at the 95% confidence level (p = 0.05).

	Unsynced				SyncMid				SyncEarly				SyncLate			
	SS	FG	VG	OF	SS	FG	VG	OF	SS	FG	VG	OF	SS	FG	VG	OF
Mark/Cons	1	1	1	1	none	1	none	1	1	1	4	4	1	8	16	4
Space-Time	1	1	1	1	none	1	2	1	1	1	1	1	1	1	1	2
|GC|	1	1	16	1	none	1	16	1	1	1	4	4	1	1	1	1
WB Stores	n/a	16	16	1	n/a	32	16	none	n/a	2	8	4	n/a	2	4	8

Table IV. Smallest granularity (in KB) at which each metric becomes significantly different, by simulation method and collector. Differences were tested using a two-tailed t-test at the 95% confidence level (p = 0.05). This table considers only data from traces with a maximum live size of 2MB or more.

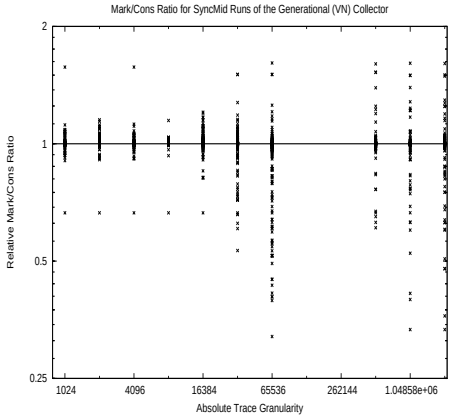
	Unsynced				SyncMid				SyncEarly				SyncLate			
	SS	FG	VG	OF	SS	FG	VG	OF	SS	FG	VG	OF	SS	FG	VG	OF
Mark/Cons	1	1	4	32	none	512	none	64	8	512	none	8	32	1	1024	16
Space-Time	4	1	512	1	1	1	512	32	1	1	512	2	16	1	512	512
|GC|	32	1	512	16	none	1	512	1024	64	1	512	8	16	1	64	8
WB Stores	n/a	512	2098	512	n/a	16	1	none	n/a	32	1	8	n/a	16	1024	16

ered only differences at the 95% confidence level or higher ($p \leq 0.05$) to be statistically significant (more than the result of the random variations observable in the simulator results). When the t-test finds that the granulated results are significantly higher at the 95% confidence level, it signifies that were the experiment repeated with similarly granulated traces, 95% of repeated experiments will also find that the granulated trace mean will be larger than results generated from perfect traces [Natrella 1963]. A similar argument exists for results that the t-test determine are significantly lower. Table III shows the smallest granularity, in Kbytes, at which we observe a statistically significant difference for each combination of collector, metric, and simulation method. It includes the mark/cons ratio, *Space-Time*–which measures fragmentation, |GC|–the number of collections, and *WB Stores*–the number of pointers the write barrier must record for incremental collection (i.e., older to younger pointers in FG and VG, and cross increment pointers in OF). Section 4.1 describes these in more detail.

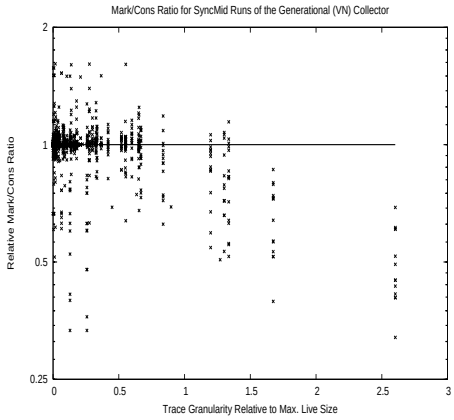
Programs with smaller *From* space and nursery (window) sizes will obviously be more sensitive to trace granularity. Just as we removed simulations where the granularity was over half of *From* space size, we re-ran our analysis using only those traces that, at some point, had enough live objects to equal the largest trace granularity. The excluded programs are small enough that a trace generator using the brute-force method of lifetime analysis can generate perfect traces in under 8 hours. The traces remaining in this analysis are those for which tracing using the brute-force method would need to generate granulated traces. The number of remaining simulations ranged from around 40 (for SS) to around 220 (for OF) at the 1KB granularity and does not vary by more than 1 or 2 until the 2048KB granularity where the counts of the OF and all Unsynchronized simulations decrease by about 10%. The results of this analysis appear in Table IV.

5. TRACE GRANULARITY DISCUSSION

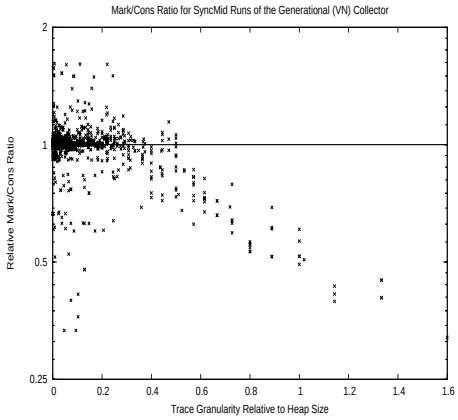
The data in Table III are quite revealing about the effects of trace granularity and the usefulness of the different schemes in handling granulated traces. While Figure 2(a) shows that there can be a considerable range of errors, Table III shows that this still isn't enough to



(a) Mark/cons results for SyncMid runs of VN by trace granularity. At even the smallest granularities, the errors can be quite large.



(b) Mark/cons results for SyncMid runs of VN by ratio of trace granularity to maximum live size.



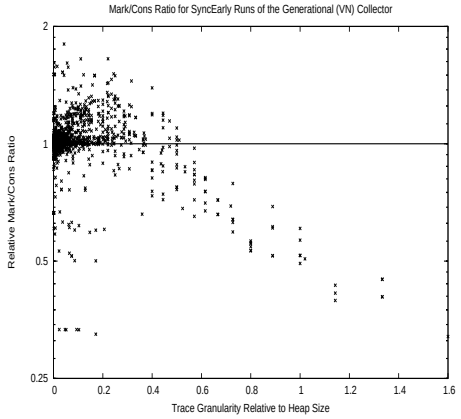
(c) Mark/cons results for VN by ratio of trace granularity to the simulated heap size. Shows larger relative granularities can cause smaller simulated mark/cons values.

Fig. 2. Qualitative analyses of the effects of trace granularity on simulator fidelity of mark/cons measurements for runs of VN using SyncMid. While relatively large errors occur at even the smallest trace granularities, patterns emerge when the results are plotted against the ratio of trace granularity versus simulated heap size.

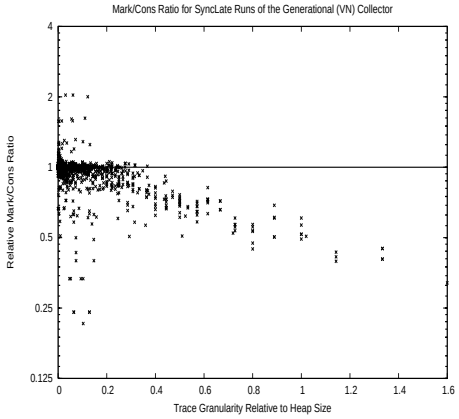
establish statistically significant distortions. For a majority of the metrics, however, a granularity as fine as one kilobyte is enough to cause this distortion. Clearly, trace granularity significantly affects the simulator results.

5.1 Unsynchronized Results

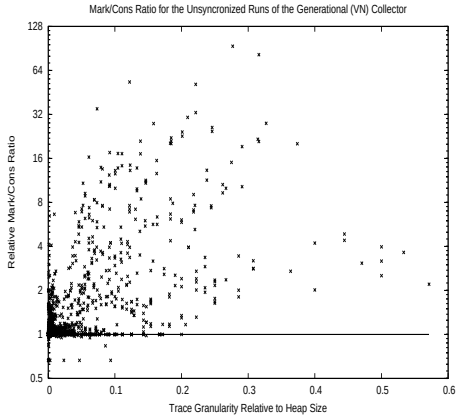
Unsynchronized collections dramatically distort simulation results. In Table III, two collectors (SS and OF) have statistically significant differences for every metric at the 1KB granularity. In both cases, the granulated traces copied more bytes, needed more col-



(a) Mark/cons results for SyncEarly runs of VN versus the ratio of trace granularity to simulated heap size. While the results initially tend to be too high, at the largest relative granularity the results are too low.



(b) Mark/cons results for SyncLate runs of VN versus the ratio of trace granularity to simulated heap size. These results always tend to be too low, though this worsens as the relative granularity increases.



(c) Mark/cons results for Unsynchronized runs of VN by trace granularity relative to simulated heap size. This graph shows the largest set of distortions for VN.

Fig. 3. Qualitative analyses of the effects of trace granularity on simulator fidelity for measurements of mark/cons on VN. At even the smallest granularities, there are wide ranges in simulator results.

lections, and their heaps were consistently fuller. For both collectors the differences were actually significant at the 99.9% confidence level or higher ($p \leq 0.001$), meaning we would expect similar results in 999 out of 1000 experiments. The generational collectors did not fare much better. VG and FG simulations using traces with only 1KB of granularity averaged 2.8% and 5.0% higher mark/cons ratios than with perfect traces, respectively. As one would expect, these distortions grew with the trace granularity. In Unsynchronized simulations, collections may come at inaccurate points in the trace; the garbage collector must process and copy objects that are reachable only because the trace has not reached the

next set of death records. Once copied, these objects increase the space-time product and cause the heap to be full sooner, and thus require more frequent GCs. At the 16KB granularity, FG averaged "only" 2.0% more collections—the other collectors averaged from 6.9% (VG) to 10.5% (SS) more. As these incorrectly promoted objects cause needless promotion of the objects to which they point, this process snowballs so that even small granularities quickly produce significant differences. Only the number of write-barrier stores for the generational collectors and the number of collections required for VG are not immediately affected. There are not significantly more pointers from the older generation to the nursery because Unsynchronized collections tend to incorrectly promote objects that are unreachable and cannot be updated.

We expect simulations using larger heaps to be less affected by these issues. The results in Table IV show that this hypothesis is true. The space-time product and mark/cons results for SS show that objects are staying in the heap longer. For VG simulations, however, we do not see a significant increase in the number of collections (at 16KB granularity, these simulations average only 0.09% more collections); the extra objects require the collector to perform more whole-heap collections and not just nursery collections. Therefore each collection does more work: a conclusion validated by the significantly higher mark/cons ratio (at 16KB granularity VG's mark/cons ratio is 15.7% greater on average than perfect simulation). Irrespective of the collection algorithm, Unsynchronized simulations clearly distort the results. This finding suggests that trace file formats should clearly label the points in the trace with perfect knowledge.

5.2 Synchronized Results

Synchronized simulations tend to require slightly higher granularities than Unsynchronized before producing significant distortions. As can be seen in Table III, every Synchronized scheme significantly distorts the results for each metric for at least one collector and at least one metric for each collector. Examining the results from Table III and Table IV reveals a few patterns. Considering all the traces, SyncEarly and SyncLate still produce differences from simulations using perfect traces, but require slightly larger trace granularities than Unsynchronized before the differences become statistically significant. SyncMid has several cases where significant distortions do not appear, but this result is both collector- and metric-dependent. In addition, there are still statistically significant distortions when using traces with granularities as small as 1KB. In Table IV, which considers only traces with larger maximum live sizes, Synchronized simulations provide better estimates of the results from simulating perfect traces. There still exist significant differences at fairly small granularities, however.

Because Synchronized simulations affect only when the collections occur, they do not copy unreachable objects merely because the object death record has not been reached. Instead, adjusting the collection point causes other problems. Objects that are allocated and those whose death records should occur between the natural collection point and the Synchronized collection point are initially affected. Depending on the Synchronized scheme, these objects may be removed from the heap or processed and copied earlier than in a simulation using perfect traces. Once the heap is in error (containing too many or too few objects), it is possible for the differences to be compounded as the Synchronized simulation may collect at points even further away (and make different collection decisions) than the simulation using perfect traces. Just as with Unsynchronized simulations, small initial differences can snowball.

SyncEarly. SyncEarly simulations *tend* to decrease the space-time products and increase the number of collections, write-barrier stores, and mark/cons ratios versus simulations using perfect traces. While generally true, FG contradicts this trend, which produces a higher space-time product at smaller granularities. Normally, FG copies objects from the nursery because they have not had time to die before collection. SyncEarly exacerbates this situation, collecting even earlier and copying more objects into the older generation than similar simulations using perfect traces. At even 1KB of granularity, the average FG simulation's space-time product is more than 1.0% larger than identical simulations using perfect traces considering all experiments and just those with larger live sizes. As trace granularity grows, however, this result disappears (the simulations still show significant distortions, but in the expected direction) because the number of points in the trace with perfect knowledge limits the number of possible GCs.

SyncLate. In a similar, but opposite manner, SyncLate simulations *tend* to decrease the mark/cons ratio and number of collections. As trace granularity increases, these distortions become more pronounced as the number of potential collection points is limited as well. Not every collector produces the same distortion on the same metric, however. Excluding the traces with smaller live sizes, FG averages 1.8% higher mark/cons ratios and 0.5% more GCs versus perfect traces at even a 1KB granularity. While SyncLate simulations allow it to copy fewer objects early on, copying fewer objects causes the collector to delay whole-heap collections. The whole-heap collections remove unreachable objects from the older generation and prevent them from forcing the copying of other unreachable objects in the nursery. The collector eventually promotes more and more unreachable objects, so that it often must perform whole-heap collections soon after nursery collection, boosting both the mark/cons ratio and the number of GCs.

SyncMid. As expected, the best results are for SyncMid. From Table IV, the larger *From* space sizes produce similar results for SyncMid simulations and simulations using perfect traces at even large granularities. The design of SyncMid averages the times that it collects too early with those it collects too late. This balance makes the effects of trace granularity hard to predict. Both SyncEarly and SyncLate showed collector-dependent behavior. While conclusions for a new or unknown collector should not be drawn from their results, one could make assumptions about how they affect simulated metrics. In contrast, SyncMid simulations produce biases that are dependent upon both the metric and the collector: at a 2KB granularity, FG averages a mark/cons ratio 1.6% higher than simulations with perfect traces while VG's average mark/cons ratio is 0.4% too low. While the results were very good on the whole, there is still not a single metric for which every collector returned results without statistically significant distortions.

5.3 Trace Granularity Conclusion

While Unsynchronized simulations clearly caused extreme distortions, SyncMid sometimes allowed the use of traces with very small granularities to be simulated without significant differences. However, all of the Synchronized simulations suffer from statistically significant deviations. Because the metrics are distorted differently depending on the metric and simulated garbage collection algorithm, it would be impossible to "adjust" simulator results for novel algorithms or optimizations. Although we simulate copying garbage-collection, most of the metrics and algorithms are not dependent on copying, and

should hold for other algorithms such as mark-sweep (see Section 10 for additional discussion). These results prove the need for an accurate tracing and simulation environment in which to evaluate and compare garbage collection algorithms.

6. TRACE GENERATION USING MERLIN LIFETIME COMPUTATION

Life can only be understood backwards; but it must be lived forwards. –Søren Kierkegaard

The previous section motivates accurate traces for use in GC simulations, but the cost of whole-heap collection after each object allocation in a tracing collector is prohibitive. This section presents our new *Merlin Algorithm* for computing accurate object lifetimes. We designed Merlin for use with tracing copying collectors that we had already built. However, the key propagation of time stamps is similar to the use of decrements in reference counting (see Section 9) and could easily be used with other collectors, such as mark-sweep (see Section 10). Merlin reduces the time needed to generate the Java traces discussed in Section 4.2 from several years to a single weekend. Merlin does not require frequent collections and thus places less stress on the underlying system than the brute-force method of computing object lifetimes.

According to Arthurian legend, the wizard Merlin began life as an old man. He then lived backwards in time, dying at the time of his birth. Merlin's knowledge of the present was based on what he had already experienced in the future. Merlin, both the mythical character and our algorithm to compute object lifetimes, works in reverse chronological order so that each decision can be made correctly based upon knowledge of the outcome. Because our algorithm works backwards in time, the first time Merlin encounters an object in its calculation is the time the object dies (i.e., is not reachable).

The remainder of this section overviews how Merlin computes when objects transition from reachable to unreachable, gives a detailed explanation of why Merlin works, and discusses implementation issues. While our initial discussion focuses on using Merlin online for generating the perfect traces needed for simulation, we also present how Merlin can be used to compute object lifetimes off-line from an otherwise complete trace, and finally we discuss, if using granulated traces is appropriate, how Merlin can generate them.

6.1 Merlin Algorithm Overview

The brute-force method of computing object lifetimes is slow because, at each possible time, it computes which objects are unreachable by collecting the entire heap. The Merlin algorithm improves upon brute force by instead computing *the last time objects are reachable*. Since time advances in discrete steps, an object's death time is the time interval immediately following the one when it was last reachable.

Merlin has three key parts: (1) a forward pass that records events that make objects unreachable, (2) garbage collections that identify dead objects, and (3) a backward pass that computes for all dead objects the time at which they become unreachable.

During the forward pass, Merlin *timestamps* each object with the current time *whenever* it may become unreachable—i.e., whenever an object loses an incoming reference. If the object later loses another incoming reference (because the earlier update did not leave it unreachable), then Merlin simply overwrites the previous timestamp with the current time. Since an object only dies once when it becomes unreachable, Merlin computes this time after it knows an object is dead. Merlin could compute this time at the end of program execution when all objects are dead. Merlin instead uses a more efficient solution that pig-

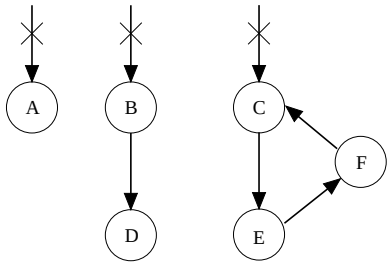


Fig. 4. When the program eliminates the last incoming references to objects A and B, they transition to unreachable. When the program eliminates the last reachable reference to object C, it becomes unreachable, even though it has other incoming references. Updates to objects that point directly or transitively to objects D, E, and F make them unreachable.

Table V. How objects become unreachable

(1)	A pointer update transitions an object from one to zero incoming references. For example, objects A and B in Figure 4.
(2)	A pointer update transitions an object from n to $n-1$ incoming references, and now all $n-1$ references are from unreachable objects. For example, object C in Figure 4.
(3)	An object's number of incoming references does not change, but a pointer update transitions the last reachable objects pointing to it to unreachable. For example, objects labeled D, E, and F in Figure 4.

gybacks on a host system garbage collection to identify garbage objects periodically. Given a set of dead (unreachable) objects, Merlin then computes *when* they were last reachable in a backward pass.

If a dead object has no incoming references, its current timestamp directly indicates its death time. However, some objects become unreachable even though they still have incoming references as shown in Figure 4. Merlin thus performs a timestamp propagation phase on unreachable objects. (By definition, no reachable object points to an unreachable one.) It starts with the unreachable object with the latest timestamp (ts) and continues processing unreachable objects in decreasing timestamp order. Sorting the list is $\Theta(n \log n)$ in the number of dead objects. For each object with a pointer ($s \Rightarrow t$), if $s_{ts} > t_{ts}$, Merlin propagates the later timestamp from the source to the target. Otherwise, Merlin stops propagating. Since it starts with the latest timestamp, worst case processing time is the number of unreachable (dead) objects.

6.2 Details and Implementation

This section expands on the key insights and implementation issues for Merlin. It first compares the time complexity of the brute-force and Merlin algorithms. It then discusses trace requirements, object reachability, timestamp propagation, and other uses of Merlin.

Finding dead objects requires a reachability analysis which with brute-force tracing on every allocation costs:

$$\sum_{i=1}^{n} |\text{live objects}| \text{ at } a_i$$

where n is the number of objects the program allocates, and a_i is an allocation event. Merlin eliminates the need to perform reachability analysis on every allocation. Merlin instead records object timestamps when an object loses an incoming pointer, and delays the bulk

of its propagation step until it can piggyback on a reachability analysis that occurs during a garbage collection. After a collection, Merlin works backward in time to find exactly when each dead object was last reachable. Merlin's execution time is thus proportional to processing each object once plus the number of times each object loses an incoming reference, m.

$$\sum_{i=1}^{n} |\text{object allocated}| \text{ at } a_i + \sum_{j=1}^{m} r_j$$

6.2.1 *Trace Requirements.* The in-order brute-force method processing adds death records as it produces the trace. Since Merlin determines death times out-of-order, it needs to introduce *timekeeping* into the traces. Time is related to trace granularity; time must advance wherever object death records may occur.[2]

6.2.2 *How Objects Become Unreachable.* Table V lists a series of generalizations that demonstrate how objects within the heap transition from reachable to unreachable. Scenarios 1 and 2 describe an object that is reachable until an action involving the object; Scenario 3 describes an object that becomes unreachable without direct involvement in an action. Not every pointer store kills an object, but if an object d dies, d either loses an incoming pointer or some other object o loses a reference which points to d directly or indirectly (the transitive closure of reachability from o).

6.2.3 *Finding Potential Last Reachable Times.* We propagate time stamps after an object is dead, instead of when it loses a reference. This section presents the Merlin pseudocode used to compute these last reachable times.

Instrumented Pointer Stores. Most pointer stores can be instrumented by a write barrier. The Merlin write barrier timestamps objects losing an incoming reference (the old target of the pointer) with the current time. Since time increases monotonically, each object will ultimately be stamped with the final time it loses an incoming reference. If the last incoming reference is removed by an instrumented pointer store, the Merlin code shown in Figure 6 stamps the object with the last time it was reachable.

Uninstrumented Pointer Stores. Because root pointers (especially ones in registers or thread stacks) are updated very frequently, instrumenting root pointer stores is prohibitively expensive and is rarely done. An object that is reachable until a root pointer update may not have the time it transitions from reachable to unreachable detected by any instrumentation. Just as a normal GC begins with a root scan, the Merlin algorithm performs a modified root scan at each allocation. This modified root scan enumerates the root pointers, but merely stamps the target objects with the current time. While root-referenced, objects are always stamped with the current time; if an object was reachable until a root pointer update, the timestamp will be the last time the object was reachable. Figure 7 shows Merlin's pseudocode executed whenever the root scan enumerates a pointer.

Referring Objects Become Unreachable. We also compute the time an object was last reachable for objects unreachable only because the object(s) pointing to them are unreachable (Scenario 3 of Table V). To handle pointer chains, updating the last reachable time for

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

one object requires recomputing the last reachable times of the objects to which it points. We simplify this process by noting that each of these object's last reachable time is the latest last reachable time of an object containing the former in its transitive closure set.

6.2.4 *Computing When Objects Become Unreachable.* Because the Merlin algorithm is concerned with *when* an object was last reachable and cannot always determine *how* the object became unreachable, the issue is to find a single method that computes every object's last reachable time. The methods in Figures 6 and 7 timestamp the correct last reachable time for those objects that are last reachable as described in Scenarios 1 and 2 of Table V. By combining the two timestamping methods with computing death times by membership in transitive closure sets of reachability, Merlin can determine the last reachable time of every object.

To demonstrate that this combined method works, we consider each scenario from Table V. Since an object last reachable as described by Scenario 1 is not the target of a pointer after it is last reachable, it is only a member of its transitive closure set, and the last reachable time Merlin computes will be the object's own timestamp. For Scenario 2 the last reachable time Merlin computes will also be the time with which the object is stamped: since the source of any pointers to the object must already be unreachable when the object is last timestamped, the source objects' last reachable times must be earlier. We show above that this combined method computes last reachable times for objects in Scenario 3, so Merlin can compute last reachable times by combining timestamping and computing the transitive closures, and need not know how each object transitioned from reachable to unreachable.

6.2.5 *Computing Death Times Efficiently.* Computing the full transitive closure sets is a time consuming process, requiring $O(n^2)$ time. But Merlin needs to find only the *latest* object containing the former object in its transitive closure set. Merlin performs a depth-first search from each object, propagating the last reachable time forward to the objects visited in the search. To save time, our implementation of Merlin first orders the unreachable objects from the earliest timestamp to the latest and then pushes them onto the search stack so that the latest object will be popped first. Figure 5(a) shows this initialization. Upon removing a new source object from the stack, the Merlin algorithm analyzes it for pointers to other (target) objects. If any target objects are stamped with an earlier time, the algorithm updates their timestamp with that of the source object. If the target object is definitely unreachable (e.g., will be reclaimed when the collection completes), it is pushed onto the stack also. Figures 5(b) and 5(c) show examples of this analysis. If the target object's timestamp is equal to that of the source object, then we do not need to push it on the stack, since we either have found a cycle (e.g., Figure 5(c)) or the target object is already on the stack. We also do not push the target object onto the stack if its timestamp is later than the source object's timestamp, since the target object must have remained reachable after the time currently being propagating. Pushing objects onto the stack from the earliest stamped time to the latest means each object is processed only once. The search proceeds from the latest stamped time to the earliest; after a first examination, any repeated examinations of an object must be computing earlier last reachable times. Hertz et al. [2002b] proved this asymptotically optimal method of finding last reachable times requires only $\Theta(n \log n)$ time, limited only by the sorting of the objects, where n is restricted to dead objects for this collection. Figure 8 shows the Merlin pseudo-code for this modified depth-first search.

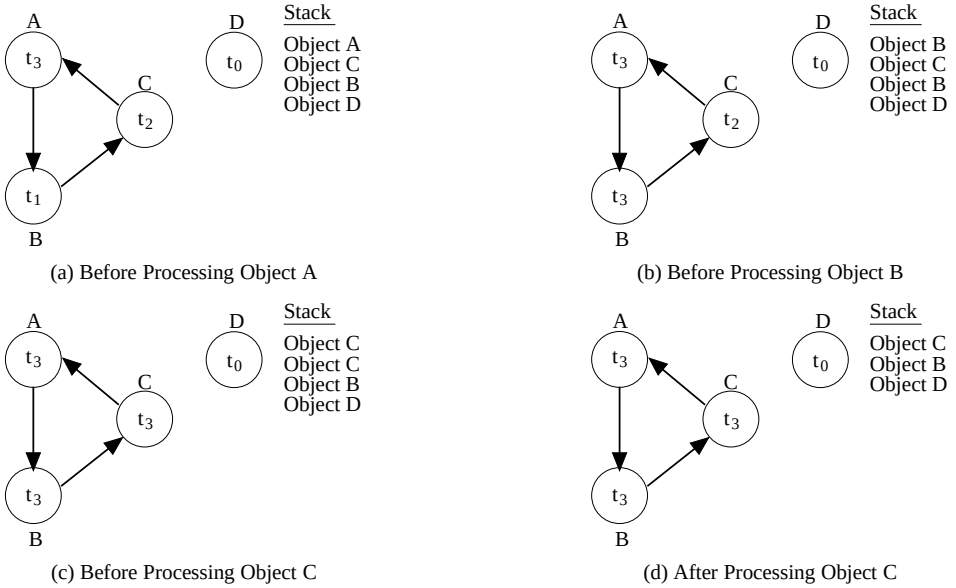


Fig. 5. Computing object death times, where $t_i < t_{i+1}$. Since Object D has no incoming references, Merlin's computation cannot change its timestamp. Although Object A was last reachable at its timestamp, care is needed so that the last reachable time does not change via processing its incoming reference. In (a), Object A is processed finding the pointer to Object B. Object B's timestamp is earlier, so Object B is added to the stack and last reachable time set. We process Object B and find the pointer to Object C in (b). Object C has an earlier timestamp, so it is added to the stack and timestamp updated. In (c), we process Object C. Object A is pointed to, but it does not have an earlier timestamp and is not added to the stack. In (d), the cycle has finished being processed. The remaining objects in the stack will be examined, but no further processing is needed.

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

Fig. 6. Code for Merlin's pointer store instrumentation

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

Fig. 7. Code for Merlin's root pointer processing

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

Fig. 8. Code of Merlin trace generation last reachable time computation

6.3 The Merlin Algorithm

As described so far, Merlin is able to reconstruct *when* objects were last reachable. However, it is still unable to determine *which* objects are no longer reachable. The Merlin algorithm uses two simple solutions to overcome this problem. Whenever possible, it delays computation until immediately after a collection, but before any memory is cleared. At this time, the object lifetime computation algorithm has access to all of the objects within the heap *and* the garbage collector's reachability analysis. By piggybacking upon this work, Merlin saves a lot of duplicative analysis. At other times (e.g., just before a program terminates), GC may not occur but the algorithm still needs a reachability analysis. In this case, Merlin first stamps the root-referenced objects with the current time and then computes the last reachable times of every object in the heap as usual. Objects with a last reachable time equal to the current time are still reachable and do not need object death records. All other objects must be unreachable and death records for them are added to the trace as usual. This method of finding unreachable objects enables the Merlin algorithm to work with any garbage collector. Even if the garbage collector cannot guarantee that it will collect all unreachable objects, Merlin performs the combined object reachability/last reachable time analysis just before the program terminates to find all of the last reachable times.

As stated in Section 2.1, we rely upon a couple of assumptions about the host garbage collector. First, we assume that any object the collector is treating as live will have the objects it points to also treated as live, as is required among GC algorithms without additional information. The collector thus removes an object only when all other objects pointing to it are provably unreachable. Second, the Merlin algorithm assumes that there are no pointer stores involving an unreachable object. Therefore, we assume that once an object becomes unreachable, its incoming and outgoing references are constant. Both of these preconditions are important for our transitive closure computation, and languages such as Java, C#, and Smalltalk satisfy them. Last, the Merlin algorithm adds an additional requirement, the reasons for which are explained in Section 6.2.3, that the instrumented pointer stores has access to the old value of the pointer. As the trace generator must already include a write

barrier to output pointer updates, and many write barriers already include these values (e.g., a reference counting write barrier), this additional requirement is not a hardship.

The order in which a trace generator using the Merlin algorithm adds object death records to the trace is an issue. As discussed in Section 6.2.1, the Merlin algorithm requires that the trace generator use the concept of time to determine where in the trace to place each object death record. The object death records either are added to the trace in chronological order before writing the trace to disk, or are included in the trace as they occur and a post-processing step places the records into proper order. Holding all the trace records in memory until Merlin computes all the object deaths is a difficult challenge; with larger traces, holding these records requires significant amounts of memory. Our trace generation implementation using Merlin for object lifetime computation uses an external post-processing step that sorts and integrates the object death records. Either way of handling this issue has advantages and disadvantages, but adds very little time to trace generation.

6.4 Using Merlin Off-line

Merlin does not need to perform its object lifetime analysis on-line: researchers have successfully used Merlin to compute object lifetime information from an otherwise complete garbage collection trace [Hertz and Berger 2004; Hirzel et al. 2003]. As described, the Merlin algorithm only needs to track pointer updates and to enumerate root pointers. This information can be obtained through instrumenting pointer store operations and performing a periodic modified root scan, but can also be acquired from a file that faithfully records all pointer stores and enumerates all root pointers. With this file, a simulator can generate the state of the program heap over the course of the program execution and use Merlin to compute the object lifetime information missing from the trace. Computing object lifetimes off-line can save substantial time when the lifetimes for only a subset of the objects are desired (e.g., only objects allocated during a particular phase of program execution).

6.5 Using Merlin for Granulated Traces

Our discussion of the Merlin algorithm has, until now, focused on the perfect traces required for GC simulation. GC traces are used not only for simulations, however, but have also been used to gain a deeper understanding of the issues affecting object lifetimes [Hirzel et al. 2002a; Hirzel et al. 2002b; Shaham et al. 2000] and to measure the effects of GC optimizations [Shaham et al. 2002]. Because of the speedup in trace generation achieved by the Merlin algorithm, it is now feasible to consider generating traces at granularities finer than each allocation. For instance, using Merlin, the trace generator could create a dynamic “escape-analysis” trace that is accurate at each method exit.

As described in Section 6.2.3, Merlin advances the trace time and enumerates and processes the root pointers at each allocation to help generate a perfect trace. However, these actions should occur whenever the trace must be accurate (which is every allocation in a perfect trace, but would be every method exit for a dynamic escape-analysis trace).[3] The Merlin algorithm is identical for any trace generation, the only change being how often the time is updated and the modified root scan is performed; the algorithm otherwise acts the same after each collection and at every instrumented pointer update.

[3]These arguments could also be used to generate coarser-grained traces.

7. EVALUATION OF MERLIN

We implemented a trace generator in the Jikes RVM that can use either the brute-force method or the Merlin algorithm to compute object lifetimes. We then performed timing runs on a Macintosh Power Mac G4, with two 533 MHz processors, 32KB on-chip L1 data and instruction caches, 256KB unified L2 cache, 1MB L3 off-chip cache and 384MB of memory, running PPC Linux 2.4.3. We used only one processor for our experiments, which were run in single-user mode with the network card disabled. We built two versions of the VM with trace generation, one using Merlin for object lifetime computation and one using the brute-force method. Whenever possible we used identical code in the two VMs. For these experiments, the trace generator employed the semi-space collector needed by the brute-force method so as to keep the two systems as similar as possible.

Merlin's running time is spent largely in performing the modified root scan required after every allocation. We further improved Merlin's running time by including a number of optimizations to minimize the number of root pointers that must be enumerated at each of these locations. Our first optimization was to instrument pointer store operations involving static pointers. With this instrumentation, Merlin need not enumerate these pointers during its root scan. Instead, it can treat them as it does heap pointers, since any stores to these pointers will be processed by the same instrumentation. Because Java allows functions to access only their own stack frame, repeated scanning within the same method always enumerates the same objects from the pointers below this method's frame. We implemented a *stack barrier* that is called when frames are popped off the stack, enabling Merlin to scan less of the stack [Cheng et al. 1998]. We do not include the stack barrier in the brute-force generator because it introduces overhead on each method invocation, and it was beyond the scope of this work to evaluate it.

We generated traces at different granularities across a range of programs. Because of the time required for brute-force trace generation, we limited some traces to only the initial 4 or 8 megabytes of data allocation (which still required over 34 hours in one case). Working with common benchmarks and identical granularity, trace generation using Merlin achieved speedup factors of up to 816. In the time needed by the system using the brute-force method to generate traces with granularities of 16K to 1024K bytes, trace generation with Merlin completed perfect traces. Figure 9 shows the speedup to the trace generator when using Merlin, generating perfect traces, versus using the brute-force method at different levels of granularity. Clearly, Merlin can greatly reduce the time needed to generate a trace. However, as seen in Figure 9, the speedup is less when granularity increases. The time required largely depends on the time needed to generate object death records—the trace granularity. Brute force limits object death time processing to only those points where the trace is accurate; as the granularity increases it performs fewer GCs and the time needed greatly diminishes. Even though Merlin performs fewer actual collections than brute force with a large granularity, the cost of enumerating the roots at every allocation and updating timestamps can become greater than the collection cost at large granularities.

These results are typical. For programs with larger average maximum live size and total allocation volume, Merlin should provide further speedups due to the differences between its death time propagation algorithm and root scanning costs compared to the larger cost of repeatedly tracing the heap.

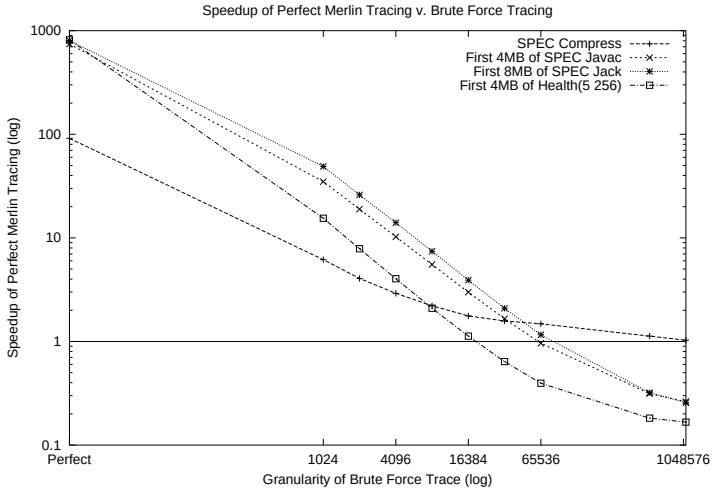


Fig. 9. The speedup of perfect Merlin trace generation versus Brute-Force granulated trace generation. Note the log-log scale.

7.1 Granulated Trace Generation

When generating escape-analysis traces (i.e., small granularities), it is clear from the above results that Merlin will be much quicker than brute force. An open question, however, is the fastest way to generate traces with *large granularities*. These traces could not, of course, be used for GC simulations, but could still be used to tune profile-driven feedback optimizations [Ungar and Jackson 1992; Cheng et al. 1998; Blackburn et al. 2001] or to gain a deeper understanding of the issues affecting object lifetimes [Hirzel et al. 2002a; Hirzel et al. 2002b; Shaham et al. 2000]. While we show in Section 5 how the compounding of these lifetime errors results in statistically significant distortions for simulation results, when analyses consider each object's lifetime independently the error is bounded by at most one trace granule, and snowballing cannot occur. Previously, Hirzel et al. [2002b] showed that their analysis was not altered by the use of granulated traces.

Even with the improvement Merlin provides to trace generation, the time required to generate a trace is 70–300 times slower than running the program without tracing. As shown by Figure 10, granulated traces require much less time to generate, and they are thus attractive when granulation does not distort results. Given a heap that actually has a maximum live size of 10MB, for example, a trace with a 10KB granularity will overestimate the maximum live size by at most 0.1%.

Figure 10 shows that while introducing some trace granularity allows Merlin tracing to run faster, there is little gain in generating traces with a granularity above 4096 bytes. Since Figure 9 shows that the time needed for trace generation using brute force continues to improve even when the trace granularity is increased from 512KB to 1MB, it still is not clear what is the best way to generate a granulated trace. Figure 11 examines the speedup that generating a granulated Merlin trace offers versus generating a granulated brute-force trace.

As seen in Figure 11, Merlin outperforms brute force at all tested granularities and over all of the benchmarks examined. While all the work required by brute force (performing a GC) is directly related to the granularity of the trace generated, some of Merlin's workload (enumerating and scanning the root pointers) is related to the trace granularity and some work is constant (timestamping objects losing incoming references via instrumented

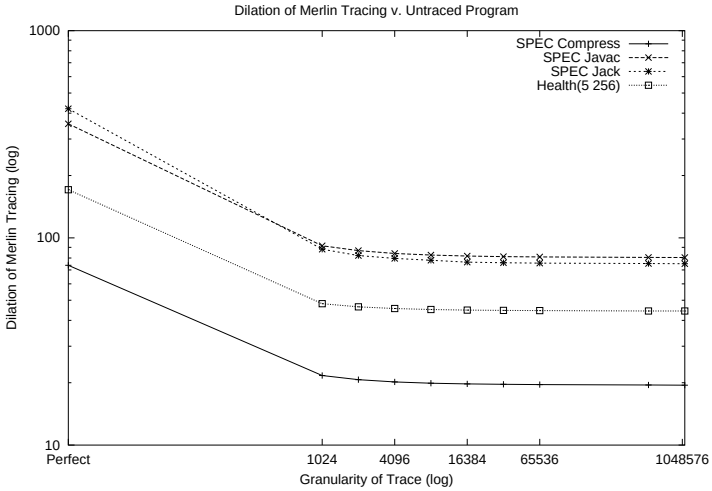


Fig. 10. The cost of Merlin versus trace generation without lifetimes. Merlin imposes a substantial slowdown when generating a perfect trace. If approximate object lifetimes are desired, generating a slightly granulated trace can require $\frac{1}{5}$ the time.

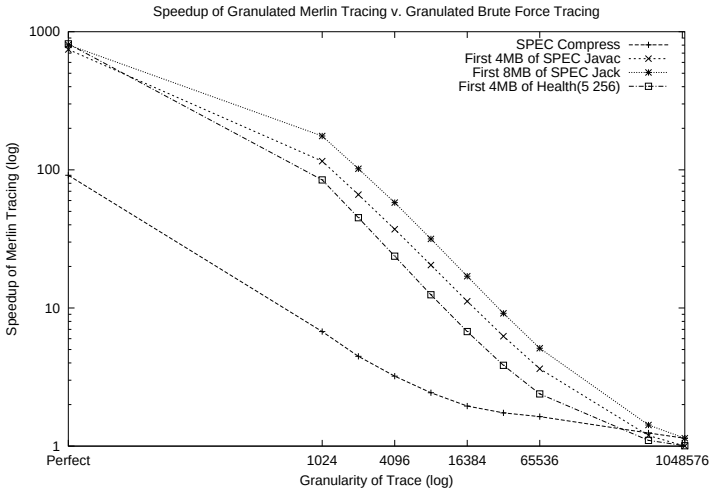


Fig. 11. The speedup of trace generation using Merlin versus brute force. Merlin tracing is faster for each benchmark at every trace granularity tested. Note the log-log scale.

pointer stores). Because of this constant work overhead for Merlin, the improvement in generating a trace of SPEC _228_jack slowly drops from a speedup factor of 817 for perfect traces to a factor of 5 at a granularity of 64K and finally to a factor of 1.14 at 1MB granularity. Even at this very high granularity, however, the speedup of not needing to perform the repeated garbage collections makes Merlin the winner. When these results are combined with those from Figure 10, they provide a persuasive argument for using Merlin to compute object lifetimes even for granulated traces.

8. PROGRAM HEAP VISUALIZATION

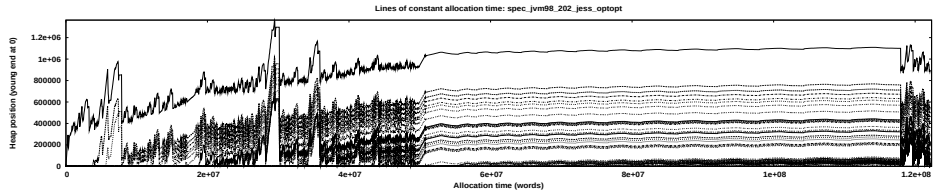
Whether creating new GC optimizations, explaining the performance of an existing algorithm, or developing a set of benchmarks to test GC, researchers need to understand the lifetimes of objects in the heap and how they interact. Researchers have used program heap graphs, visualizations showing the composition of the heap, identifying the locations of unreachable object over the entire program run, to develop and share this knowledge (e.g., [Runciman and Wakeling 1992; Runciman and Röjemo 1995; Sansom 1994; Sansom and Jones 1994; Rojemo and Runciman 1996; Stefanović 1999; Shaham et al. 2000]).The resolution of a visualization is dependent on the granularity of the trace used; granulated traces can generate the powerfully simple graphs (such as those in [Shaham et al. 2000]), while precise graphs capable of zooming in to show very fine details (such as the graphs in [Stefanović 1999] and the figures in this section) require perfect traces.

This section presents several program heap visualizations from Jikes RVM produced with Merlin, which reveal object lifetimes and lifetime patterns. Section 8.1 analyzes a few of these graphs to show how they provide insight into potential GC optimizations and Section 8.2 illustrates how these visualizations can help evaluate benchmark programs.

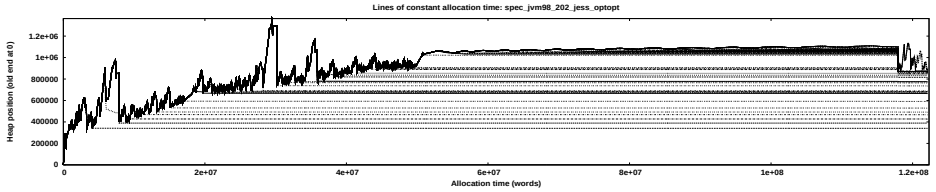
8.1 Program Heap Behavior Insights

The simplest *heap profile* visualizations show the composition of the heap over a program run, providing a means of seeing where, in an age-ordered heap, the reachable objects exist. Figure 12(a) shows a heap profile of SPEC _202_jess when run with the Jikes RVM Opt (optimizing) compiler. The Y-axis of this graph represents the position of reachable objects in an age-ordered heap, while the X-axis represents time (measured as the total number of bytes allocated into the heap so far). At the start of each program "segment" (some set number of bytes of allocation), we introduce a new line along the X-axis. Lasting until program termination, the line shows, at each moment, the position in the heap of the boundary between the objects allocated before and after this point. In this graph, we can see the program run through three distinct phases: startup, stable running, and finishing. The startup phase, lasting the first 50000000 words of allocation, shows the variable live sizes and object lifetimes arising from compilation. The second phase of this profile shows a regular pattern of very short lived objects — the actual running of the jess benchmark and the last phase shows a brief return of compilation as the program reaches the SpecApplication termination code. Given this complex behavior, a garbage collector could benefit from using phase detection to moderate any dynamic optimizations. During the long stable (middle) phase of the run, optimizations may yield little or no benefit as most collectors would already perform well. Rather then spend time working for little benefit, a system would be better served saving that time and using the default behavior.

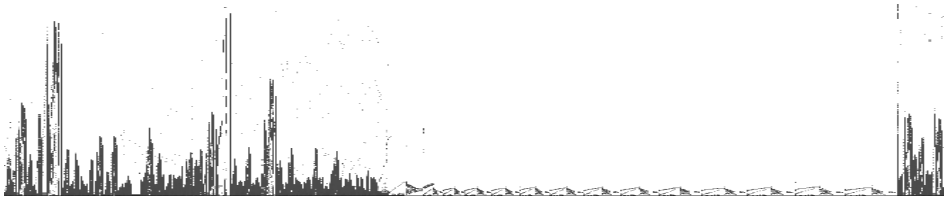
Figure 12(b) also shows a heap profile of SPEC _202_jess run with the Jikes RVM optimizing compiler. This heap profile differs from Figure 12(a), by showing the *oldest* objects of the age-ordered heap along the Y-axis and adding newly allocated objects to the *top* of the graph. Long-lived objects appear as a horizontal line of constant live amounts in these figures. When some objects die at some point, the line segments get closer together. Figure 12(b) shows many immortal objects that are created during the first (compilation) phase of the program. As shown in [Blackburn et al. 2001], these immortal objects present inviting targets for optimizations such as pretenuring. The second phase of the trace (when most compilation is complete and the benchmark is actually running) shows new program



(a) Heap profile of SPEC _202_jess. In this graph, the oldest objects are shown at 0 on the Y-axis. The graph shows that the program goes through three phases: it begins by compiling the program, then solves the fifteen puzzles that comprise the benchmark, and finally outputs the result.



(b) Inverted heap profile for _202_jess. In this graph, the youngest objects are shown at 0 on the Y-axis. It is easy to see the compiler's initial immortal allocations, the short-lived allocations of the benchmark, and the medium-lived allocations from when the program terminates.



(c) Demise graph for _202_jess. In the demise map, the youngest objects appear at 0 on the Y-axis. This graph shows that the program has different phases of object lifetime behavior with the benchmark allocating only short-lived objects. The black vertical lines in the demise map show large numbers of objects becoming unreachable at once, suggesting death of large linked structures such as trees or lists.

Fig. 12. Three different heap visualizations for SPEC _202_jess using the optimizing compiler at run time. While each of these graphs summarizes the composition of the heap over the run, the different ways of expressing this composition can highlight different information. Using all three graphs in combination is an easy way to gain a good understanding of the object lifetime behavior of the program.

segments barely rising from the graph and then rapidly disappearing, i.e., allocation of many very short-lived objects. The last phase of the program shows the system compiles the methods corresponding to the final code for the program. The optimizing compiler uses short-lived objects and outputs the long-lived blocks of machine code causing the behavior seen during this final phase. The very different lifetime behaviors at different points of the program suggests that a garbage collector that could detect these phases and change its behavior accordingly could perform well on this benchmark.

Another type of heap visualization is the *demise map*, an example of which can be seen in Figure 12(c). Like the previous visualizations, a demise map's X-axis is the number of bytes allocated and its Y-axis is the heap position in the age-ordered heap. However, points on the demise map indicate an object's becoming unreachable. We represent the density of objects becoming unreachable at the same location (in an age-ordered heap) by

the darkness of the point on the map. The demise map in Figure 12(c) is also for the runs of _202_jess using the Jikes RVM optimizing compiler. The graph again shows the program running through several phases of object lifetime behavior over the run. Examining the demise map provides some useful information. At several places in the trace, we can see a number of objects become unreachable at the same time (as dark vertical bands in the demise map). By grouping these objects together and delaying their collection until they all become unreachable, a collector could greatly improve its performance.

8.2 Evaluation

Heap visualizations help reveal how demanding a benchmark is with respect to its memory management needs. We present heap profiles for the SPECjvm98 benchmarks and pseudoJBB, a version of SPECjbb modified to run for a specified number of transactions rather then a specified length of time. Figure 13 shows heap profiles with the youngest objects at the bottom of the y-axis; and Figure 14 shows heap profiles with the youngest objects at the top of the y-axis.

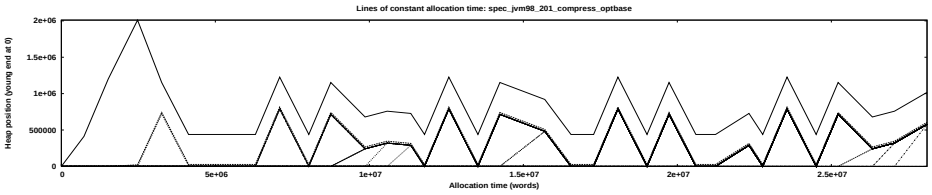
The figures indicate a range of challenges for garbage collection. For instance, _201_compress and _222_mpegaudio do not stress garbage collectors much, whereas _209_db, pseudoJBB, and _213_javac demonstrate richer memory management behavior. We analyze each of these programs below.

Consider Figure 13(a) which presents a run of _201_compress. While the irregular allocation peaks in Figure 13(a) suggest that it could be useful for analyzing phase change optimizations or comparing algorithms that dynamically select heap sizes, this benchmark would not be useful for comparing statically sized heaps. While a statically chosen heap size must be sufficient to hold the initial peak (approximately 2MB), this space is larger than the rest of the program needs. _201_compress thus exercises the garbage collector only at the smallest heap sizes.

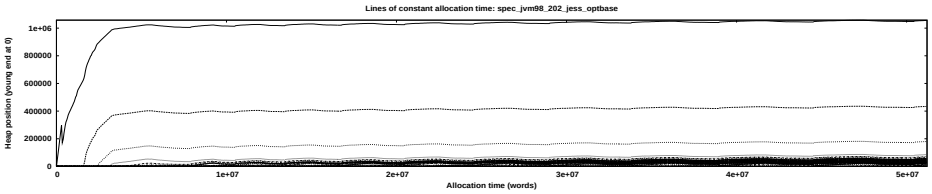
The heap profile of _222_mpegaudio, Figure 13(f), shows that it has a low ratio of bytes allocated to maximum live size (a ratio of only 2.1:1). The heap profile also shows that objects allocated by this program are either immortal or immediately become unreachable. However, it shows two phases. The program allocates so little that it can steadily increase its live size for the entire duration of the program.

While _209_db, shown in Figure 13(d), maintains a constant live size, the heap profile indicates that it allocates ten times as much data as this live size, which limits how much stress it places on the garbage collector. After _209_db populates its database with "immortal" objects (roughly the first 3MB of allocation), the program allocates objects which immediately become unreachable. Combining these two behaviors, the heap profile in Figure 13(d) shows that with a large enough nursery, a generational garbage collector should perform well on _209_db and whole heap collections are a waste of time. However, this behavior is not the whole story for _209_db, because its choice of allocator and collector radically affect its performance through the locality behavior they induce [Blackburn et al. 2004a; Huang et al. 2004; Hertz and Berger 2004].

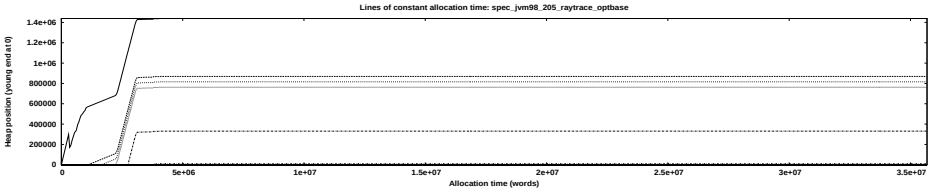
PseudoJBB only allocates about ten times as many bytes as its maximum live size, as does _209_db. The allocated objects, however, have more complex lifetime patterns. After initially allocating and building a large structure (at 1.4MB), the program makes the majority of this unreachable. The program again allocates a large amount of immortal objects. It then allocates short-lived objects and periodically causes these to become unreachable. Unlike _209_db, however, these short-lived objects do not immediately become



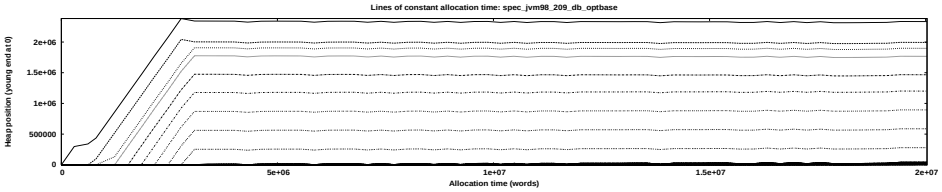
(a) Heap Profile of _201_compress



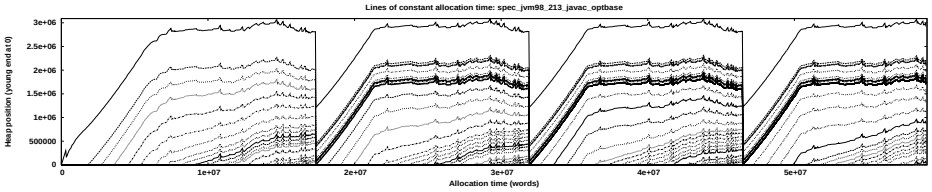
(b) Heap Profile of _202_jess



(c) Heap Profile of _205_raytrace



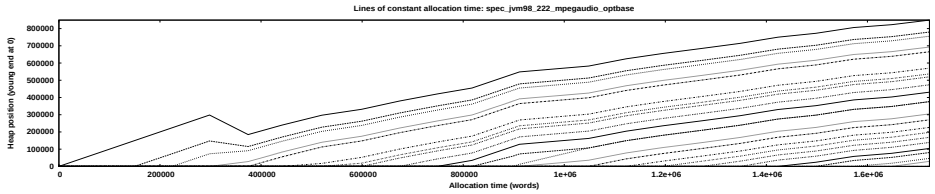
(d) Heap Profile of _209_db



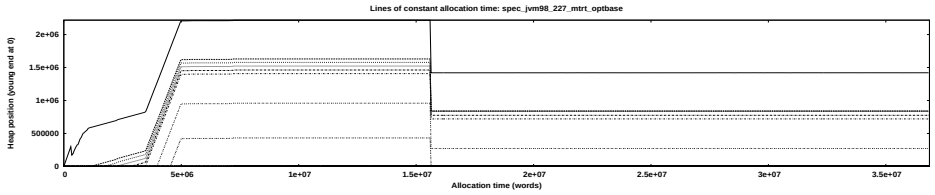
(e) Heap Profile of _213_javac

unreachable but must remain in the heap for a time. Figure 14(i) shows pseudoJBB begins allocating the next period of short-lived objects before it has made all objects from the previous period unreachable. This behavior guarantees that some objects will survive simple nursery collections in generational collectors, and need a more expensive collection to reclaim them.

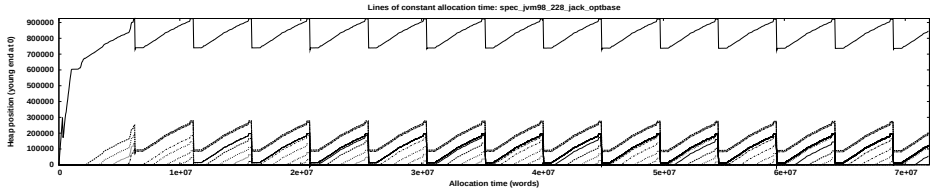
The heap profile of _213_javac shows this program periodically building and then making large structures unreachable; few generational algorithms would normally size their



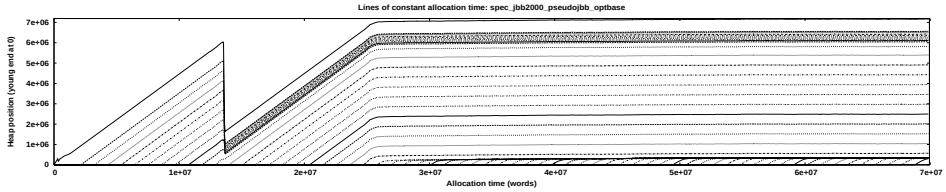
(f) Heap Profile of _222_mpegaudio



(g) Heap Profile of _227_mtrt



(h) Heap Profile of _228_jack



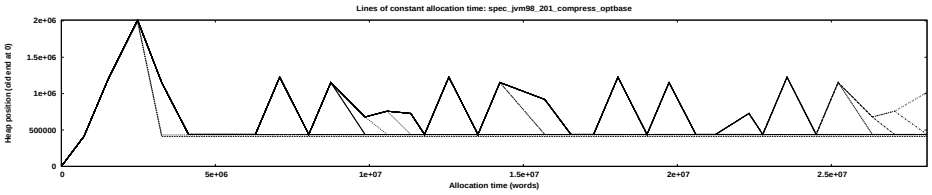
(i) Heap Profile of pseudoJBB

Fig. 13. Heap profile graphs for the SPECjvm 98 benchmarks and pseudoJBB. Newly allocated objects are added at the bottom of these heap profiles. To limit the influence the host JVM has on these graphs, they were generated from runs using the Jikes RVM baseline JIT compiler.

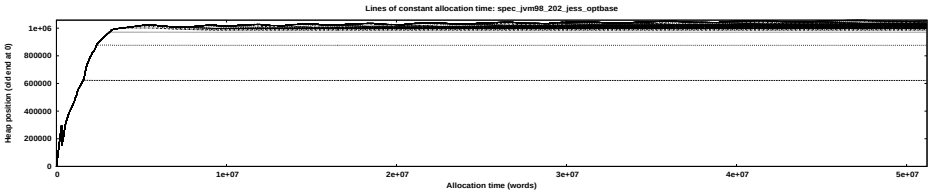
nursery or Eden space large enough to hold these structures. This behavior ensures that some objects will be promoted into the mature space and need full heap collections to be reclaimed. Especially when combined with _213_javac's high ratio of allocation to maximum live size, it is clear this benchmark will highlight garbage collector performance differences.

9. RELATED WORK

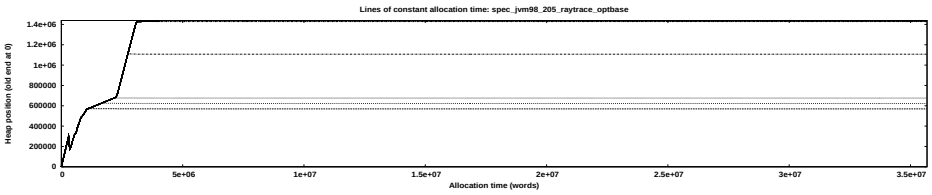
We now discuss the prior research on which this study builds. There are 3 areas of research that are most relevant: reference counting, approximating object lifetimes, and generating perfect (accurate) traces.



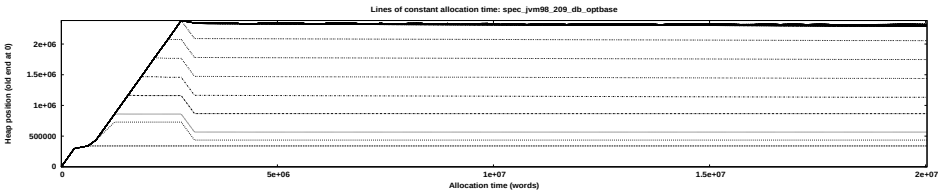
(a) Inverted Heap Profile of _201_compress



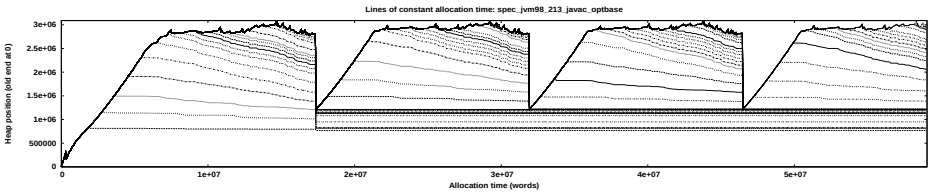
(b) Inverted Heap Profile of _202_jess



(c) Inverted Heap Profile of _205_raytrace



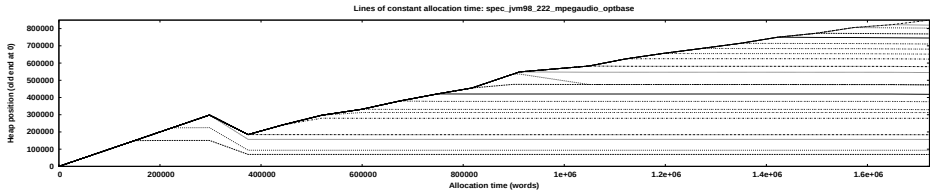
(d) Inverted Heap Profile of _209_db



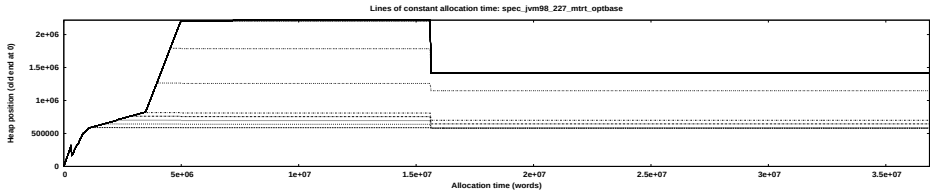
(e) Inverted Heap Profile of _213_javac

Reference Counting. While the Merlin algorithm does not do reference counting (RC), issues that arise from its time stamping are similar to those from counting references. As a result of these similar issues, RC collectors are often closely related to the Merlin algorithm and we describe them here.

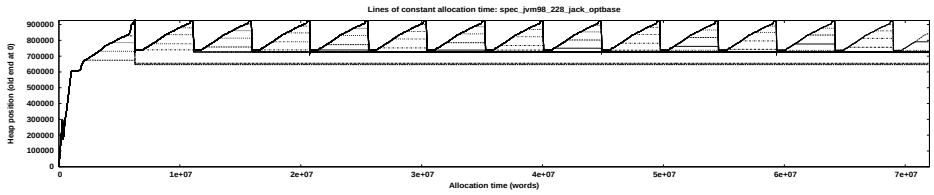
Reference counting associates a count of incoming references with each object; when the count is 0, it frees the object [Collins 1960]. To improve efficiency, modern deferred reference counters do not count the numerous updates to stack variables and registers [Deutsch



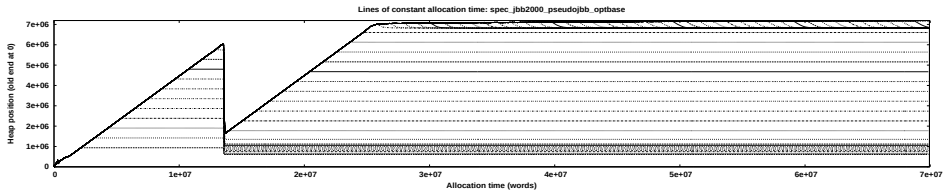
(f) Inverted Heap Profile of _222_mpegaudio



(g) Inverted Heap Profile of _227_mtrt



(h) Inverted Heap Profile of _228_jack



(i) Inverted Heap Profile of pseudoJBB

Fig. 14. Inverted heap profile graphs for the SPECjvm 98 benchmarks and pseudoJBB. Newly allocated objects are drawn at the top of these heap profiles. To limit the influence the host JVM has on these graphs, they were generated from runs using the Jikes RVM baseline JIT compiler.

and Bobrow 1976], but instead compute correct counts periodically. As with other algorithms, RC must enumerate the stacks and registers when it collects the heap. Since reference counting cannot find dead cycles [Weizenbaum 1962], modern implementations add periodic tracing collection or perform trial deletion [Vestal 1987; Bacon and Rajan 2001]. Trial deletion keeps objects that lost a pointer, but whose count did not reach 0, in a "candidate set". It then recursively performs trial deletions on the objects in this set and those objects reachable from them. When all the reference counts go to zero, the objects form a dead cycle and can be reclaimed.

At first glance, adding time stamps to RC might seem faster than piggybacking on a tracing collector, but cycles complicate this argument. To compute a perfect trace using an RC (and ignoring cycles), we could extend the object headers to include a time stamp,

update the time stamp with each decrement, update the reference counts at every allocation, and record and propagate time stamps when the object's reference count goes to zero. Since cycles must be unreachable at program termination, we could then propagate these time stamps to accurately compute the remaining death times. However, never collecting cycles might cause the program to fail by running out of memory. Adding RC tracing or trial deletion reverts trace generation to the cost of the brute-force method plus additional reference counting overheads. To add Merlin to an existing RC system is thus likely to yield similar or worse performance than using Merlin with a tracing collector.

Unlike RC, the Merlin algorithm is not a garbage collector, but merely computes object lifetimes. While there are similarities between Merlin and RC (deferred reference counting is similar to Merlin's time stamping), Merlin relies upon an underlying collector to actually reclaim objects whereas RC performs this reclamation. While RC *can* use an additional tracing collector to detect dead cycles, the Merlin algorithm *needs* a garbage collector to compute which objects are unreachable.

Lifetime Approximation. To cope with the cost of producing GC traces, there has been previous research into approximating the lifetimes of objects. These approximations model the object allocation and object death behavior of actual programs. One paper described mathematical functions that model object lifetime characteristics based upon the actual lifetime characteristics of 58 Smalltalk and Java programs [Stefanović et al. 2000]. Zorn and Grunwald [1992] compare several different models one can use to approximate object allocation and object death records of actual programs. Neither study attempted to generate actual traces, nor does either study consider pointer updates; rather, these studies attempted to find ways other than trace generation to produce input for memory management simulations.

Perfect Tracing. Our previous work [Hertz et al. 2002a] presented the effects of trace granularity on GC simulator fidelity. Additionally, it described how Merlin can be used to generate the perfect traces needed for GC simulation, and presented a preliminary comparison between generating perfect Merlin traces and perfect and granulated brute-force traces. Because of this work, others have begun to re-examine their analyses to see if their results were affected by trace granulation [Hirzel et al. 2002b]. We presented additional work proving that the Merlin algorithm runs in asymptotically optimal time [Hertz et al. 2002b]. Our previous work did not demonstrate how to use Merlin to generate granulated traces, nor did it include the more detailed timing results we present here. The current work also discusses additional uses of Merlin and presents program heap visualizations that are only possible due to Merlin's reduced processing time.

10. APPLICABILITY TO OTHER COLLECTION ALGORITHMS

We built these and other copying algorithms in GCTk [Blackburn et al. 2002; Blackburn et al. 2001; Stefanović et al. 2002], a freely available memory management toolkit, for use with Jikes RVM. Although our results are for copying collectors, there is no reason to believe they will not hold for mark-sweep (MS) collectors and hybrid copying and MS collectors, such as the popular copying nursery/Eden space and MS old space. Product VMs often use this later variation due to its high performance.[4] MS offers significant

[4]A more recent toolkit MMTk [Blackburn et al. 2004b; 2004a] contains MS, reference counting, and their generational variants. Experimental comparisons of copying versus MS collection of the mature space show neither is

space efficiency over copying [Hertz and Berger 2004] which is especially important in the old space. MS collectors thus trigger collections less often than copying. However, if collecting the same region as a copying collector, MS finds exactly the same objects as live since it computes reachability the same way. Therefore, given sufficient collections, the accuracy of MS collectors should be similarly distorted by poor choice of collection point with respect to a granulated trace.

We can make no conclusions about the sensitivity to trace granulation of reference counting collectors since their liveness test is different from copying. However, our traces contain sufficient information to simulate these algorithms as well.

11. SUMMARY

The use of granulated traces for GC simulation is problematic. We first develop a method that can statistically test if a variable affects GC simulation. We then use this method to show that, over a wide range of variables, granulated traces produce results that are significantly different from those produced by perfect traces. While we show that there are ways of simulating granulated traces that are better at minimizing these issues, we find none of these methods can eliminate all the problems. With these results, we propose standard trace formats should include additional information.

We then introduce and describe the Merlin Algorithm. We show how trace generation using the Merlin algorithm can produce perfect traces more than 800 times faster than the common (brute force) method of trace generation. We also describe how, for new analyses, Merlin makes it possible to generate traces at even finer granularities, and when it may be permissible to use coarser traces. Finally, we show that given the Merlin algorithm there is never a reason to generate traces coarser than a 4KB granularity. Thus, the Merlin algorithm makes trace generation quick and easy, and eliminates the need for using granulated traces in simulation.

Finally, we present several examples of program heap visualizations, powerful tools that, with traces like those generated by Merlin, are easy to generate. With graphs of several well-known, commonly used benchmark programs, we show how they provide insights that can be used to design future GC optimizations and evaluate a program's memory management needs.

Acknowledgments

We would like to thank Aaron Cass and John N. Zigman for their help building and refining the simulator and simulation analysis infrastructure, Emery Berger for his suggestions, and the anonymous reviewers whose suggestions have helped improve this paper. The authors also appreciate the time and effort spent by Martin Hirzel validating the Merlin analysis. We would like to acknowledge Dr. Mario Wolczko and Antonio Cunei, who were simultaneously and independently inventing and implementing the Merlin algorithm.

REFERENCES

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., BURKE, M. G., CHENG, P., CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M. F., NGO, T., SARKAR, V., SERRANO, M. J., SHEPHERD, J. C., SMITH, S. E., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 2000. The Jalapeño virtual machine. *IBM Systems Journal 39(1)*.

clearly best [Blackburn et al. 2004a].

ALPERN, B., ATTANASIO, C. R., BARTON, J. J., COCCHI, A., HUMMEL, S. F., LIEBER, D., NGO, T., MERGEN, M., SHEPHERD, J. C., AND SMITH, S. 1999. Implementing Jalapeño in Java. In *Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Languages, & Applications*. ACM SIGPLAN Notices, vol. 34(10). ACM Press, Denver, CO, 314–324.

APPEL, A. W. 1989. Simple generational garbage collection and fast allocation. *Software Practice and Experience 19(2)*, 171–183.

BACON, D. F. AND RAJAN, V. T. 2001. Concurrent cycle collection in reference counted systems. In *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, J. L. Knudsen, Ed. Springer-Verlag, vol. 2072. Springer-Verlag, Budapest, Hugary, 207–235.

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004a. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*. NY, NY, 25–36.

BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. 2004b. Oil and water? High performance garbage collection in Java with JMTk. In *Proceedings of the 26th International Conference on Software Engineering*. Scotland, UK, 137–146.

BLACKBURN, S. M., JONES, R. E., MCKINLEY, K. S., AND MOSS, J. E. B. 2002. Beltway: Getting around garbage collection gridlock. In *Proceedings of the SIGPLAN 2002 Conference on Programming Language Design and Implementation*. Berlin, Germany, 153–164.

BLACKBURN, S. M. AND MCKINLEY, K. S. 2003. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA)*. Anaheim, CA, 244–358.

BLACKBURN, S. M., SINGHAI, S., HERTZ, M., MCKINLEY, K. S., AND MOSS, J. E. B. 2001. Pretenuring for Java. In *Proceedings of SIGPLAN 2001 Conference on Object-Oriented Programming, Languages, & Applications*. ACM SIGPLAN Notices, vol. 36(10). ACM Press, Tampa, FL, 342–352.

CAHOON, B. AND MCKINLEY, K. S. 2001. Data flow analysis for software prefetching linked data structures in Java controller. In *2001 International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*. Barcelona, Spain, 280–291.

CHENG, P., HARPER, R., AND LEE, P. 1998. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN 1998 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, vol. 33(5). ACM Press, Montreal, Canada, 162–173.

COLLINS, G. E. 1960. A method for overlapping and erasure of lists. *Communications of the ACM 3(12)*, 655–657.

DEUTSCH, L. P. AND BOBROW, D. G. 1976. An efficient incremental automatic garbage collector. *Communications of the ACM 19*, 9 (September), 522–526.

HERTZ, M. AND BERGER, E. D. 2004. Automatic vs. explicit memory management: Settling the performance debate. Tech. Rep. TR-04-17, University of Massachusetts. Mar.

HERTZ, M., BLACKBURN, S. M., MOSS, J. E. B., M^{c}KINLEY, K. S., AND STEFANOVIĆ, D. 2002a. 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*. ACM SIGMETRICS Performance Evaluation Review, vol. 30(1). ACM, Marina Del Rey, CA, 140–151.

HERTZ, M., IMMERMAN, N., AND MOSS, J. E. B. 2002b. 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)*, R. Baeza-Yates, U. Montanari, and N. Santoro, Eds. IFIP Conference Proceedings, vol. 223. Kluwer, Montreal, Canada, 230–241.

HIRZEL, M., DIWAN, A., AND HENKEL, J. 2002b. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems 24(6)*, 593–624.

HIRZEL, M., DIWAN, A., AND HERTZ, M. 2003. Connectivity-based garbage collection. In *Proceeding of SIGPLAN 2003 Conference on Object-Oriented Programs, Systems, Languages, & Applications*. ACM SIGPLAN Notices, vol. 38(10). ACM Press, Anaheim, CA, 359–373.

HIRZEL, M., HENKEL, J., DIWAN, A., AND HIND, M. 2002a. Understanding the connectivity of heap objects. In *ISMM 2002 Proceedings of the Third International Symposium on Memory Management*. ACM SIGPLAN Notices, vol. 37(1). ACM Press, Berlin, Germany, 36–49.

HOSKING, A. L., MOSS, J. E. B., AND STEFANOVIĆ, D. 1992. A comparative performance evaluation of write barrier implementations. In *Proceeding of SIGPLAN 19923 Conference on Object-Oriented Programs,*

Systems, Languages, & Applications. ACM SIGPLAN Notices, vol. 27(10). ACM Press, Vancouver, Canada, 92–109.

HUANG, X., WANG, Z., BLACKBURN, S. M., MCKINLEY, K. S., MOSS, J. E. B., AND CHENG, P. 2004. The garbage collection advantage: Improving mutator locality. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, & Applications (OOPSLA)*. Vancouver, BC.

JONES, R. AND LINS, R. 1996. *Garbage collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, NY.

LIEBERMAN, H. AND HEWITT, C. E. 1983. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM 26*, 6, 419–429.

NATRELLA, M. G. 1963. *Experimental Statistics*. US Department of Commerce, Washington, DC.

NYSTROM, N. 1998. Bytecode-level analysis and optimization of Java classfiles. M.S. thesis, Purdue University, West Lafayette, IN.

ROJEMO, N. AND RUNCIMAN, C. 1996. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *International Conference on Functional Programming*. 34–41.

RUNCIMAN, C. AND RÖJEMO, N. 1995. Lag, drag and post-mortem heap profiling. In *Implementation of Functional Languages Workshop*. Båstad, Sweden.

RUNCIMAN, C. AND WAKELING, D. 1992. Heap profiling of lazy functional programs. Tech. Rep. 172, University of York, Dept. of Computer Science, Heslington, York Y01 5DD, United Kingdom. April.

SANSOM, P. M. 1994. Execution profiling for non-strict functional languages. Ph.D. thesis, University of Glasgow, Glasgow, Scotland.

SANSOM, P. M. AND JONES, S. L. P. 1994. Time and space profiling for non-strict, higher-order functional languages. Tech. Rep. Research Report FP-1994-10, Dept. of Computing Science, University of Glasgow, Glasgow, Scotland.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2000. On the effectiveness of GC in Java. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*. ACM SIGPLAN Notices, vol. 36(1). ACM Press, Minneapolis, MN, 12–17.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2002. Estimating the impact of heap liveness information on space consumption in Java. In *ISMM 2002 Proceedings of the Third International Symposium on Memory Management*. ACM SIGPLAN Notices, vol. 37(1). ACM Press, Berlin, Germany, 64–75.

SPECjvm98 1998. Standard Performance Evaluation Corporation (SPEC). Available at http://www.spec.org/osg/jvm98.

STEFANOVIĆ, D. 1999. Properties of age-based automatic memory reclamation algorithms. Ph.D. thesis, University of Massachusetts, Amherst, MA.

STEFANOVIĆ, D., HERTZ, M., BLACKBURN, S. M., MCKINLEY, K. S., AND MOSS, J. E. B. 2002. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *MSP 2002 Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*. ACM SIGPLAN Notices, vol. 38(1). ACM Press, Berlin, Germany.

STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. 1999. Age-based garbage collection. In *Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Languages, & Applications*. ACM SIGPLAN Notices, vol. 34(10). ACM Press, Denver, CO, 379–381.

STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. 2000. On models for object lifetimes. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*. ACM SIGPLAN Notices, vol. 36(1). ACM Press, Minneapolis, MN, 137–142.

UNGAR, D. M. 1984. 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*. ACM SIGPLAN Notices, vol. 19(5). ACM Press, Pittsburgh, PA, 157–167.

UNGAR, D. M. AND JACKSON, F. 1992. An adaptive tenuring policy for generational scavengers. *ACM Transaction of Programming Languages and Systems 14(1)*, 1–27.

VESTAL, S. C. 1987. Garbage collection: An exercise in distributed, fault-tolerant programming. Ph.D. thesis, University of Washington, Seattle, WA.

WEIZENBAUM, J. 1962. Knotted list structures. *Communications of the ACM 5(3)*, 161–165.

ZORN, B. AND GRUNWALD, D. 1992. Evaluating models of memory allocation. Tech. Rep. CU-CS-603-92, University of Colorado at Boulder, Boulder, CO. July.

ZORN, B. G. 1989. Comparative performance evaluation of garbage collection algorithms. Ph.D. thesis, University of California at Berkeley, Berkeley, CA.