# Error-Free Garbage Collection Traces: How to Cheat and Not Get Caught*

Matthew Hertz† Stephen M Blackburn† J Eliot B Moss†
Kathryn S. McKinley‡ Darko Stefanović§

† Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{hertz,steveb,moss}@cs.umass.edu

‡ Dept. of Computer Science
University of Texas at Austin
Austin, TX, 78712
mckinley@cs.utexas.edu

§ Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131
darko@cs.unm.edu

## ABSTRACT

Programmers are writing a large and rapidly growing number of programs in object-oriented languages such as Java that require garbage collection (GC). To explore the design and evaluation of GC algorithms quickly, researchers are using simulation based on traces of object allocation and lifetime behavior. The *brute force* method generates perfect traces using a whole-heap GC at every potential GC 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 GC algorithms in two ways. First, we present a systematic methodology and results on the effects of trace granularity for a variety of copying GC algorithms. We show that trace granularity often distorts GC performance results compared with perfect traces, and that some GC algorithms are more sensitive to this effect than others. Second, we introduce and measure the performance of a new precise algorithm for generating GC traces which is over 800 times faster than the brute force method. Our algorithm, called Merlin, frequently timestamps objects and later uses the timestamps of dead objects to reconstruct precisely when they died. It performs only periodic garbage collections and achieves high accuracy at low cost, eliminating any reason to use granulated traces.

## 1. INTRODUCTION

While languages such as LISP and Smalltalk have always used garbage collection (GC), the dramatic increase of people writing programs in Java and other modern languages has seen a corresponding surge in GC research. A number of studies use object lifetime traces and simulations to examine the effectiveness of new GC algorithms [13, 19]. Others use traces to tune garbage collection via profile feedback [5, 6, 12, 16]. The demand for traces is sufficient that the GC research community is discussing a standard file format to enable sharing of traces [7].

Producing perfectly accurate traces is currently a very time consuming process; for many benchmarks (such as SPEC _202_jess, SPEC _213_javac, or SPEC _228_jack), the *brute force* method of producing traces can require over a month for each trace, since it performs a whole-heap collection at each allocation point. To reduce this cost, people often use *granulated* traces, which they generate by collecting after every *k* bytes of allocation. Unfortunately, researchers have not studied the effects of granularity on garbage collection simulations. While there has been research into better methods of approximating traces [18], the research did not study what effects these approximations have. We show here that granulated traces can produce *significantly* different results. Thus, past research based on the simulation of granulated traces may be problematic. This result also suggests a new requirement for any standard trace format: that it should include information recording the accuracy/granularity of the trace.

To address the efficiency problems of the brute force method and the accuracy problems of granulated traces, we propose the Merlin trace generation algorithm. The Merlin algorithm frequently timestamps live objects and later uses the timestamps to reconstruct the time at which the object died. Because it uses timestamps rather than collections to identify time of death, the new algorithm does not require frequent collections. Rather, it makes use of normal collections to identify *which* objects have died and then uses timestamps to identify *when* they died. Ordering the dead objects from the latest timestamp to the earliest, the algorithm works from the current time backwards. It thus determines when each object was last alive, saving the trace generator from having to process the object further. By avoiding frequent collections, the Merlin algorithm can run 800 faster than the brute force approach. It makes perfect tracing efficient and obviates the need for granulated tracing.

The remainder of this paper analyzes the effects of trace granularity on GC simulation fidelity for a number of GC algorithms and then introduces the Merlin trace generation algorithm. Section 2 gives some background on garbage collection, GC traces, and trace granularity. Section 3 describes the experimental methodology we used to analyze the effects of trace granularity. Section 4 presents the results of our granularity analysis and Section 5 discusses these

*This work is supported by NSF ITR grant CCR-0085792, NSF grant ACI-9982028, NSF grant EIA-9726401, DARPA grant 5-21425, DARPA grant F33615-01-C-1892 and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

results. Section 6 then introduces our new trace generation algorithm and describes how it improves on the existing algorithm. Section 7 presents and analyzes results from the new tracing algorithm. Finally, Section 8 presents related studies and Section 9 summarizes this study.

## 2. BACKGROUND

Three concepts are central for understanding this research: *garbage collection*, garbage collection *traces*, and garbage collection *trace granularity*.

### 2.1 Garbage Collection

Garbage collection automates reclamation of objects that are not needed from within the heap. While a wide variety of systems use garbage collectors, we assume a system that uses an implicit-free environment to make our explanations simpler, i.e., an explicit `new` command allocates objects, but there is no `free` command. Instead, an object is removed from the heap during a GC when the collector determines that the object is no longer reachable.

Since, without additional information, GCs cannot know which objects the program will use in the future, a garbage collector *conservatively* collects only objects it determines the program cannot reach and therefore cannot use now or in the future. To determine reachability, GCs begin at a program's roots. The roots contain all the pointers from outside of the heap into the heap, such as the program stack and static variables. Any objects in the heap not in the transitive closure of these pointers are unreachable. Since once an object becomes unreachable it remains unreachable (and cannot be updated or used), these objects can be safely removed from the heap.

In whole-heap collection, the collector determines the reachability of every object and removes all unreachable objects. Many collectors (e.g., generational collectors) often collect only part of the heap, limiting the work at each 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 also remain in the heap. Since objects in the uncollected region are not even examined, collectors use *write barriers* to find pointers into the collected region. The write barriers are instrumentation invoked at every pointer store operation. A write barrier typically tests if the pointer target is in a region that will be collected before the region containing pointer source and records such pointers in some data structure.

We assume that every pointer store is instrumented with a write barrier. In many systems this assumption is not true for root pointers, such as those in the stack. In this case, we enumerate the root pointers at each potential GC point, which is much less expensive than a whole-heap collection, and can be further optimized using the techniques of Cheng et al. [6].

### 2.2 Copying Garbage Collection Algorithms

We use four copying garbage collection algorithms for our evaluation: a semi-space collector, a fixed-nursery generational collector [15], a variable-sized nursery generational collector [3], and an Older-First collector [13]. We briefly describe each of these here for the reader who is unfamiliar with the GC literature.

A semi-space collector (SS) allocates into *From* space using a bump pointer. When it runs out of space, it collects this entire region by finding all reachable objects and copying them into a second *To* space. The collector then reverses *From* and *To* space and continues allocating. Since all objects in *From* space may be live, it must reserve half the total heap for the *To* space, as do the generational collectors that generalize this collector.

A fixed-nursery (FN) two generation collector divides the *From* space of the heap into a nursery and an older generation.[1] 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 (the older generation).

A variable-size nursery collector (VN) also divides the *From* space into a nursery and an older generation, but does not fix their boundary. In steady state, the nursery is some fraction of *From* space and when it fills up, VN copies live objects into the older fraction. The new nursery size is reduced by the size of the survivors. When the nursery gets too small, VN collects all of *From* space.

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

### 2.3 Garbage Collection Traces

A garbage collection trace is a chronological recording of every object allocation, heap pointer update, and object death (object becoming unreachable) over the execution of a program. These events include all the information that a memory manager needs for its processing. Processing an object allocation requires an identifier for the new object and how much memory it needs; pointer update records include the object and field being updated and the new value; object death records define which object became unreachable. These events comprise the minimum amount of information that GC simulations need.[2]

Simulators use a single trace file to analyze any number of different GC algorithms and optimizations applied to a single program run. The trace contains all the information to which a garbage collector would actually have access 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. Thus, researchers can simulate a single implementation of a garbage collector with traces from any number of different languages. For these reasons, GC simulators are useful when prototyping and evaluating new ideas. Since (non-concurrent) garbage collection is deterministic, simulations can return exact results for a number of metrics. When accurate trace files are used as input, results from a GC simulator can be relied upon, making simulation attractive and accurate traces critical.

Garbage collection 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 code for trace gen-

[1]The obvious generalization to $n$ generations applies.

[2]While some optimizations and collectors may need additional information, it can be added to the trace file so that the majority of simulations do not need to process it. Since most GC algorithms use only this information, here we assume only this minimal information.

eration instead of ordinary memory management code at each object allocation point and pointer update. The trace can log pointer updates by instrumenting pointer store operations; this instrumentation is particularly easy if the language and GC implementation use write barriers, since it then simply instruments those write barriers.

A reachability analysis of the heap from the program's root set determines object deaths. The common brute force method of trace generation determines reachability by performing a whole-heap garbage collection. Since the garbage collector marks and processes exactly the reachable objects, any objects unmarked (unprocessed) at the end of the collection must be unreachable and the trace generator produces object death records for them.

For a perfectly accurate trace, we must analyze the program at each point in the trace at which a garbage collection could be invoked. For most GC algorithms, collection may be needed whenever memory may need to be reclaimed: immediately before allocating each new object, assuming only object allocation triggers GC. Thus, brute force trace generators have the expense of collecting the *entire* heap prior to allocating *each* object. If the simulated GC algorithms allow more frequent garbage collection invocations, the reachability analyses must be undertaken more often, as well. These frequent reachability analyses are also difficult because of the stress they place on the system and how they expose errors in the interpreter or virtual machine.

## 2.4 Garbage Collection Trace Granularity

A common alternative to generating perfect traces is to perform the reachability analysis only periodically. Limiting the analysis to every *N* bytes of allocation makes the trace generation process faster and easier. It also causes the trace to be guaranteed accurate only at those specific points; the rest of the time it may over-estimate the set of live objects. Any simulation should 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.

Although trace granularity is related to time, its most appropriate unit of measurement depends on how GC is triggered. Since most collectors perform garbage collection only when memory is exhausted, the most natural measure of granularity is the number of bytes allocated between accurate points in the trace.

# 3. EXPERIMENTAL DESIGN

This section describes our methodology for evaluating experimentally the effect of trace granularity on simulating the four copying garbage collectors. We start by describing our simulator and programs. We then describe how to deal with granularity in simulation.

## 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 garbage collection algorithms: SS, FN, VN, and OF, as described in Section 2.2. The first three collectors are in widespread use. For each collector, we simulated eight different *From* space sizes from 1.25 to 3 times the maximum size of the live objects within the heap, at .25 increments. For FN and VN we simulated each heap size with five different nursery sizes and for OF with 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 works independently of the simulated GC algorithm. They explore the limits of trace granularity by affecting *when* the collections occur.

**Unsynced:** When we began this research, our simulator used this naive approach to handling trace granularity: it did nothing; we call this method *Unsynced*. Unsynced simulations allow a GC to occur at any time in the trace; collections are simulated 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. Unsynced simulations may treat objects as reachable because the object death record was not 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.

Three other schemes, which we call *Synced* (synchronized), simulate garbage collection invocations within the trace only at points with accurate knowledge of unreachable objects. The schemes check if a garbage collection is needed, or will be needed soon, only at the accurate points and perform a collection only at these points. Figure 1 shows how each of the Synced schemes makes collection decisions. In each of these figures, the solid line N is the natural collection point for the algorithm. The triangles denote points with perfect knowledge. The shaded region is as large as 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 G.

**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 period equal to one granule of the trace, SyncEarly forces a GC invocation. SyncEarly always performs a collection *at or before* the natural point is reached. SyncEarly simulations may 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 time of one granule of the trace, SyncLate invokes a garbage collection. SyncLate collects *at or after* the natural point is reached. SyncLate simulations may GC 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 Synced 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 the trace granularity in the past or future. SyncMid requires a collection at the point with perfect knowledge *closest* to the natural collection point. Doing this, 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 still enforced immediately following a collection.

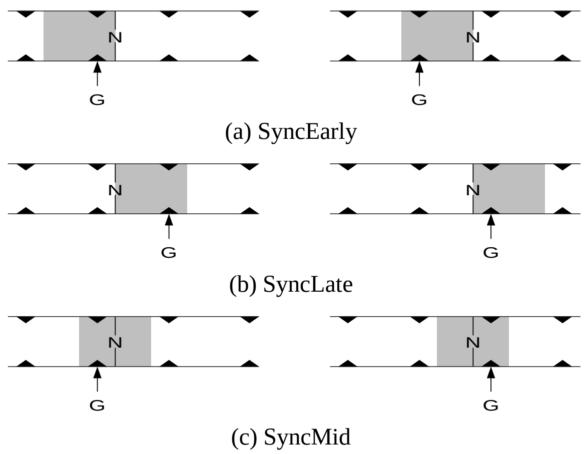


**Figure 1: When each of the Sync schemes decides to collect. The triangles denote points in the trace with perfect knowledge. The natural collection point is shown as the solid line labeled N. The shaded region is as large as one granule of the trace and shows the region in which garbage collection is allowed. A GC is forced at the point in the trace with perfect knowledge within the shaded region, shown by the arrow labeled G.**

# 4. TRACE GRANULARITY RESULTS

In this section, we present our data analysis and results.

## 4.1 GC Simulation Metrics

During a garbage collection simulation we measure a number of metrics: the number of collections invoked, the mark/cons ratio, the number of interesting stores, and the space-time product. Since the metrics we consider are deterministic, simulators can quite accurately return these results.

The mark/cons ratio is the number of bytes that the collector copied divided by the number of bytes allocated. The ratio serves as a measure of the amount of work done by a copying collector. Higher mark/cons ratios suggest an algorithm will need more time, because it must process and copy more objects.

Another metric we report is the number of interesting stores for a program run. Since many garbage collectors do not collect the entire heap, they use a write barrier to find pointers into the region currently collected (as we mentioned in Section 2.1). The write barrier instruments pointer store operations to determine if the pointer is one of which the garbage collector needs knowledge. The number of pointer stores, and the cost to instrument each of these, does not vary in a program run, but the number of pointer stores that must be remembered varies between GC algorithms at run time and will affect their performance.

We also measure the space-time product. While this is not directly related to the time required by an algorithm, it measures another important resource: space. This metric is the sum the number of bytes used by objects within the heap at each allocation point multiplied by the size of the allocation (or the integral of the number of bytes used by objects within the heap with respect to time measured in bytes of allocation). Since the number of bytes allocated does not vary between different algorithms, this metric measures how well an algorithm manages the size of the heap 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 the models developed that combine the data to determine the total time each algorithm needs [13].

## 4.2 GC Traces

We used 15 GC traces in this study. Nine of the traces are from the Jalapeño JVM (now know as the Jikes RVM) [2, 1], a compiler and run-time system for Java in which we implemented our trace generator. The nine Java traces are: bloat-bloat (Bloat [11] using its own source code as input), two different configurations of Olden health (5 256 and 4 512), and SPEC compress, jess, raytrace, db, javac, and jack. We also have six GC traces from the University of Massachusetts Smalltalk Virtual Machine. The Smalltalk traces are: lambda-fact5, lambda-fact6, tomcatv, heapsim, tree-replace-random, and tree-replace-binary. More information about the traces appears in Table 1.

We implemented a filter that takes perfect traces and a target value and outputs traces with the targeted level of granularity. We first generated perfectly accurate traces for each of the programs and then our filter generated 10 versions of each trace with granularity ranging from 1KB to 2048KB. Then our simulator used the perfect and granulated traces as input.

## 4.3 Analysis

We began by simulating all combinations of program trace, trace granularity, granularity scheme, GC algorithm, and *From* space and nursery (window) size. We record the four metrics from above for each combination. Table 2 shows an example of the simulator output. With this large population of data (approximately 600 simulations for each GC/granularity scheme combination), we perform a detailed statistical analysis of the results. For this analysis, we remove any simulation that required fewer than 10 garbage collections. In simulations with few GCs, the addition or removal of a single collection can create dramatically different effects and furthermore the garbage collector would rarely make a difference in the program's total running time. For these reasons, these results would rarely be included in an actual GC implementation study either. We also remove any simulation where the trace granularity equaled 50% or more of the simulated *From* space size, since trace granularity would obviously impact these results. We prune these cases, since the data will only bolster our claims that granularity is important. In addition, we only include simulations where both the perfect trace and the granulated trace completed. Occasionally, simulations of the granulated trace would complete merely because the simulator expanded the heap and delayed collection until an accurate point. There were also simulations of granulated traces that did not finish because garbage collection was invoked earlier than normal, causing too many objects to be promoted. Because any metrics generated from simulations that did not finish would be incomplete, we did not include them in our analysis. The number of experiments remaining at the 1KB granularity was about 90 for SS, 200 for VN, 250 for FN, and 425 for OF. The number of valid simulations does not vary more than by 2% or 3% until the 32KB granularity. At the 32KB granularity, there are 20% fewer simula-

| Program | Description | Max. Live | Total Alloc |
|---|---|---|---|
| bloat-bloat | Bytecode-Level Optimization and Analysis Tool 98 using its own source code as input | 3 207 176 | 164 094 868 |
| Olden Health (5 256) | Columbian health market simulator from the Olden benchmark suite, recoded in Java | 2 337 284 | 14 953 944 |
| (4 512) | | 1 650 444 | 9 230 756 |
| SPEC _201_compress | Compresses and decompresses 20MB of data using the Lempel-Ziv method. From SPECJVM98 | 8 144 188 | 120 057 332 |
| SPEC _202_jess | Expert shell system using NASA CLIPS. From SPECJVM98. | 3 792 856 | 321 981 032 |
| SPEC _205_raytrace | Raytraces a scene into a memory buffer. From SPECJVM98. | 5 733 464 | 154 028 396 |
| SPEC _209_db | Performs series of database functions on a memory resident database. From SPECJVM98. | 10 047 216 | 85 169 104 |
| SPEC _213_javac | Sun's JDK 1.0.4 compiler. From SPECJVM98. | 11 742 640 | 274 573 404 |
| SPEC _228_jack | Generates a parser for Java programs. From SPECJVM98. | 3 813 624 | 322 274 664 |
| lambda-fact5 | Untyped lambda calculus interpreter evaluating 5! in the standard Church numerals encoding | 25 180 | 1 111 760 |
| lambda-fact6 | Untyped lambda calculus interpreter evaluating 6! in the standard Church numerals encoding | 54 700 | 4 864 988 |
| tomcatv | Vectorized mesh generator | 126 096 | 42 085 496 |
| heapsim | Garbage collected heap simulator | 549 504 | 9 949 848 |
| tree-replace-random | Builds a binary tree then replaces random subtrees at a fixed height with newly built subtrees | 49 052 | 2 341 388 |
| tree-replace-binary | Builds a binary tree then replaces random subtrees with newly built subtrees | 39 148 | 818 080 |

**Table 1: Traces used in the experiment. Sizes are expressed in bytes.**

| gc num | alloc b | alloc o | copy b | copy o | xcopy b | xcopy o | garbge b | garbge o | mark/con | xcp/cp | mut. i/s | gc i/s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 5 221 236 | 148 532 | 1 098 480 | 27 504 | 268 088 | 5 558 | 3 770 048 | 121 022 | 0.210 387 | 0.244 054 | 14 243 | 0 |
| 10 | 9 230 756 | 353 094 | 1 552 152 | 48 481 | 284 404 | 6 379 | 6 622 732 | 278 931 | 0.168 150 | 0.183 232 | 40 675 | 0 |

(a) Perfect Trace

| gc num | alloc b | alloc o | copy b | copy o | xcopy b | xcopy o | garbge b | garbge o | mark/con | xcp/cp | mut. i/s | gc i/s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 4 787 328 | 125 037 | 1 443 608 | 32 306 | 355 768 | 7 173 | 2 824 328 | 92 722 | 0.301 548 | 0.246 444 | 11 644 | 0 |
| 11 | 9 230 756 | 353 094 | 200 7252 | 58 368 | 375 464 | 8 164 | 6 392 528 | 290 239 | 0.217 453 | 0.187 054 | 41 949 | 0 |

(b) SyncMid With 1KB Granularity

**Table 2: Simulator output from a fixed-sized nursery 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.**

tions. 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 lowest granularity.

The goal of this experiment is to determine if trace granularity affects GC simulations. To aggregate the remaining data, we normalize the granulated trace simulation results to the results of the simulation using a perfect trace with an identical configuration. In order that results that are too low and too high balance, we use the logarithm of this ratio. For each metric and combination of garbage collector and granularity scheme we performed two-tailed t-tests on the aggregated results. Following convention, we considered only differences at the 95% confidence level or higher ($p \leq 0.05$) to be statistically significant or more than the result of random fluctuations. When the t-test finds that the granulated results are significantly higher at the 95% confidence level we expect that if the experiment is repeated with similarly granulated traces, 95% of the time the means from these repeated experiments will also be larger than the results generated from perfect traces [10]. A similar argument exists for results that the t-test determine are significantly lower. Table 3 shows the smallest granularity, in kilobytes, at which we observe a statistically significant difference for each combination of collector, metric, and simulation method.

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 also 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 the brute force algorithm can generate perfect traces in under 8 hours. The traces remaining in this analysis are those for which brute force tracing 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 decrease by about 10% of OF and all the Unsynced simulations. The results of this analysis are shown in Table 4.

# 5. TRACE GRANULARITY DISCUSSION

The data in Table 3 are quite revealing about the effects of trace granularity and the usefulness of the different schemes in handling granulated traces. From these data it is clear that the use of granulated traces distorts GC performance results, compared with perfect traces. For a majority of the metrics, a granularity of only one kilobyte is enough to cause this distortion! Clearly, trace granularity significantly affects the simulator results.

## 5.1 Unsynced Results

Unsynced collections dramatically distort the simulation results. In Table 3, 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 GCs, and used more space. 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. Both collectors saw granulated traces producing significantly higher mark/cons ratios than the perfect traces. As one would expect, these distortions grew with the trace granularity. In Unsynced 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 prod-

| | Unsynced | | | | SyncLate | | | | SyncEarly | | | | SyncMid | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SS | FN | VN | OF | SS | FN | VN | OF | SS | FN | VN | OF | SS | FN | VN | OF |
| Mark/Cons | 1 | 1 | 1 | 1 | 1 | 8 | 16 | 4 | 1 | 1 | 4 | 4 | none | 1 | none | 1 |
| Space-Time | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | none | 1 | 2 | 1 |
| Num. of GCs | 1 | 1 | 16 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 4 | none | 1 | 16 | 1 |
| Int. Stores | n/a | 16 | 16 | 1 | n/a | 2 | 4 | 8 | n/a | 2 | 8 | 4 | n/a | 32 | 16 | none |

**Table 3: Earliest 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 | | | | SyncLate | | | | SyncEarly | | | | SyncMid | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SS | FN | VN | OF | SS | FN | VN | OF | SS | FN | VN | OF | SS | FN | VN | OF |
| Mark/Cons | 1 | 1 | 4 | 32 | 32 | 1 | 1024 | 16 | 8 | 512 | none | 8 | none | 512 | none | 64 |
| Space-Time | 4 | 1 | 512 | 1 | 16 | 1 | 512 | 512 | 1 | 1 | 512 | 2 | 1 | 1 | 512 | 32 |
| Num. of GCs | 32 | 1 | 512 | 16 | 16 | 1 | 64 | 8 | 64 | 1 | 512 | 8 | none | 1 | 512 | 1024 |
| Int. Stores | n/a | 512 | 2098 | 512 | n/a | 16 | 1024 | 16 | n/a | 32 | 1 | 8 | n/a | 16 | 1 | none |

**Table 4: Earliest 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 of 2MB or more**

uct and cause the heap to be full sooner, thus require more frequent GCs. This process snowballs, so that even small granularities quickly produce significant differences. Only the number of interesting stores for the generational collectors and the number of collections required for VN are not immediately affected. There are not significantly more pointers from the older generation to the nursery because Unsynced collections tend to promote objects that are truly unreachable and, therefore, do not have any pointer updates.

We expect simulations using larger heaps to be less affected by these issues. The results in Table 4 show that this is true. The space-time product and mark/cons results for SS show that objects are staying in the heap longer. For VN simulations, however, we do not see a significant increase in the number of collections; the extra objects require the collector to perform more whole-heap collections and not just nursery collections. Therefore each collection does more work: the number of collections remains similar to results with perfect traces by producing a significantly higher mark/cons ratio. No matter the collection algorithm, Unsynced simulations clearly distort the results. This result suggests a new requirement for the trace file format: it should clearly label the points in the trace with perfect knowledge.

## 5.2 Synced Results

Synced simulations tend to require slightly higher granularities than Unsynced before producing significant distortions. However, every Synced scheme significantly distorts the results for each metric for at least one collector. Examining the results from Table 3 and Table 4, reveals a few patterns. Considering all the traces, SyncEarly and SyncLate still produce differences from simulations using perfect traces, but slightly larger trace granularities may be required 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 at traces with granularities as small as 1KB. In Table 4, when considering only traces with larger maximum live sizes, Synced simulations provide better estimates of the results from simulating perfect traces. But, there still exist significant differences at fairly small granularities.

Because Synced simulations affect only when the collections occur, they do not copy unreachable objects merely because the object deletion 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 Synced collection point are initially affected. Depending on the Synced 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 Synced simulation may collect at points even further away (and make different collection decisions) than the simulation using perfect traces. Just as with Unsynced simulations, small initial differences can snowball.

**SyncEarly:** SyncEarly simulations *tend* to decrease the space-time products and increase the number of GCs, interesting stores, and mark/cons ratios versus simulations using perfect traces. At smaller granularities, FN produces higher space-time products. Normally, FN 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. 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 begins to limit the collectors as well. Not every collector produces the same distortion on the same metric, however. FN produces significantly higher mark/cons ratios and more garbage collections at small granularities. 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:** The best results we found are for SyncMid. From Table 4, the larger *From* space sizes produce similar results for SyncMid simulations and simulations using perfect traces at even large granularities. The design of SyncMid tries to balance the times that it collects too early with those times it collects too late. As a result, it tends to balance collections distorting the results in one direction and collections distorting results in the other. While this is a benefit, it also makes the effects of trace granularity hard to predict. Both SyncEarly and SyncLate showed collector-dependent behavior. While we showed that it would not be sound to base conclusions for a new or unknown collector from their results, one could make an assumption about their effect on the metrics. SyncMid simulations, by comparison, produced biases that were dependent upon both the metric and collector. When significant differences occur, it is not clear in which way the metric will be skewed. 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

From the above, it is clear that trace granularity has a significant impact on the simulated results of garbage collection algorithms. When using traces to compare and measure new GC algorithms and optimizations, there is not a clear way to use granulated traces and have confidence that the results are valid.

# 6. MERLIN TRACE GENERATION

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

In this section we present our new *Merlin Trace Generation Algorithm,* which generates perfect traces up to 800 times faster than the dominant brute force method of trace generation. Given the speed with which it can generate perfect traces, the Merlin algorithm removes the need to use granulated traces and avoids the issues that their use can cause.

The Merlin algorithm has other advantages over brute force trace generation. As discussed in Section 2.4, implementing the latter algorithm is difficult. For brute force to work, all GC and GC-affecting code must be *completely* error-free and the system must support whole-heap garbage collection. Our new trace generator can work with almost any garbage collection algorithm and stresses the system less.

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. The Merlin tracing algorithm works in a similar manner. Because it computes when each object died backwards in time, the first time the Merlin trace generation algorithm encounters an object in this calculation is the time of the object's death; any other possible death times would be earlier in the running of the program (but later in Merlin's processing), and need not be considered. Merlin, both the mythical character and our trace generator, works in reverse chronological order so that each decision, once made, never has to be revisited.

This remainder of this section overviews how Merlin computes when objects transition from reachable to unreachable, then gives a detailed explanation of why Merlin works, and discusses implementation issues. The method of finding object allocations and pointer updates is similar to the above description, but we describe how this works with the Merlin algorithm.

## 6.1 Merlin Algorithm Overview

The Merlin algorithm improves upon brute force trace generation by computing when objects were last reachable rather than when objects become unreachable. Knowing the last moment that an object was reachable, the death time for an object can be easily determined: since time advances in discrete steps, the death time of an object is the time interval immediately following the one in which the object was last reachable. By computing the last time objects are reachable, Merlin needs to perform only occasional garbage collections, saving substantial work.

To find when objects are last reachable, we stamp objects with the current time *whenever* they may transition from reachable to unreachable — whenever objects may lose an incoming reference. If the object later loses another incoming reference (because the earlier update did not leave it unreachable), then Merlin will simply overwrite the previous timestamp with the current time.

Now suppose that the system runs, performing occasional garbage collections. Consider the situation immediately following one of these GCs. The collector determines which objects are unreachable and which may still be live. For tracing purposes, we need to compute exactly *when* the unreachable objects were last reachable. The timestamps can be used to compute these times.

Consider a dead object with the latest timestamp. The object must have been last reachable at that time, for if it were reachable later, it would have been pointed to by an object with an even later timestamp — but this is the latest time. Now consider the pointers in the dead object with the latest death time. Any objects that are the target of these pointers would have also been reachable at the time stamped into the original object. Thus we propagate the last reachable time from the first object to the objects to which it points. In fact, we should propagate this last reachable time from a dead object to the objects to which it points until we can propagate it no further. To prevent infinite propagation through cycles, the algorithm simply stops if an object was last reachable at a time equal to or later than the last reachable time of the source object.

Once this processing is completed for the object with the latest timestamp, we have found the objects that were last reachable at that time. We can then remove them from the set of dead objects and consider the latest timestamp among the remaining objects. The last reachable time arguments apply iteratively, so we can determine this time for every object that the GC found was unreachable.

## 6.2 Merlin Details and Implementation

While the previous section provides an overview of Merlin, this section presents a detailed discussion of why the Merlin algorithm works and discusses implementation issues.

As discussed in Section 2.3, finding which objects are dead requires a reachability analysis. Our new algorithm cannot change this requirement, but instead improves upon the previous brute force method in computing the last instant that an object was reachable. To compute when objects were last reachable, the Merlin algorithm does a small amount of work as the program runs and when the trace must be accurate, and then performs less frequent GCs during trace generation.

After the system invokes a GC, the Merlin algorithm works backward in time to find exactly when each object the garbage collector found was unreachable was last reachable. In brute force trace generation, a death record is appended to the trace when an ob-

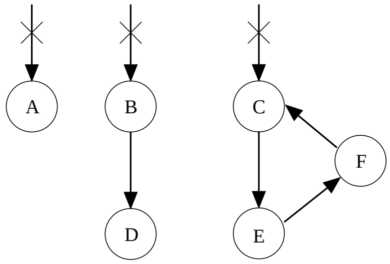



**Figure 2: Objects A and B are reachable until their last incoming reference is removed. Object C is last reachable when an incoming reference is removed, even though it has others. Objects D, E, and F are reachable until an action that does not affect their incoming references.**

ject is found to be unreachable. Whenever objects could be dead, the trace generator must find which objects are unreachable. Separating computing *when* objects were last reachable from *whether* objects are unreachable saves Merlin substantial amounts of work, but requires the introduction of *time* into trace generation. Where in the trace to add these "death" records is specified by the object's last reachable time. Time is related to trace granularity; time must advance wherever object death records may occur: at the points in the trace with perfect knowledge.

### *6.2.1 How Objects Become Unreachable*

To understand how the Merlin algorithm works backward in time to compute when an object was last reachable, it is important to understand how objects become unreachable. Table 5 is a series of generalizations about how objects within the heap transition from reachable to unreachable. Scenarios 1 and 2 of this table describe an object that is reachable until an action involves the object; Scenario 3 describes an object that is last reachable without it being directly involved in an action. Clearly, not all pointer stores are the last time an object is reachable, but any object that does become unreachable because of a pointer store must be in the transitive closure set of the object that lost an incoming reference.

### *6.2.2 Finding Potential Last Reachable Times*

Knowing how objects transition from reachable to unreachable and using the concept of time, is is now possible to find objects' last reachable time. Since it is not always clear if a pointer store is the last time an object is reachable (if a pointer update leaves an object with no incoming references, it is clear the pointer update is the last time the object is reachable; if an update leaves the object with $n$ remaining incoming references, it is not clear if the object continues to be reachable), just counting the number of incoming references (reference counting) is not sufficient to determine last reachable times. The following paragraphs consider the different methods by which objects transition from reachable to unreachable and present the Merlin pseudo-code to compute these last reachable times.

**Instrumented Pointer Stores:** Most pointer stores will be instrumented by a write barrier. Objects may be reachable until a pointer store, caught by a write barrier, removes an incoming reference. The Merlin trace generator stamps the object 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 4 stamps the object with the last time it was reachable.

1. An object transitions from one to zero incoming references via a pointer update. Objects A and B in Figure 2 are examples of this case.
2. An object transitions from $n$ to $n-1$ incoming references via a pointer update, where all $n-1$ references are from unreachable objects. An example of this case is object C in Figure 2.
3. An object's number of incoming references does not change, but all the reachable objects pointing to it become unreachable. The objects labeled D, E, and F in Figure 2 are examples of this case.

**Table 5: How objects become unreachable**

**Uninstrumented Pointer Stores:** Root pointers may not have their pointer stores instrumented. 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, our trace generator performs a modified root scan when the trace must be accurate. This modified root scan also enumerates the root pointers, but merely stamps the root-referenced 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 hold the last time the object was reachable. Figure 5 shows Merlin's pseudo-code 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 5). For chains of these objects, updating the last reachable time for one object requires recomputing the last reachable times of 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.

### *6.2.3 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 from Figures 4 and 5 timestamp the correct last reachable time for those objects that are last reachable as described in Scenarios 1 and 2 of Table 5. By combining the two timestamping methods with computing last reachable times by membership in transitive closure sets, Merlin can determine the last reachable time of every object.

To demonstrate that this combined method works, we consider each scenario from Table 5. Since no object continues to point to an object last reachable as described by Scenario 1 of Table 5 after it is last reachable, the latter object will only be a member of its own transitive closure set. Therefore, the last reachable time Merlin computes will be the object's own timestamp. The last reachable time computed for an object that is last reachable as in Scenario 2 of Table 5 will also be the time with which it is stamped. This object was last reachable when its timestamp was last updated. Since any objects that point to it must be unreachable, the pointing objects could not have later last reachable times. Thus, the transitive closure computation will determine the object was last reachable at the time with which it is already stamped. We show above that this combined method computes last reachable times for objects that are last reachable as in Scenario 3 of Table 5, 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.

#### 6.2.4 *Computing Death Times Efficiently*

Computing the full transitive closure sets is a time consuming process, requiring $O(n^2)$ time. But finding an object's last reachable time requires knowing only the *latest* object containing the former object in its transitive closure set. Rather than formally computing the transitive closure sets, 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, Merlin begins by ordering the objects from the earliest timestamp to the latest and then pushing them onto the search stack so the latest object will be popped first. Figure 3(a) shows this initialization. Upon removing an object from the stack, the Merlin algorithm analyzes its fields to find pointers to other objects. If a pointed-to object could be unreachable and is stamped with an earlier time than the referring object, then the pointed-to object is stamped with this later time. If the object is definitely unreachable, it is pushed onto the stack after its timestamp is updated (e.g., Figure 3(b) and 3(c)). If a pointed-to object's time is equal to that of the referring object, then either we have found a cycle (e.g., Figure 3(c)) or the pointed-to object is already on the stack to propagate this time. Either way, the pointed-to object does not need to be pushed on the stack. If a pointed-to object's time is later, then the object remained reachable after the time being propagated and this possible last reachable time is unimportant. 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; later examinations of an object are computing earlier last reachable times. This method of finding last reachable times requires only $\Theta(n \log n)$ time, the sorting of the objects being the limiting factor. Figure 6 shows the code the Merlin algorithm uses for this modified depth-first search.

### 6.3 The Merlin Trace Generator

As described so far, Merlin is able to reconstruct *when* objects were last reachable. However, it is unable to determine *which* objects are no longer reachable: it still needs a reachability analysis. The Merlin algorithm uses two simple solutions to overcome this. Whenever possible, it delays computation until immediately after garbage collection. Before any memory is cleared, the trace generation algorithm has access to objects within the heap *and* the garbage collector's reachability analysis. This piggy-backing saves a lot of duplicative analysis. At other times (e.g., when a program terminates), garbage collection may not be invoked but the algorithm needs a reachability analysis. We first stamp the root-referenced objects with the current time and then compute the last reachable times of every object in the heap. Objects with a last reachable time equal to the current time must be reachable from the program roots and therefore are still alive. All other objects are unreachable and their death records are added to the trace. 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, when the program terminates Merlin performs the combined object reachability / last reachable time analysis to find the unreachable objects and their last reachable times.

As stated in Section 2.1, we rely upon a couple of assumptions about the host GC. First, that any unreachable object the GC is treating as live will have the objects it points to treated as live, as is required among many GC algorithms. Thus no object is removed from the heap until all objects pointing to it are removed. 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 and Smalltalk satisfy them.

The order in which the Merlin trace generator adds information to the trace is an issue. As discussed in Section 6.2, our trace generator needs the concept of time to determine where in the trace each object death record should be placed. The object death records either must be added to the trace in chronological order before writing the trace to disk, or can be appended to the trace with a post-processing step placing the trace into proper order. Holding all the trace records in memory until all object deaths are found is a difficult challenge; with larger traces holding these records can require significant amounts of memory. Our implementation of the Merlin algorithm 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.

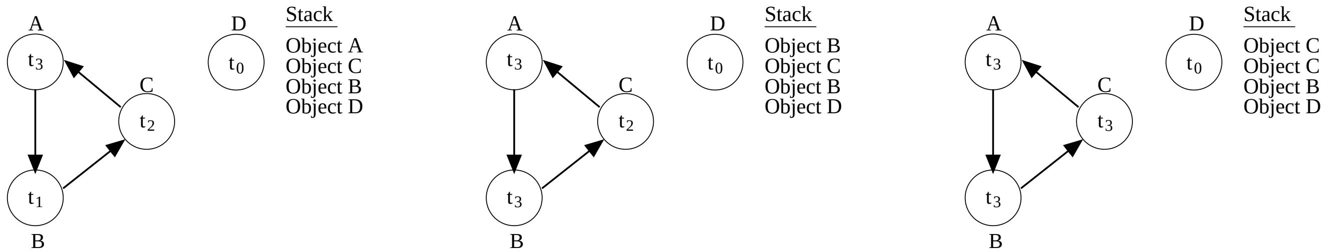
### 6.4 Object Allocations and Pointer Updates

Trace generation is already efficient at finding and reporting object allocations and pointer updates. As discussed in Section 2.3, even the brute force method of trace generation can find and record these actions in linear time. Our new algorithm, like those before it, instruments the host system's memory manager to determine when memory is allocated for new objects. At those times, Merlin records the ongoing object allocation.

Finding and reporting pointer updates also does not change. Like brute force trace generation, the Merlin algorithm instruments the heap pointer store operations (preferably by augmenting existing write barriers). Our new trace generation algorithm does add an additional requirement, the reasons for which are explained in Section 6.2.2. Unlike brute force, our trace generator requires access to the object being updated, the new value of the pointer, and the old value of the pointer. As many write barriers are already implemented to access these values (e.g., a write barrier capable of reference counting), this additional requirement is not a hardship. Allowing our trace generator to work with almost any garbage collector (rather than requiring a semi-space collector) makes the instrumentation to record pointer updates easier to add. While a semi-space collector does not require a write barrier, many algorithms (e.g., generational and OF collectors) do. Moreover, specific languages/systems require a write barrier for their own reasons. Combining our trace generator with these algorithms allows the use of the existing write barriers, enabling the Merlin trace generator to leverage this code.

## 7. EVALUATION OF MERLIN

We implemented both Merlin and the brute force trace algorithm within the Jikes virtual machine. We then performed some initial timing runs on a Macintosh Power Mac G4, with two 533 MHz processors, 32KB on-chip L1 data and instruction caches, 256KB




(a) Before Processing Object A (b) Before Processing Object B (c) Before Processing Object C

**Figure 3: Computing object death times, where $t_i < t_{i+1}$. Since Object D doesn't have any 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), Object C is processed. Object A is pointed to, but it does not have an earlier timestamp and is not added to the stack. After (c), 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);
```

**Figure 4: Code for Merlin's pointer store instrumentation**

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

**Figure 5: 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();
          Time objTime = obj.timeStamp;
          if (objTime <= lastTime)
               lastTime = objTime;
               for each (field in obj)
                    if (isPointer(field) && obj.field ≠ null)
                         Object target = getMemoryWord(obj.field);
                         Time targetTime = target.timeStamp;
                         if (isUnreachable(target) && targetTime < lastTime)
                              target.timeStamp = lastTime;
                              stack.push(target);
```

**Figure 6: Code of Merlin trace generation last reachable time computation**

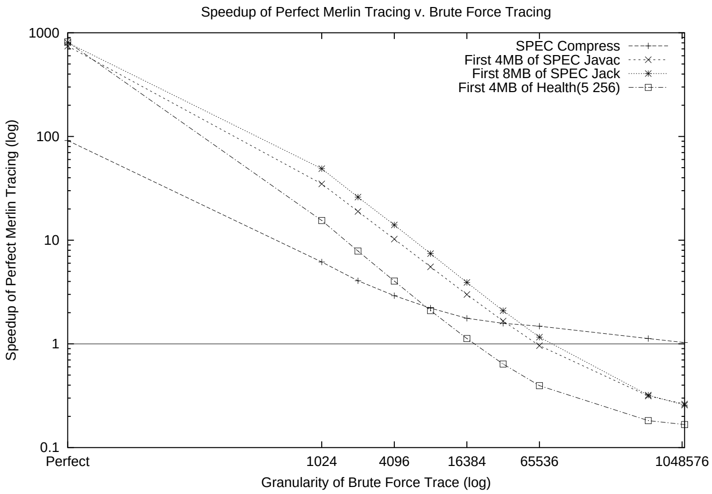



**Figure 7: The speedup of Merlin versus Brute Force trace generation. Note the log-log scale.**

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, one for each of the algorithms. Whenever possible we used identical code for the two JVMs, so Merlin is implemented with a semi-space collector.

Merlin's running time is spent largely in performing the modified root scan that is required at every accurate point in the trace. We further improved Merlin's running time by including a number of optimizations that minimize the number of root pointers that must be enumerated at each of these locations. The first optimization was to instrument pointer store operations involving static (global) pointers. With this instrumentation Merlin does not need to enumerate the static pointers at each accurate point, as the instrumentation marks objects whenever they lose an incoming reference from the static fields. 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 the stack less deep and further reduce the time needed for Merlin tracing [6]. Because they would not improve brute-force tracing, these optimizations were used only with Merlin tracing.

We generated traces at different granularities across a small range of programs. Because of the time required for brute force trace generation, we limited some traces to only the initial few megabytes of data allocation. Working with common benchmarks and generating traces of identical granularity, Merlin achieved speedup factors of up to 816. In the time that brute force needed to generate traces with 16 to 1024KB of granularity, Merlin generated perfect traces. Figure 7 shows the speedup Merlin, generating perfect traces, achieves over the brute force algorithm generating traces at different levels of granularity. Clearly, Merlin can greatly reduce the time needed to generate a trace. However, as seen in Figure 7, the speedup is less as granularity increases. The time required depends on the time needed to generate object death records and, therefore, on trace granularity. Brute force limits object death time processing to only when the trace must be accurate; as the granularity increases the time needed greatly diminishes. While Merlin needs to perform only periodic collections, it also must perform a small set of actions at each pointer update and location in the trace with perfect knowledge. Even with brute force performing more frequent GCs, the cost of Merlin's frequent root enumerations and updating timestamps becomes too great.

These results are promising, but we can speed up performance of the Merlin tracing algorithm even more. As a program's memory footprint grows, and as more accurate points are needed, the Merlin algorithm is far less affected than brute force.

## 8. RELATED WORK

We do not know of any previous research into the effects of trace granularity or different methods of generating garbage collection traces. In this section, we discuss the research from which this study draws its roots.

**Using Knowledge of the Future:** Belady's [4] optimal virtual memory page replacement policy, MIN, decided which blocks should not be paged to disk by analyzing future events. At each decision point, the MIN algorithm considers future memory accesses, stored within an available file, until it determines the single block to evict. Because the algorithm did not cache results, at each decision point the MIN algorithm begins a new analysis. While Belady's algorithm used knowledge of future events to perform optimally, it processes events in chronological order. Each time it is invoked, the MIN algorithm looks only far enough into the future as is necessary to make the current decision.

**Cyclic Reference Counting:** One of the earliest methods of garbage collection was to use reference counts: each object has a count of its incoming references so, when the count reaches 0, the object can be freed [8]. McBeth was the first to appreciate that this approach cannot collect cycles of objects, since the reference counts would never reach zero [9]. Many different schemes have been developed to deal with cycles. Trial deletions [17] collects cycles of objects by removing a pointer thought to be within a cycle. After removing the pointer, trial deletion updates the reference counts. If, in updating the reference counts, the source object for the removed pointer is found unreachable, then a cycle exists and the objects are dead. Otherwise a dead cycle may not exist, the deleted pointer is reestablished and the original reference counts restored. This method can handle and detect cycles, but it may incorrectly guess that some objects are in a cycle and cannot take advantage of other object reachability analyses.

Merlin does not perform any explicit reference counting, though it marks objects whenever they lose an incoming reference. Generally, reference counting methods cannot properly determine when cycles of objects become unreachable. While methods, like trial deletion, have been developed to avoid this problem, these methods cannot guarantee that they will determine when each object is unreachable in addition to processing each object only once. Using Merlin, as opposed to reference counting, allows both of these requirements to be met.

**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 [14]. Zorn and Grunwald compare several different models one can use to approximate object allocation and object death records of actual programs [18]. Neither study attempted to generate actual traces, nor does either study consider the effects of pointer updates; rather, these studies attempted to find ways other than trace generation to produce input for memory management simulations.

## 9. SUMMARY

The use of granulated traces for garbage collection simulation raises a number of issues. We first develop a method by which any variable that affects garbage collection simulations can be statistically tested. 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. Additionally, we show that there are ways of simulating granulated traces that are better at minimizing these issues. With these results, we propose changing the trace format standard to include additional information.

Finally, we introduce and describe the Merlin Trace Generation Algorithm. We show that the Merlin algorithm can produce traces more than 800 times as fast as the common brute force method of trace generation. By generating traces with Merlin, we can generate perfect traces in less time than previously required for granulated traces. Thus, the Merlin algorithm makes trace generation quick and easy, and eliminates the need for granulated traces.

**Acknowledgments** We would like to thank John N. Zigman for his work in developing *gc-sim* and also Aaron Cass for his help processing much of this data.

## 10. REFERENCES


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

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

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

[4] BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal 5(2)* (1966), 78–101.

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

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

[7] CHILIMBI, T., JONES, R. E., AND ZORN, B. Designing a trace format for heap allocation events. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management* (Minneapolis, MN, Oct. 2000), vol. 36(1) of *ACM SIGPLAN Notices*, ACM Press, pp. 35–49.

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

[9] MCBETH, J. H. On the reference counter method. *Communications of the ACM 6(9)* (Sept. 1963), 575.

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

[11] NYSTROM, N. Bytecode-level analysis and optimization of Java classfiles. Master's thesis, Purdue University, West Lafayette, IN, May 1998.

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

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

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

[15] UNGAR, D. M. 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* (Pittsburgh, PA, Apr. 1984), vol. 19(5) of *ACM SIGPLAN Notices*, ACM Press, pp. 157–167.

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

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

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

[19] ZORN, B. G. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Berkeley, CA, Mar. 1989.