Elephant Tracks: Portable Production of Complete and Precise GC Traces

Nathan P. Ricci
Tufts University
nricci01@cs.tufts.edu

Samuel Z. Guyer
Tufts University
sguyer@cs.tufts.edu

J. Eliot B. Moss
University of Massachusetts Amherst
moss@cs.umass.edu

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory Management–(garbage collection); C.4 [*Performance of Systems*]: [measurement techniques]

General Terms Languages, Measurement

Keywords Traces; garbage collection; Merlin algorithm; Java

Abstract

We present *Elephant Tracks* (ET), a dynamic program analysis tool for Java that produces detailed traces of garbage collection-related events, including object allocations, object deaths, and pointer updates. Like prior work, our tracing tool is based on the Merlin algorithm [6, 7], but offers several substantial new capabilities. First, it is much more precise than previous tools: it traces method entries and exits and measures time in terms of them, allowing it to place events precisely in the context of the program structure. Second, it is implemented using a combination of JVM Tool Interface (JVMTI) [13] callbacks and bytecode rewriting, and works with any standard JVM. Finally, it produces complete traces, including weak references, events from the Java Native Interface and `sun.misc.Unsafe`, and VM start up objects. In this paper we also explore the general design space of tracing tools, and carefully define the execution model that the traces represent.

1. Introduction

Garbage collection tracing tools have been instrumental in the development of new garbage collection algorithms. A GC tracing tool produces an accurate trace of all the dynamic program events that are relevant to memory management, including allocations, pointer updates, and object deaths. We can quickly test a new GC algorithm by building a simulator that reads the GC trace, instead of developing a full GC implementation in a real virtual machine, which is a considerable undertaking.

One of the widely used GC tracing tools for Java, GCTrace, is available as a component of the JikesRVM Java virtual machine [2]. That tool, like ours, is based on the Merlin algorithm [6, 7], but suffers from several limitations. First, the implementation is integrated directly into the garbage collector. Due to the ongoing evolution of the JikesRVM Memory Management Toolkit, it no longer functions with recent versions of JikesRVM, and older versions will not run modern Java software. Second, GCTrace measures time only in terms of bytes allocated, a fine metric for GC simulation, but not useful for program analysis since it cannot readily be tied back to points in the program. Third, allocation time is not very precise for events other than allocation: many pointer updates and object deaths can occur at various points in between two allocations. Finally, the existing tool does not support a number of features found in real programs, including weak references and multithreading.

In this paper we present *Elephant Tracks*, a new GC tracing tool that is precise, informative, and can run on top of any standard JVM. Our goal is not simply to address the limitations of prior work, but to provide new capabilities that allow our tool to support a wider variety of program analysis and runtime systems research. Our implementation uses a combination of bytecode instrumentation and JVMTI (JVM Tool Interface) callbacks to construct a shadow heap on which it runs the Merlin algorithm to compute idealized object death times. Its attributes include:

Precise: Trace time is configurable and can be made arbitrarily precise. We currently measure time in terms of method calls (i.e., the clock ticks at every method entry and exit), which are *much* more frequent than allocations.[1]

Complete: The implementation properly handles all relevant events, including difficult cases, such as weak references, the Java Native Interface, `sun.misc.Unsafe`, and JVM start up objects.

Informative: Traces include much more than just GC-related events. We emit a record for every method call and return, allowing us to tie memory behavior back to the program structure. In fact, we can reconstruct the complete dynamic calling context for any time step. We also record information about threads and exceptions, and, optionally, counts of heap reads and writes and number of bytecodes executed.

Well-defined: We carefully define the trace execution model, which embodies a number of subtle design issues that affect the meaning of the traces. These include trace time, the definition of object lifetime, and the ordering of events in multithreaded programs. In this paper we explore these issues in detail and justify our choices.

Portable: Elephant Tracks is implemented as a JVMTI agent that runs on any compliant Java virtual machine.

Fast: OK, not fast, but faster. Elephant Tracks includes performance tuning and optimizations to reduce overhead, which is critical for larger, long-running programs.

In the following sections we explore the design space of GC tracing tools, and explain the choices made for Elephant Tracks. We

ISMM '13, June 20, 2013, Seattle, WA.
Copyright © 2013 ACM 978-1-4503-2100-6/13/06. . . $10.00

[1] This includes constructor calls, thus tightly bounding most allocations as well.

discuss the technical challenges of building such a tool using the JVMTI interface, which does not provide direct access to the JVM's representation of Java Objects, or to the garbage collector. We also discuss the handling of weak references. This proved difficult because the JVM is able to side-step some of our instrumentation techniques in this case. Finally, we present some results, including overhead measurements, as well as new insights about the benchmarks gleaned from our precise traces.

2. Background and related work

In this section we describe the general garbage collection tracing problem and existing solutions, and motivate the need for a new trace generator.

2.1 Garbage collection tracing

A GC trace is a record of the sequence of memory management events that occur during a program's execution. The events of interest may vary depending on the intended use of the trace, but typically include object allocation, object reclamation, and mutations in the heap. Many of these events are straightforward to capture, such as object allocation, since they are explicitly invoked in the code.[2] We can instrument those operations directly to emit a trace record with the relevant information.

The central challenge in GC tracing is determining object death times. An obvious solution is to emit an object death record when the garbage collector actually reclaims each object. This approach is easy to implement using JVMTI, but is unappealing for at least two reasons. First, the particular timing of these events is collector-specific—we would be measuring a property of the GC algorithm used during trace generation, rather than a fundamental property of the program. Second, the resulting information is very imprecise. Most garbage collectors run infrequently, reclaiming large numbers of objects long after they are no longer needed by the program. As a consequence, object deaths appear far removed from the program events that actually cause them. This makes the traces poorly suited to evaluating new GC algorithms, as the Merlin work showed [7].

2.1.1 Idealized death times

Our goal is to generate traces with idealized death records. That is, each object death appears in the trace at the earliest time at which the object *could be* reclaimed. An idealized trace captures the behavior of a program independent of any particular GC algorithm, with object death events appearing close to the program actions that cause them. The exact nature of this problem depends on how we define "the earliest time". For example, we could compute death times based on object *liveness*: an object is dead immediately after its last use. While interesting as a lower bound, this level of precision is potentially expensive to compute and cannot be exploited by any real memory manager. Therefore, we adopt the definition used in garbage collection and in prior work on tracing: an object is dead when it is no longer *reachable* from the roots (local and global variables) directly or indirectly through any sequence of pointers. This choice still leaves many subtle issues, however, including the granularity of trace time and the liveness of root variables, which are discussed in more detail in Section 3.

A naive algorithm for computing idealized death times is to run the garbage collector much more frequently. For example, we could produce a very precise trace by invoking the collector at every program point where an object could become dead. Not surprisingly, this approach is totally impractical.

[2] This is more subtle than you might think. In Java, the virtual machine may allocate application-visible objects as side-effects of other actions, such as class loading, and native libraries can also do so. Similar remarks apply to pointer updates.

2.1.2 The Merlin algorithm

The Merlin algorithm, introduced by Hertz et al. [6], solves this problem by using timestamps to *infer* the idealized death times of objects when they are reclaimed at regularly scheduled garbage collections. During normal execution the algorithm timestamps objects whenever they lose an incoming pointer. At any point in time an object's timestamp represents the last time it was *directly* observed to be reachable. When the collector reclaims an object, however, its timestamp is not necessarily its death time. In many cases an object becomes unreachable *indirectly*, when an object that points to it becomes unreachable. In this case we need to determine which event occurred later: the direct loss of an incoming pointer (the timestamp), or the indirect loss of reachability (the death times of the referring objects). So, the idealized death time of an object ($T_d(o)$) is computed from its timestamp ($T_s(o)$) and the death times of any objects that point to it:

$$T_d(o) = Max(T_s(o), \{T_d(p), \forall p : p \rightarrow o\})$$

This insight leads to a practical approach for GC tracing that is also at the heart of the system we present in this paper:

- During normal execution:
  - Record ordinary events in the trace as they occur (e.g., object allocations and pointer updates).
  - Timestamp objects whenever they might become directly unreachable (i.e., when they lose an incoming pointer).
- At GC time:
  - Compute idealized death times using the formula above (implemented roughly as a depth-first search on the graph of dead objects, pushing computed death times across the pointers).
  - Generate a death event record for each reclaimed object and insert it in the proper place in the trace.
  - Flush records to disk, and continue ...

An important implication of the Merlin algorithm is that it requires a notion of *trace time* for use in the timestamps. In factm all trace records need timestamps because the object death records are generated out of order—we discover the true death times of objects only at GC time, which is typically much later. The model of trace time (in particular, its granularity) has a profound impact on the implementation of the trace generator and the precision of the traces it generates.

2.2 Why a new trace generator?

The first realization of Merlin took the form of a customized garbage collector called GCTrace, implemented in JikesRVM. The main advantage of this approach is that the implementation can be integrated directly into the virtual machine code. The compiler can be modified to add instrumentation to object allocations and pointer updates, and the garbage collector can be modified to perform the extra death time computation. While GCTrace has proved to be a valuable tool, it has a number of serious limitations, several of which are a consequence of its dependence on JikesRVM:

Imprecise: GCTrace uses *allocation time* for its traces: the trace time clock "ticks" at each object allocation. As a result, object deaths and other events that occur in between allocations cannot be ordered or precisely localized at any finer granularity.

Divorced from program structure: A related problem is that allocation time does not correspond to anything static in the program itself, so figuring out where events occur relative to the code is very difficult (e.g., "In which method did the death of object 739229 occur?").

Incomplete: GCTrace ignores a number of difficult corner cases, including multithreading, JNI, and the various forms of weak references.

Java-in-Java: Because JikesRVM is implemented in Java, great pains must be taken not to include VM events and objects in the trace. Furthermore, parts of the algorithm, such as sorting death records, are extremely painful to implement because they must run inside the garbage collector and therefore cannot allocate any ordinary data structures themselves.

Performance: While all trace generators are likely to be slow, GCTrace is particularly slow for several reasons. First, it performs local variable timestamping using an expensive stack walk at every time step (every allocation). Second, it works only with the non-optimizing compiler. Third, it supports only a simple whole-heap garbage collector.

Application limitations: While in principle any Java application runs under any JVM, in practice there are variations. JikesRVM, which is maintained by volunteers, tends to lag commercial implementations to some extent, so there are applications of interest that run on commercial JVMs but not on JikesRVM.

Bit rot: MMTk (the memory management toolkit used in JikesRVM) has undergone a number of radical refactorings, often leaving the GCTrace implementation out of date.

2.3 Related work

There is a huge body of work on tracing programs to produce a record of various runtime events. The work most closely related to ours is the original GCTrace implementation of the Merlin algorithm [7], which is discussed in detail throughout this paper. Foucar reimplemented GCTrace using a shadow heap implemented in C++, like Elephant Tracks, but otherwise preserving the execution model and dependence on JikesRVM [4].

Another potential approach is to use non-deferred reference counting, which reclaims objects as soon as their reference counts becomes zero. Like reference counting collection, however, this approach cannot directly detect the death of cycles of objects, and would require frequent tracing collections to achieve high precision.

Two prior papers explore the relationship between liveness and reachability for garbage collection. Agesen et al. [1] examine the effects of applying different levels of liveness analysis to the GC root set (variables on the stack). They found that on average the differences were small, but on occasion static liveness analysis would improve collection efficiency noticeably. This result suggests that our dynamic liveness model is reasonable for most purposes, but could be improved (see later discussion). Hirzel et al. [8] additionally consider the difference between reachability from live roots and true liveness of objects. They also find that schemes based on liveness of variables have little impact on reachability. True object liveness, however, is significantly different. Elephant Tracks currently cannot compute equivalent information, since it does not record reads from objects, but there is no fundamental impediment to adding this feature.

GC traces have been used to drive empirical studies of heap behavior, especially those examining the distribution and predictability of the lifetimes of objects [10, 11]. At a coarse level, allocation time and method time do not produce dramatically different lifetime distributions. For analyses that are sensitive to program structure, however, small differences in allocation time can span many methods. In addition, allocation time is not stable across runs of a program under different inputs.

Jones and Ryder [11] offered perhaps the most well-known study of object demographics. They show that the calling context of object allocation correlates well with lifetime. They could not determine, however, whether the calling context of object *death* correlates with lifetime, which might be a more useful fact for further improving garbage collection.

Inoue et al. [10] look at what information is needed to precisely predict the lifetime of an object at its allocation point. They define a *fully precise* predictor as one that is accurate to within a single unit of time. By using allocation time, however, they significantly reduce the coverage and accuracy of their predictors. The lifetime of an object in allocation time is much less stable than the calling context of its death, since the latter is directly related to its cause of death, while the former includes many irrelevant events (i.e., unrelated allocations). This instability is particularly acute across runs of a program with different inputs.

Compile-time GC [5] and connectivity-based garbage collection [9] are two examples of techniques where knowing the program location at which an object dies is crucial. Such techniques are often evaluated using trace-driven simulation before embarking on a full implementation. Using Elephant Tracks traces would yield a more accurate assessment of their potential.

Lambert et al. present a system for performing platform-independent JVM timing [12]. Although similar in spirit to our JVM-independent execution model, the focus of this work is on developing a model of code execution, rather than heap memory behavior.

Uhlig and Mudge [14] present a survey of memory tracing techniques. While their focus is on tracing memory accesses for architecture and system research, they enumerate a set of features they consider desirable for tracing systems in general: completeness (all relevant events are recorded), detail (events are associated with program-level information), low distortion (tracing does not change the program's behavior), and portability. Elephant Tracks achieves many of these goals, although it significantly distorts actual running time, however, which is why we use a separate notion of time.

3. Elephant Tracks Design

Our goals in designing a new trace generator are to address the limitations of prior systems and to add new functionality to support new kinds of program analysis and memory management research. The central features of this design are (1) the kinds of program events recorded in a trace, and (2) the accuracy of this information with respect to some model of program execution. In this section we present the design of Elephant Tracks, and we discuss our choices in the context of the general GC tracing design space. In Section 4 we describe how this design is implemented.

3.1 Kinds of trace records

A minimal GC trace consists of just a sequence of object allocations and object deaths, labeled with the trace time and thread ID of each event. Without more information, though, such a trace has limited utility. In practice we add trace records for other kinds of relevant events to provide context for program analysis and to enable more kinds of trace-based simulations. For garbage collection research, for example, it is useful to add trace records for pointer updates in the heap, allowing a simulator to maintain an accurate heap model. Elephant Tracks can be configured to produce different kinds of trace records. We currently support the following kinds of records:

- Object allocations and object deaths (with idealized death times computed using the Merlin algorithm).

- Pointer updates in the heap: These records include the source and target objects, as well as the object field or array index being updated. We also include updates of static fields.
- Method entry and exit: These records allow trace times to be mapped to specific methods, and even more precisely, to specific calling contexts.
- Exceptions: We augment method entry and exit to indicate when an exception is thrown, the sequence of method calls (if any) that are terminated early because of the exception, and the entry to a handler for the exception. The main purpose of these events is to provide accurate information about method execution context.
- (Optional) Heap read/write counts: Each time the clock ticks or a basic block ends, we generate a counts record that summarizes the number of heap reads and writes (of pointers and non-pointers), and the number of bytecodes executed, since the last counts record. Note, however, that multiple counts records can occur between clock ticks, and they cannot be ordered with respect to object deaths in the same tick.

Separately from the trace, Elephant Tracks also emits information about each class loaded, each field declared in the class, each method declared in the class, and each allocation site in each method. This information is referred to by the trace, e.g., the trace will mention a unique allocation site number, which can be found in the side description file.

We currently do not generate trace records for object timestamps or for general memory accesses (including stack reads and writes). This information would enable an even wider range of applications, such as cache simulations. These events are extremely frequent, however, and would result in overwhelmingly large traces. In addition, instrumenting every single variable access would be technically challenging—bytecode rewriting might not be the best approach for this level of detail.

3.2 Execution model

Ideally, we would like to generate perfect traces, in which every event is recorded with a perfectly accurate and precise time. But this goal raises a critical question: accurate with respect to *what*? That is, what is the *execution model* that we want the trace to represent? Elephant Tracks, like other trace generators, relies on a host virtual machine to execute the target program. It runs alongside the VM, recording relevant events. The problem is that the timing of some events is highly VM dependent—directly recording these events as they occur produces a trace that has the VM's execution model "baked in." Instead, we want to generate traces that have their own well-defined, less VM-specific, execution model. The possible models range widely, with some elements closer to the VM (essentially profiling the VM), and other elements more abstract, capturing an idealized execution of the program.

The main components of a GC tracing execution model are (1) the definition of object lifetime (in particular, when are objects considered dead), and (2) the definition of trace time (i.e., when does the trace time clock "tick" and with what frequency). The overall goal of the Elephant Tracks execution model is to define these components in such a way that events can be localized precisely within the structure of the code. The model is idealized for object lifetimes, but resorts to VM timing in cases where an idealized model is not possible, such as the interleaving of concurrent threads and the clearing of weak references.

3.2.1 Defining object lifetime

Object lifetimes are delineated by allocation and death events. Most object allocations are explicit in the program, so simply recording them as they occur produces a VM-independent trace. We have found, however, that there are several other sources of allocations, including VM internal allocations (e.g., `String` constants in class files and `Class` objects themselves), objects allocated by the VM before it can even turn instrumentation on, and objects allocated by JNI calls. We capture all of these, but cannot associate them with a usual allocation site in the application, and for those allocated very early in the run, we cannot provide relative time or context of allocation.

For object deaths, however, an explicit goal of GC tracing is to compute idealized death times. Both Elephant Tracks and GCTrace adopt the standard GC definition: an object is dead when it is no longer reachable from the roots (local and global variables). Even within this seemingly narrow definition, however, there are a range of possible models. To see why, consider the program events that can cause an object to become unreachable:

- The program overwrites a pointer in the heap (`putfield`, etc.)
- The program overwrites a static (global) reference (`putstatic`)
- A local reference variable goes out of scope
- The program changes the value of a local reference variable
- A weak reference is cleared by the garbage collector

While the first two (heap and global writes) are straightforward to instrument, local variables and weak references are more difficult to pin down. Furthermore, there are roots inside the VM that we cannot observe and that the VM does not necessarily inform us about when they change. Fortunately these are mostly "immortal" references, such as to class objects, or relate to constants constructed from class files (these may come and go).

Local variables

Tracing local variables presents many design choices and challenges. The key question is: at what point is a local pointer variable dead, and therefore no longer keeping the target object alive? At one end of the spectrum we could consider local variables live throughout the method invocation with which they are associated. In practice, however, most virtual machines apply some form of static liveness analysis to compute more precise lifetimes. The virtual machine uses this information to construct GC maps, which tell the garbage collector which variables to consider as GC roots at a given point in the method.

GCTrace uses the GC maps in JikesRVM to determine which variables are live. The advantage of this approach is that it is straightforward to implement. The downside is that the timing of the object death records depends on the specific liveness analysis algorithm and choice of GC points made in JikesRVM.

Elephant Tracks currently uses a form of dynamic liveness to determine the lifetimes of local variables. This choice reflects implementation decisions (described in more detail below). A variable is considered dead after its last dynamic use. We define a *use* as one of the following: (1) a direct dereference (access to an object or array), (2) a type test, such as `instanceof`, (3) obtaining an array's length, (4) use as a receiver of a dynamic method dispatch, or (5) a reference test, such as `ifnull`.

Dynamic liveness, however, is more precise than static liveness analysis, primarily because it is not conservative about liveness on different execution paths. The resulting traces show some object death times earlier than any real garbage collector could achieve. For example, a reference variable that is passed through a series of methods, but never used, is considered dead in all the methods. As partial compensation we consider a variable live if it is passed to a method call as a parameter, or returned.

In the future, we plan to add one or more reference implementations of static liveness analysis that allows us to control the model of variable lifetimes more precisely, and thus model more closely what idealized optimizing and non-optimizing compilers might do. For example, an idealized non-optimizing compiler would keep a variable live as long as its type, as computed according to the JVM specification, is a pointer type. On the other hand, an idealized optimizing compiler would apply a backwards data flow analysis to determine a static estimate of liveness. However, real compilers may transform code in various ways, such as inlining methods and duplicating tails to form superblocks. While we admit that such transformations occasionally affect liveness for some objects, at the same time we contend that the vast majority of cases will be approximated well, for purposes of evaluating GC algorithms, by a suitable idealized model. Elephant Tracks is a good foundation from which to explore questions like this.

Weak references

Weak references present an interesting challenge. In principle, the garbage collector can choose to clear weak references at any time (or not at all) once an object is no longer strongly reachable. In practice, they will only be cleared when the collector is actually run. Further, soft references are cleared "at the discretion" of the collector, in response to memory pressure. Phantom references are similarly affected by the timing of collector runs by the host VM. For a trace, though, this leaves no obvious idealized model of when to clear a weak reference. Both Elephant Tracks and GCTrace opt to record these events when the VM chooses to perform them. Given that programs can perceive and respond to the collector's decision, there is no good alternative to this approach.

3.2.2 Trace time

For Merlin-based tracing we need a notion of trace time, so that object death records, which are generated only at GC time, can be inserted in their proper place in the trace. The choice of trace time has a profound affect on the implementation of the tool and on the resulting traces.

Real time is a bad choice, since it is dependent on many factors, including the virtual machine, the operating system, and the hardware. In addition, tracing tends to slow programs down significantly, so the real times are likely to be significantly different from uninstrumented runs. Real time is also, in some sense, too precise: we do not want the trace to reflect the time it takes to actually perform a timestamp or record a trace record.

The solution is to express time in terms of some program-level event: each time the event is encountered we tick the trace clock. In this way, time depends only on a property of the program, not on the VM or underlying machine. This model breaks time into discrete steps, each of which represents a small region of program execution.

The choice of which event(s) to use for the clock affects the granularity of time, which ultimately determines the precision of the trace, since trace records labeled with the same time cannot be ordered or localized within the region covered by that time step. The trade-off is that more fine-grained notions of time are more difficult to implement correctly, since we need to place the instrumentation more precisely to make sure that every event is labeled with the correct time. They may also incur more overhead.

Allocation time vs method time

GCTrace measures time in terms of the number of bytes allocated since the program started (called *allocation time*). At each allocation, time advances by the number of bytes allocated. Allocation time is good for basic GC research, since the traces are precise enough to drive simulations of experimental GC algorithms. Allocation time is fairly coarse, however, and a single time step can cover a large region of the code spanning multiple method calls. Answering questions like "What caused this object to die?" is not possible.

Elephant Tracks measures time in terms of the number of method entries and exits executed (which we call *method time*). To get a sense of the difference in precision, consider that across the DaCapo benchmarks there are, on average, 70 method entry/exit events between any two allocations (we present more measurements in Section 5). Method time is almost a strict superset of allocation time, since every allocation of a scalar object also calls at least one constructor. The exception is array allocations, but in our experience these are not frequent enough to change the results significantly. Also, if a constructor receives as an *argument* (not the receiver) a new object, there can be two allocations without an intervening constructor call. Again, this is not common.

Bytecode time

The ideal notion of time is probably something like "bytecodes executed", since a single bytecode is the finest grain event that is still VM-independent. A reasonable alternative might be to tick the clock at both method call/return *and* object allocations. As we discuss further in Section 4, ticking the clock more frequently necessitates more object timestamping operations, and thus increases overhead (and can risk expanding methods such that they exceed the maximum allowed method size of 65,535 bytes).

While it may seem that the heap read/write counts records that we make available as an option allow one to use "bytecodes executed" as a measure of time, the counts records do not "tick" the clock, in part for the reasons just mentioned. Further, using them as the clock may make the idealized model too "tight," allowing little re-ordering of the kind typically done by optimizing compilers.

3.2.3 Concurrency

Most modern software uses concurrency in some form, which raises the question of how to order trace events that occur in different threads. We adopt a straightforward approach in which time is global, but trace records include both the time[3] and the ID of the thread in which the event occurred. In the current implementation, however, timestamps on objects do not include the thread ID, so object deaths cannot necessarily be assigned to particular threads.

One problem with this approach is that the resulting traces encode the scheduling decisions of the VM and operating system. Furthermore, trace instrumentation perturbs program execution significantly, resulting in schedules that could be quite different from the uninstrumented programs. While interesting, this problem is difficult to address without controlling the scheduler directly—for example, by replaying a schedule from a real run. One potential solution is to represent time using vector clocks, which would encode only the necessary timing dependences between threads. However, this would still suffer from particularity of orders of interactions. We hope to investigate alternative designs in the future.

4. Implementation

Elephant Tracks is implemented as a Java *agent* that uses the Java Virtual Machine Tool Interface (JVMTI). The primary components of a system using ET are: the JVM itself, including its JVMTI and JNI support; the application; the Elephant Tracks agent; the `ElephantTracks` Java class file, which connects bytecode instrumentation to the agent via Java Native Interface (JNI) calls; and the instrumenter, which rewrites the bytecode of classes as they are loaded.

4.1 Timestamping strategy

For Merlin to produce precise death times, the timestamp on an object must always be the time at which the object last lost an in-

[3] We do not actually output the time value, but it can be derived by knowing which events "tick" the clock.

coming reference. This invariant is easy to maintain for heap and static references, since we can directly instrument these operations, timestamping the old target before allowing the store to proceed. For stack references, however, there is no explicit operation denoting the end of a variable's scope. There are essentially two strategies for solving this problem: (1) timestamp all live variables at every time step, or (2) timestamp each variable exactly when its lifetime ends. (Recall that we define a *variable* as being live only up to its last actual use.)

GCTrace uses strategy (1), which has the advantage that it is straightforward to implement: at each tick of the clock, walk the stack and timestamp each object referred to by a live variable. This strategy, however, creates a trade-off between performance and precision. Walking the stack is a starkly expensive operation, so it cannot be performed frequently, limiting the granularity of the clock. The problem is particularly acute when using allocation time, since a single time step can span multiple methods, requiring a full walk of the call stack at every tick. We believe that stack walking also inhibits code optimizations (or forces de-optimization), further slowing execution. Furthermore, as mentioned in Section 3 it relies on the VM's GCMaps to define variable liveness.

Elephant Tracks uses strategy (2). This approach requires more instrumentation to timestamp a variable's referent whenever the variable is used. It has several advantages, though. The most important is that it works correctly for any granularity of time. In addition, it gives the trace generator explicit control over the model of variable liveness. Finally, it is amenable to an instrumentation-time optimization (described below) that eliminates redundant timestamping operations.

4.2 The instrumenter

The instrumenter is ordinary Java code and is written using the ASM bytecode rewriting tool [3]. The current version of ET is written to use ASM 3.3.1. In order to avoid possible tangle between instrumenter code and the application, we run the instrumenter in a separate operating system process, connected with the agent via pipes in both directions. The agent uses the JVMTI `ClassFileLoadHook` callback, which causes the JVM to present to the agent each class that the JVM wants to load, and to give the agent the opportunity to substitute other bytecode for what the JVM presents. The ET agent sends the bytecode to the instrumenter, which sends back an instrumented class file.

The instrumenter assigns a unique number to each class, each field, each method, and each allocation site (for both scalars and arrays) in each method, writing them to what we call the *names* file. The instrumenter also sends the class and field information to the agent. (At present the agent has no need to maintain tables for the other information, so it is not sent.)

4.2.1 Ordinary instrumentation

We defer to Section 4.2.2 some special cases, and describe now the usual instrumentation added by the ET instrumenter. We organize the description by feature.

Method entry and exit: On entry, and just before a return, we insert a call noting the id of the method and the receiver (for instance methods). In a constructor we cannot actually pass the receiver (it's not initialized yet), so we pass `null` and the agent uses a JNI `GetLocalObject` call to retrieve the receiver from the stack frame.

Exception throw: At an `athrow` bytecode we insert a call that passes the exception object, the method id, and the receiver (for instance methods). The same special handling of the receiver in constructors happens here, too.

Exception exit: To detect exceptional exit of a method, we wrap each method's original bytecode with a catch-anything exception handler, which makes a call indicating the same information as for a throw, and then re-throws the exception.

Exception handle: At the start of each exception handler we insert a call that notes the same information as for a throw.

Scalar object allocations: The basic idea is to insert, after the `new` bytecode, a call that indicates the new object, its class, and the allocation site number. However, we cannot pass the new object directly since it is uninitialized. Further, it is on the JVM stack, not in a local variable, so the JNI `GetLocalObject` function will not work. Our solution is to add one extra local variable to any method that allocates a scalar. We `dup` the new object reference and `astore` it to the extra local. In the call to the agent we indicate which local variable the agent should examine to obtain the object reference. Strictly speaking, we do not need to pass the class, since the agent can figure it out; we may remove that in the future.

Array allocations: New arrays start life fully initialized, so we simply pass them in a call to the agent, along with the allocation site number. For `multianewarray` we call an out-of-line instrumentation routine that informs the agent of each of the whole collection of new arrays that are created. This could also be done in the agent, if desired.

Pointer updates: For `putfield` of a reference and for `aastore` we insert, before the bytecode, a call that notes the object being changed, the object reference being stored, and the field (or index, for an array) being updated. Java allows `putfield` on uninitialized objects (mostly so that an instance of an inner class can have its pointer to its "containing" outer class instance installed, *before* invoking the inner class constructor). In that case we use the same technique as for scalar allocations to indicate to the agent the object being updated.

Uses of objects: As mentioned in Section 3, ET timestamps objects when they are used. We mentioned there the cases in which that happens. We simply insert a call, passing the object to be timestamped. On method entry we timestamp all pointer arguments, including the receiver. (In constructors we make a slightly different call since we cannot pass the receiver; the agent fetches it out of the frame.) For efficiency on method entry, we have timestamp calls that take 2, 3, 4, or 5 objects to stamp.

Counts: As an extension controlled by a command line flag, the instrumenter will also track the number of heap read and writes, the number of heap reads and writes of reference values, and the number of bytecodes executed, and insert calls reporting these just before each action that advances the timestamp clock, and just before control flow branch and merge points.

We further include a simple kind of optimization to reduce the number of timestamp calls. We track which variables (locals and stack) have been timestamped since the last tick or the last bytecode frame object. (Frames occur at control flow merge points, and detail the types of the local and stack variables at that point.) We avoid timestamping an object twice in the same tick. This optimization requires tracking object references as bytecodes move them around, but is straightforward. The optimization is effective, and we found it necessary in order to avoid having some methods increase in size so much, because of added instrumentation, that they overflow the maximum allowed method size.

4.2.2 Instrumentation special cases

We now detail various special cases (beyond access to uninitialized new objects, mentioned in the previous section).

Native methods: In order to indicated when a native method is called and returns, we change its name, prepending `$$ET$$`, and insert a non-native method that calls the native method. We instrument the non-native method essentially as usual. A number of native methods require special treatment, however:

`getStackClass`**:** This method of `java.lang.Class`, and several similar methods, include an argument specifying the number of stack frames to go up to look for some information. To wrap these native methods, the ET non-native wrapper adds one to the number of frames before calling the native. This properly adjusts for the extra level of call that the wrapper adds.

`getClassContext`**:** This method of the IBM J9 `ClassLoader` probes a specific number of frames up the stack, so wrapping it disturbs the result. With regret, we do not wrap it. (We contend that native methods subject to this problem should be redesigned, like `getStackClass` described above, so that they can be wrapped.) A number of other methods exhibit essentially the same problem.

Several native methods of class `Object`**:** Specifically, `getClass`, `notify`, `notifyAll`, and `wait` do not operate correctly if wrapped, so we omit them.

`initReferenceImpl`**:** This method of class `Reference` initializes the `referent` field of a weak reference object. We instrument it specially so that the agent can observe the update to the field, which otherwise would be hidden to ET.

Several methods of `sun.misc.Unsafe`**:** for `allocateInstance` we note the allocation; for a successful `compareAndSwapObject` we note the pointer update, as we do for `putObject`, `putObjectVolatile`, and `putObjectOrdered`. All of these updating operations work in terms of the offset of a field or array element into the object, a fact not readily available to the agent. Therefore we instrument `objectFieldOffset`, `staticFieldBase`, `staticFieldOffset`, `arrayBaseOffset`, and `arrayBaseScale` to inform the agent of the base or offset information they return, so that the agent can map the offsets and bases back to fields and array elements.

`System.arraycopy`**:** We instrument this specially so that the agent can note all the resulting updates to arrays of objects. The agent does the actual work *and* notes the effects, taking care to deal correctly with situations that will throw an exception, etc.

Class `Object`**:** We instrument `Object.<init>` to report the newly initialized object. Sometimes this is the first time we see an object, e.g., for some objects created via JNI calls. We carefully *avoid* instrumenting `Object.finalize` since having any bytecode in that method will cause every object to be scheduled for finalization (which breaks JVMs). Any `finalize` method in another class *is* instrumented, so finalizations are visible in the trace.

Timestamping new objects: Trying to obtain a reference to and timestamp a new object in `Object.<init>` or `Thread.<init>` fails, but the object will be reported soon anyway, so skipping the timestamp operation is not harmful.

4.3 The agent

The agent performs these functions to support ET's goals:

- Sends classes to the instrumenter and returns instrumented classes to the JVM.
- Notes several actions of the JVM and responds appropriately. These include: changes in the JVMTI phase of execution (`VMStart`, `VMInit`, and `VMDeath`); `GarbageCollectionFinish`, which triggers a scan (described further below) to see if any weak references have been cleared; and `VMObjectAlloc`, to detect objects allocated directly by the VM.
- Intercepts various JNI calls so that it can emit suitable trace records, specifically, `AllocObject`, `ThrowNew`, and the various `NewObject` and `NewString` calls, to note the new object; and `SetObjectField`, `SetStaticObjectField`, and `SetObjectArrayElement` to note reference updates.
- Handles the various instrumentation calls from the `ElephantTracks` class and (generally) creates a trace record, inserting it into a buffer.
- Maintains a model of the heap graph. Each node represents an object and each directed edge a pointer. The model also includes static variables, but does not (cannot) include various VM internal roots, and as previously described, we do not model stack roots directly, but employ timestamping to determine liveness from thread stacks.
- To help maintain the heap graph model, and to identify objects in trace records, the agent uses the JVMTI object tagging facility to associate a unique serial number with each object, as early as possible after the object is created.
- Maintains a table of object liveness timestamps, and the timestamp "tick" clock.
- Maintains a data structure describing weak objects and their referents. Whenever the JVM runs its garbage collector, after collection completes the agent notifies a separate agent thread to check each weak object to see if its referent has been cleared. This thread will timestamp the now-unreachable referent with the current time, giving a good-faith estimate as to when it died.

Trace outputting proceeds in cycles. This is because determining which objects have died, propagating timestamps, and inserting death records at the right place in the trace, is a periodic activity, done in batches. When the agent is notified that the JVM is entering the JVMTI Live phase, the agent iterates over the initial heap and creates an object allocation record for each object and a pointer update record for each non-null instance and static field. When the agent is notified that the JVM is entering the JVMTI Dead phase (JVM shutdown), it closes out the current buffer of trace records.

In between, during the Live phase, whenever the trace buffer fills with records, the agent:

1. Forces a garbage collection and then iterates over the remaining heap. This allows the agent to detect which objects have been reclaimed since the trace buffer was last emptied.
2. Applies the Merlin algorithm to compute object death times (really "last time alive" times).
3. Checks weak objects to see if their referent as been cleared. The VM does not inform the agent directly about this, but since we note referent field initialization, we know about weak objects and their referent targets. The tables the agent maintains for these are carefully designed not to keep the objects live (we use JNI weak references).
4. Adds death records to the trace buffer, properly timestamped.
5. Sorts the records using a stable sort, and outputs them.

The last step, sorting and outputting, we observed to consume about half the time of creating a trace and so we developed a parallel version. We report performance results in Section 5.

4.4 Properties of our implementation approach

Our implementation strategy has many advantages and few drawbacks. Its advantages include:

- It works with commercial JVMs (in principle with any JVM supporting JVMTI) and can run any application. Of course timing-dependent applications may misbehave as with any tool that slows execution, etc. This prevents several DaCapo benchmarks from completing successfully.

- The run-time is implemented in C++, with all of its data structures outside of the JVM. This makes it easier to insure that ET data structures and actions are not inappropriately entwined with the application and JVM.
- The instrumenter is in a separate process, insuring it does not become tangled with the application and allowing it to run on a different model JVM, if that is convenient. Our reliance on ASM is not problematic because ASM is widely used, actively maintained, and part of the infrastructure of at least one major commercial JVM (Oracle's HotSpot).
- We capture even some very tricky cases, including weak references, field updates via `sun.misc.Unsafe`, reflective object creation, updates, and method calls, VM internal allocations, relevant JNI calls made by the VM or other native libraries, and `System.arraycopy`.

Drawbacks of ET as it stands are mostly ones that similar tools are likely to share:

- A few methods cannot be instrumented, since doing so breaks the JVM.
- Relative timing and thread interactions are affected, which may change application behavior.
- Weak reference clearing is dependent on the vagaries of the JVM.
- Precision in determining object deaths, and the general wealth of information in the traces, come at a cost: the execution dilation factor is on the order of hundreds (see Section 5 for performance results).
- The resulting system is not as simple as we would like. There are places with somewhat tricky synchronization and more data structures and mappings than we would like, but it is not easy to deal with features such as weak references and `sun.misc.Unsafe`.
- We rely heavily on correctness and completeness of JVMTI and JNI support. One implication is that, at present, JikesRVM cannot support ET. Also, we have discovered previously unreported JVM bugs, such as failure of one JVM to present for rewriting *every* class it loads, which implies that a handful of classes go uninstrumented. (That bug is being fixed, but it appears we later found a similar case whose fix will take longer.)

4.5 Some future directions

We have in mind a few things to work on in the future. One is to devote additional effort to streamlining common cases to improve ET's performance further. For example, at object allocations we do not need to pass the name of the class of the object in the instrumentation call at all, and we would save effort (and bytes in the trace) if we output the instrumenter's number for the class rather than the class name. Perhaps of more significance, we want to tune the data structures that are accessed concurrently, and in particular the locking protocols. We wish to explore different models of local variable and stack liveness, approaching more closely what bytecode interpreters and JIT compilers are likely to do. The time required to rewrite bytecodes in the instrumenter is generally quite small compared with ET's overall running time, so performing data flow analyses, etc., will not itself create bottlenecks. A semantic extension we want to explore is moving ET from being a *GC tracing* tool to being (also) a *memory tracing* tool, outputting traces that include all heap accesses. This might be useful for modeling cache and memory system behavior (neglecting the stack). It might be tricky to accomplish this without causing method size to increase so much that methods exceed the JVM specification limit of 64K bytes.

5. Results

5.1 Performance

In this section we present results from running Elephant Tracks on the DaCapo Benchmarks in order to give a sense of its performance and the properties of the resulting traces. (Unfortunately, it fails to run `tradebeans` and `tradesoap`, perhaps because of internal timeouts.) In Table 1 we present the run-time overhead of our tool under several configurations:

In the No Callback configuration, all of our bytecode instrumentation was injected, but callbacks into the JVMTI agent were disabled (resulting in an empty trace). Additionally, the No Callback configuration enables only the absolute minimum number of JVMTI features necessary to instrument classes. This represents a practical lower bound on the overhead of instrumenting class files and executing the instrumented bytecode, without the overhead of calling into the JVMTI agent, processing the events, or producing a trace record.

The Serial ET configuration periodically pauses to generate death records, put them in order, and output them to the trace file. In contrast, the Parallel ET configuration spawns a separate thread to do this work. This generally results in better performance and fewer pauses in the traced application, but may be of no benefit if the machine lacks sufficient resources to execute this thread in parallel with the application.

With a geometric mean of about 250, the overall dilation factor of Elephant Tracks is within a factor of two of the published dilation factors of GCTrace [6, 7], while providing much more information.

The dilation factors of the different benchmarks are not uniform. This diversity cannot be explained only by the differences in amount of instrumentation, since there is no simple linear relationship between the No Callback configuration and the other configurations. Similarly, it could not be explained by a simple linear model relating number of calls into the JVMTI agent and/or average heap size of the benchmark being traced (at least, no model we were able to discover). Therefore, we theorize that it relates to complex interactions between our instrumentation, Java optimizations, and/or the implementation details of the JVMTI interface.

5.2 Trace analysis

Table 2 shows the composition of the traces by event type (percentage of the trace accounted for by each type). Method entry and exit events outnumber the others significantly, which is why method time is so much more precise than allocation time. In fact, on average there are 70 method entry/exit events between any two allocations. In other words, a single tick of the allocation clock can span dozens of methods, making it difficult to localize object death events within the code. A single tick of the method time clock occasionally contains an allocation, and depending on where the starting and ending method events are found, we might not be able to tell if a death event occurred before or after the allocation. In a few very rare cases, a single unit of method time might contain multiple allocations, but they would have to be arrays, since regular objects always have a constructor call.

In order to demonstrate the value of these more precise traces, we present a few simple trace analysis examples. First, a simple escape analysis is easy to perform with ET traces. We process a trace, and upon encountering a record of object allocation, note the context in which it occurred. Then, if the death event for that same object is encountered before the associated method return, we know the object has not escaped. Conversely, if we do not find the death record before this point, the object has escaped. Note that

Benchmark	No Callback Dilation	Serial ET Dilation	Parallel ET Dilation
avrora-default	1.6	436.5	291.1
avrora-large	0.9	553.9	436.1
avrora-small	1.7	354.0	227.3
batik-default	3.7	152.5	102.6
batik-large	3.0	124.6	84.1
batik-small	2.9	58.3	41.9
eclipse-default	18.0	310.5	2110.5
eclipse-large	19.3	498.6	1603.6
eclipse-small	50.5	47.6	4297.5
fop-default	2.6	181.2	130.2
fop-small	2.8	42.0	30.7
h2-default	4.4	3137.3	3245.8
h2-large	4.3	2652.7	3583.1
h2-small	3.1	1272.9	1038.7
h2-tiny	3.9	947.5	754.7
jython-default	2.0	342.2	235.0
jython-large	2.5	949.4	774.9
jython-small	1.6	93.1	71.5
luindex-default	1.7	88.4	71.6
luindex-small	1.6	5.8	4.4
lusearch-default	2.7	385.7	304.3
lusearch-large	2.8	451.8	327.9
lusearch-small	2.9	112.5	85.7
pmd-default	2.0	276.1	134.6
pmd-large	2.4	549.1	230.2
pmd-small	1.8	7.4	5.6
sunflow-default	5.9	1830.1	1457.8
sunflow-large	6.4	2073.2	1583.3
sunflow-small	6.9	598.1	481.2
tomcat-default	1.8	100.3	72.3
tomcat-large	1.7	240.0	175.4
tomcat-small	1.8	48.4	38.4
xalan-default	3.0	482.0	372.5
xalan-large	3.7	922.6	715.4
xalan-small	2.8	114.8	95.5
geomean	3.2	257.7	245.8

Table 1. Run-time overhead for Elephant Tracks on the DaCapo benchmark suite

Benchmark	Method	Alloc + Death	Catch + Throw	Pointer Update
avrora-default	97.97	0.30	0.00	1.74
avrora-large	95.17	0.24	0.00	4.59
avrora-small	97.85	0.30	0.00	1.85
batik-default	92.01	1.70	0.00	6.29
batik-large	92.34	1.77	0.00	5.89
batik-small	92.55	2.55	0.00	4.89
eclipse-default	88.65	4.29	0.00	7.06
eclipse-large	90.24	3.23	0.00	6.52
eclipse-small	93.67	3.27	0.01	3.05
fop-default	90.30	4.63	0.00	5.07
fop-small	89.83	4.21	0.00	5.95
h2-default	94.23	2.86	0.00	2.90
h2-large	94.92	2.02	0.00	3.06
h2-small	94.05	3.04	0.00	2.91
h2-tiny	93.98	3.11	0.00	2.91
jython-default	91.64	3.08	0.02	5.26
jython-large	91.74	2.79	0.01	5.46
jython-small	97.36	1.11	0.00	1.53
luindex-default	96.37	0.28	0.00	3.36
luindex-large	90.49	5.61	0.06	3.84
luindex-small	94.91	1.36	0.00	3.72
lusearch-default	91.70	2.73	0.05	5.52
lusearch-large	91.70	2.72	0.05	5.52
lusearch-small	91.77	2.74	0.05	5.43
pmd-default	87.31	3.86	0.10	8.73
pmd-large	87.04	4.27	0.09	8.60
pmd-small	92.54	3.48	0.01	3.97
sunflow-default	94.84	3.00	0.00	2.16
sunflow-large	94.83	3.00	0.00	2.17
sunflow-small	94.90	2.96	0.00	2.14
tomcat-default	89.92	5.92	0.02	4.14
tomcat-large	90.47	6.33	0.01	3.19
tomcat-small	89.00	5.28	0.03	5.70
xalan-default	94.10	1.54	0.00	4.37
xalan-large	94.11	1.52	0.00	4.37
xalan-small	93.99	1.65	0.00	4.36
mean	92.74	2.85	0.01	4.40

Table 2. Percentage of each record type in traces of the DaCapo benchmark suite

this does not necessarily mean that there is any static analysis that could have determined in advance that the object would or would not have escaped. The results of this escape analysis are reported in Table 3, where we see that in most benchmarks a majority of objects escape their allocating context.

Second, previous work has shown that the allocation site plus some calling context is a good basis for predicting object lifetime (measured in bytes of allocation) [11]. Since Elephant Tracks' traces can also provide the calling context of an object's death, it is possible to consider whether the allocation context is also a predictor of *death context*.

As a preliminary investigation, we performed the following analysis. Each object's allocation is recorded with a triple, consisting of the allocation site, the allocating method, and the caller of the allocating method (this gives us a partial calling context). Next, the analysis finds the top ten allocation contexts (based on number of objects allocated). For each object allocated in these contexts, it determines the object's partial death context (there is no *site* for a death event, so we consider only the method in which it died, and the calling method). Finally, the analysis finds, for each allocation context, the most common death context for objects with that allocation context. In Table 4 we report the average percentage of objects allocated in the top ten contexts that die in the plurality context.

This initial work suggests that the death context of an object may be a stable and predictable feature. However, additional refinement will be required to further illuminate the relationship between an object's allocation context and its death context, as well as to determine if this relationship can be exploited for any optimization.

6. Conclusions

We have presented Elephant Tracks, a tool for efficiently generating program traces including accurate object death records. Unlike previous tools, Elephant Tracks traces allow recorded events to be placed in the context of the methods of the program being traced. It also works independently of any particular choice of the JVM to which it attaches. These two features will allow the prototyping of new GC algorithms and new kinds of program analysis, and let the tool keep up with changes in the JVM and class libraries. Elephant Tracks offers performance comparable to similar previous tools but a wealth more information and covers many more of the tricky and corner cases of Java and Java virtual machines.

Benchmark	% Escaping	Benchmark	% Escaping
avrora-default	83.53	luindex-default	54.14
avrora-large	79.39	luindex-small	46.25
avrora-small	87.41	lusearch-default	39.98
batik-default	63.79	lusearch-large	40.00
batik-large	62.97	lusearch-small	40.02
batik-small	62.22	pmd-default	53.68
eclipse-default	32.32	pmd-large	52.66
eclipse-large	41.97	pmd-small	51.78
eclipse-small	26.85	sunflow-default	68.63
fop-default	55.25	sunflow-large	68.49
fop-small	65.06	sunflow-small	68.38
h2-default	58.03	tomcat-default	25.44
h2-large	58.24	tomcat-large	21.87
h2-small	58.00	tomcat-small	32.44
h2-tiny	57.82	xalan-default	54.99
jython-default	42.13	xalan-large	55.16
jython-large	42.95	xalan-small	53.59
jython-small	68.02		

Table 3. Percentage of objects escaping their allocating context in the DaCapo benchmark suite

Benchmark	Mean %	Benchmark	Mean %
avrora-default	41.22	luindex-small	76.57
avrora-large	44.38	lusearch-default	83.39
avrora-small	30.06	lusearch-large	79.63
batik-default	63.05	lusearch-small	83.37
batik-large	59.64	pmd-default	47.82
batik-small	75.08	pmd-large	34.98
fop-default	81.24	pmd-small	57.70
fop-small	64.54	sunflow-default	86.12
h2-default	86.37	sunflow-large	80.12
h2-large	88.09	sunflow-small	89.54
h2-small	86.78	tomcat-default	75.80
jython-default	68.79	tomcat-large	72.15
jython-large	74.03	tomcat-small	79.32
jython-small	74.15	xalan-default	71.48
luindex-default	78.45	xalan-large	71.55

Table 4. Mean percentage of objects that are born in the same context and die in the same context (over top 10 allocation contexts).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-1018038 (Guyer and Ricci) and CNS-1162246 (Moss).

References

[1] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *PLDI*, pages 269–279, 1998.

[2] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2): 399–418, 2005.

[3] Eric Bruneton, Romain Langlet, and Thierry Coupaye. ASM: a code manipulation too to implement adaptable systems. In *Adaptable and Extensible Component Systems*, page 12 pages, Grenoble, France, November 2002.

[4] James Foucar. *A Platform for Research into Object-Level Trace Generation*. PhD thesis, The University of New Mexico, 2006.

[5] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-Me: A static analysis for automatic individual object reclamation. *ACM SIGPLAN Notices*, 41(6):364–375, 2006.

[6] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. Error-free garbage collection traces: How to cheat and not get caught. *SIGMETRICS Perform. Eval. Rev.*, 30:140–151, June 2002. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/511399.511352. URL `http://doi.acm.org/10.1145/511399.511352`.

[7] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems*, 28(3):476–516, 2006.

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

[9] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–373, 2003. ISBN 1-58113-712-5.

[10] Hajime Inoue, Darko Stefanović, and Stephanie Forrest. On the prediction of Java object lifetimes. *IEEE Transactions on Computers*, 55(7):880–892, 2006.

[11] Richard E. Jones and Chris Ryder. A study of Java object demographics. In *Proceedings of the 7th International Symposium on Memory Management*, pages 121–130. ACM, 2008.

[12] Jonathan M. Lambert and James F. Power. Platform independent timing of Java virtual machine bytecode instructions. *Electronic Notes in Theoretical Computer Science*, 220(3):97–113, 2008.

[13] Sun Microsystems. JVM Tool Interface, 2004. http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html.

[14] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2): 128–170, 1997.