The Dissertation Committee for Michael David Bond
certifies that this is the approved version of the following dissertation:

Diagnosing and Tolerating Bugs in Deployed Systems

Committee:

Kathryn S. McKinley, Supervisor

Stephen M. Blackburn

Keshav Pingali

Peter Stone

Emmett Witchel

Diagnosing and Tolerating Bugs in Deployed Systems

by

Michael David Bond, B.S., M.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2008

Acknowledgments

I am deeply grateful to Kathryn McKinley for supporting and mentoring me in many ways. Kathryn has provided invaluable technical expertise and feedback even as we worked in new areas. She has been unwaveringly enthusiastic and positive about students and ideas and projects, and she has been a strong advocate, putting my and other students' interests first. I entered graduate school uncertain about getting a Ph.D., and I leave inspired to pursue an academic career because of Kathryn.

Steve Blackburn has been an enthusiastic supporter and has devoted a lot of his time to solving problems and giving advice, both solicited and unsolicited. Steve, Keshav Pingali, Peter Stone, and Emmett Witchel have given helpful feedback and spent a lot of time reading long documents and attending long talks. Vitaly Shmatikov and Steve Keckler have provided valuable advice and mentoring. David Padua and Craig Zilles were supportive advisors when I was a master's student at the University of Ilinois.

The graduate student community at UT has been incredibly supportive. I am thankful to have Ben Wiedermann, Xianglong Huang, and Milind Kulkarni as friends and colleagues. I am indebted to them and to the following friends and colleagues for their technical and personal support: Kartik Agaram, Katherine Coons, Boris Grot, Sam Guyer, Jungwoo Ha, Maria Jump, Byeongcheol Lee, Bert Maher, Kristi Morton, Nick Nethercote, Phoebe Nethercote, Alison Norman, Dimitris Prountzos, Jennifer Sartor, Simha Sethumadhavan, and Suriya Subramaniam. In my cohort but outside my area, Jason

Davis, Justin Brickell, and Matt Taylor have been good friends and colleagues without the good sense to be systems researchers.

This thesis would have not have been possible without Jikes RVM. I am indebted to all Jikes RVM developers and especially Steve Blackburn, Daniel Frampton, Robin Garner, David Grove, and Ian Rogers for their generous efforts maintaining and improving Jikes RVM.

I am thankful for guidance and help provided by Gem Naviar, Tom Horn, Aubrey Hooser, and Gloria Ramirez. Gloria Ramirez and Bill Mark helped solve a two-body problem that would have prevented me from attending UT. Intel funded my research with an internship and a fellowship, and Suresh Srinivas has supported and mentored me.

I am thankful for the unconditional love and support that my parents have provided throughout my life.

My partner Julia has selflessly supported and tolerated me while also being awesome and fun, and I am grateful.

Diagnosing and Tolerating Bugs in Deployed Systems

Publication No. ______________

Michael David Bond, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Kathryn S. McKinley

Deployed software is never free of bugs. These bugs cause software to fail, wasting billions of dollars and sometimes causing injury or death. Bugs are pervasive in modern software, which is increasingly complex due to demand for features, extensibility, and integration of components. Complete validation and exhaustive testing are infeasible for substantial software systems, and therefore deployed software exhibits untested and unanalyzed behaviors.

Software behaves differently after deployment due to different environments and inputs, so developers cannot find and fix all bugs before deploying software, and they cannot easily reproduce post-deployment bugs outside of the deployed setting. This dissertation argues that *post-deployment is a compelling environment* for diagnosing and tolerating bugs, and it introduces a general approach called *post-deployment debugging.* Techniques in this class are efficient enough to go unnoticed by users and accurate enough to find and report the sources of errors to developers. We demonstrate that they help developers find and fix bugs and help users get more functionality out of failing software.

To diagnose post-deployment failures, programmers need to understand the program operations—control and data flow—responsible for failures. Prior approaches for widespread tracking of control and data flow often slow programs by two times or more and increase memory usage significantly, making them impractical for online use. We present novel techniques for representing control and data flow that add modest overhead while still providing diagnostic information directly useful for fixing bugs. The first technique, *probabilistic calling context* (PCC), provides low-overhead context sensitivity to dynamic analyses that detect new or anomalous deployed behavior. Second, *Bell* statistically correlates control flow with data, and it reconstructs program locations associated with data. We apply Bell to leak detection, where it tracks and reports program locations responsible for real memory leaks. The third technique, *origin tracking*, tracks the originating program locations of *unusable values* such as null references, by storing origins *in place* of unusable values. These origins are cheap to track and are directly useful for diagnosing real-world null pointer exceptions.

Post-deployment *diagnosis* helps developers find and fix bugs, but in the meantime, users need help with failing software. We present techniques that *tolerate memory leaks*, which are particularly difficult to diagnose since they have no immediate symptoms and may take days or longer to materialize. Our techniques effectively narrow the gap between reachability and liveness by providing the illusion that dead but reachable objects do not consume resources. The techniques identify *stale* objects not used in a while and remove them from the application and garbage collector's working set. The first technique, *Melt*, relocates stale memory to disk, so it can restore objects if the program uses them later. Growing leaks exhaust the disk eventually, and

some embedded systems have no disk. Our second technique, *leak pruning*, addresses these limitations by automatically reclaiming likely leaked memory. It preserves semantics by waiting until heap exhaustion to reclaim memory—then intercepting program attempts to access reclaimed memory.

We demonstrate the utility and efficiency of post-deployment debugging on large, real-world programs—where they pinpoint bug causes and improve software availability. Post-deployment debugging efficiently exposes and exploits programming language semantics and opens up a promising direction for improving software robustness.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Bugs in Deployed Software Systems

Software complexity. Despite a lot of effort to make deployed software robust and error free, it still contains bugs that cause systems to fail. These failures waste billions of dollars [Nat02, Sco98] and sometimes result in injury or death. Society is relying more and more on software, and according to two recent NRA reports [JTM07, WM07], public health, personal safety, and the economy depend on systems being highly robust, available, and secure.

Bugs are inherent in modern software because software behavior is difficult to reason about. This uncertainty is due in large part to software being increasingly connected, complex, and extensible. These characteristics are driven by the need for software to do more for humans. For example, software provided 8% of the F-4 fighter's capability in 1960. In 2000, software provided 85% of the F-22 fighter's capability. Large programs have tens of millions of lines of code, and if current trends continue, they are likely to have one billion lines of code in 10 years [WM07].

Software complexity makes it hard to write correct code since developers' understanding of their programs does not scale well. Complexity also makes it hard to find and fix bugs before deployment because it is infeasible to test or statically analyze all possible program behaviors prior to deployment. Program behavior differs in the deployed setting because it executes

with different *inputs* and *environments*, which are too numerous to evaluate exhaustively in complex software.

Debugging prior to deployment. Developers avoid most bugs during development or fix them during testing, which is cost effective [WM07], but their efforts fail to eliminate all bugs. Language support can eliminate many bugs, but most solutions are not practical. Functional languages eliminate whole classes of bugs but do not perform well for many applications. Annotations for high-performance languages require too much programmer effort and still rely on the programmer for correctness. *Managed languages* such as Java, C#, Python, and Ruby are increasingly popular [FL05, TIO07] because they improve programmer productivity by reducing programmer effort and eliminating a whole class of bugs. These languages provide garbage collection and type safety, which prevent many memory bugs (errors in heap management) such as (1) memory corruption due to dangling pointers, repeat frees, and bogus writes and (2) memory leaks due to unreachable objects. Unfortunately these languages still suffer from (1) memory leaks due to reachable but dead objects, (2) failures (rather than silent corruption) due to null pointer exceptions, array-out-of-bounds exceptions, and invalid type casts, (3) concurrency bugs (bugs dependent on specific interleavings of operations of concurrent execution contexts), and (4) semantic bugs (any other bug where the program does not implement what the programmer intended).

Typical industrial approaches for preventing bugs include good programming practices such as careful designs, coding disciplines, pair programming, and code reviews. These practices are critical to obtaining software's current level of robustness, but they do not eliminate all bugs because they are subject to human fallibility. To find and fix bugs, industry uses testing and, to a lesser extent, static analysis. Static analysis has the advantage that it finds bugs in *all* possible program executions, but to avoid path explosion, practical tools conservatively abstract and summarize program state, resulting in false positives. Testing uses dynamic analysis to find real errors in *some* program executions intended to approximate deployed program behavior. Testing fails to find some bugs that occur after deployment because the actual deployed inputs and environments lead to different program behavior than occurred at testing time.

In summary, real-world deployed software is never bug free. Increasing complexity makes it infeasible to understand, analyze, and test programs to eliminate all bugs early. At the same time, we need more reliable systems because society is relying more and more on software.

1.2 Improving Reliability in the Deployed Setting

Deployed software contains bugs that are generally unknown and cause failures unexpectedly. Developers need help finding and fixing these bugs, which are often hard to reproduce outside of the deployed environment. Users experiencing failures need immediate, automatic help to keep software running usefully. These approaches must be orders of magnitude more efficient than testing-time techniques, or users will not accept them.

This dissertation argues that *the deployed setting is a necessary and ideal environment* for dealing with bugs because these bugs are most problematic for users and developers. We present three low-overhead techniques that help programmers *diagnose* bugs in deployment by identifying program behavior responsible for bugs. While the deployed setting observes bugs that cause

failures, it may be too late for many applications that require high availability such as mission-critical and autonomous systems. In the meantime, users need immediate solutions to avoid downtime. We show automatic *tolerance* is a compelling direction with two techniques that tolerate memory leaks in deployed systems.

1.2.1 Diagnosing Bugs by Tracking Flow

To find and fix bugs, programmers need to know *why* and *how* failures occur. They ask questions such as "Why did this program state occur?" and "How did this variable get this value?" These questions are fundamentally about how a program changes state using control and data flow. Our techniques enable widespread tracking of control flow, data flow, and the relationship between them in order to report code and data responsible for an error. Prior work has used sampling to keep overhead low but requires many users to experience a bug in order to diagnose it [Lib04]. In contrast, our techniques need just one buggy run to report culprit code and data. This feature makes our techniques applicable to safety-critical software that fails infrequently and software with few running instances (e.g., aircraft carrier software).

Dynamic context sensitivity. Modern, object-oriented programs increasingly use more interprocedural and less intraprocedural control flow, increasing the importance of context sensitivity for analysis. *Calling context* enhances program understanding and dynamic analyses by providing a rich representation of program location. Efficiently computing calling context has been a problem for decades. Prior methods are expensive in time and space, e.g., walking the stack and building a calling context tree slow programs by a factor of two or more.

Probabilistic calling context (PCC) adds dynamic context sensitivity to analyses that detect new and potentially buggy behavior. It continuously maintains a probabilistically unique value representing the current calling context. Modern software typically has millions of unique contexts, yet a 32-bit PCC value generates just a few conflicts. A 64-bit PCC value gives just a few conflicts for billions of unique contexts, which are likely in future systems. Computing the PCC value adds overheads low enough for production use. PCC is well suited to clients that detect new or anomalous behavior since PCC values from training and production runs can be compared easily to detect new context-sensitive behavior. PCC is efficient and accurate enough for residual testing [PY99, VNC07], bug detection, and intrusion detection [FKF+03, IF02, WS02].

Statistical code-data correlation. Online bug diagnosis tools benefit from storing per-object *sites* (e.g., allocation or last-access sites) and reporting the sites for potentially bug-related objects. In modern programs with many small objects, per-object sites add high space overhead, limiting their viability in deployment.

We introduce *Bell*, a statistical technique that *encodes* per-object sites in a single bit per object. A bit loses information about a site, but given sufficient objects that use the site and a known, finite set of possible sites, brute-force *decoding* recovers the site with high accuracy. We use Bell in a leak detector to identify the sites that allocated and last used objects identified as likely leaks. We show these sites are directly useful for diagnosing leaks in applications such as Eclipse and SPECjbb2000.

Tracking bad values. Programs sometimes crash due to *unusable* values, for example, when Java and C# programs dereference null pointers and when C and C++ programs use undefined values to affect program behavior. A stack trace produced on such a failure identifies the effect of the unusable value, not its cause, and is usually not enough to understand and fix the bug. These crashes can be especially difficult to debug because they contain no useful information for the developer.

We present efficient *origin tracking* of unusable values, which maintains the program location that initially produced each unusable value. The key idea is *value piggybacking*: when the program stores an unusable value, value piggybacking instead stores origin information in the spare bits of the unusable value. Modest compiler support alters the program to propagate these modified values through operations such as assignments and comparisons. This dissertation focuses on an implementation that tracks null pointer origins in a JVM; the technique also finds undefined values in native programs running in Valgrind's Memcheck tool [BNK+07]. Our Java null pointer diagnosis tool adds no noticeable overhead, and we show that the origins it reports are useful for diagnosing real null pointer exceptions in large, open-source software.

1.2.2 Tolerating Memory Leaks

Our diagnosis techniques help developers find and fix bugs, but in the meantime, users need help with failing software. Automatic *tolerance* techniques can respond to failures by hiding, ignoring, or even attempting to fix a bug automatically. This dissertation introduces two techniques that tolerate *memory leaks* while preserving semantics.

Managed languages such as Java, C#, Python, and Ruby shield programmers from most but not all aspects of memory management. These languages eliminate memory corruption due to malloc-free misuse and out-of-bounds writes, and they eliminate memory leaks due to unreachable objects. However, a program still leaks memory if it maintains references to objects it will never use again. Garbage collection cannot collect these dead but reachable objects because it uses *reachability* to over-approximate *liveness*. Computing reachability is straightforward; collectors perform a transitive closure over the object graph from program *roots* (registers, stacks, and globals). Computing liveness is much harder and is in general undecidable.

Memory leaks hurt performance by increasing garbage collection (GC) frequency and workload, and by reducing program locality. Growing leaks eventually exhaust memory and crash the program. Virtual memory paging is not sufficient to handle these effects because (1) pages that mix leaked and non-leaked objects waste physical memory, and (2) GC's working set is all reachable objects, which causes thrashing.[1]

Leaks are hard to reproduce, diagnose, and fix because they have no immediate symptoms [HJ92]. For example, *when* a program exhausts memory depends on the total memory available, the collector, and nondeterministic factors not directly related to the leak. Leaks are difficult to find and fix, and many industrial and research tools try to help programmers diagnose them [JM07, MS03, Orab, Que, Sci].

We introduce two techniques for tolerating *memory leaks*. Our techniques preserve program semantics: they keep leaky programs running by moving unused memory out of the program and garbage collector's working

[1]We focus on tracing collectors. Reference counting also must trace to reclaim dead cycles.

set. They cannot tolerate all leaks indefinitely. They fail if they exhaust disk; if a program leaks live, not dead, objects; or if they inadvertently collect live objects.

Offloading leaks to disk. *Melt* tolerates leaks by moving likely leaked objects to disk. It identifies *stale* objects, which are objects the program has not accessed in a while, and moves them to disk, freeing virtual and physical memory used for them. It preserves semantics by moving disk objects back to memory if the program subsequently accesses them. By limiting application and collector accesses to disk and using two levels of indirection for pointers from disk to memory, Melt guarantees that time and memory usage remain proportional to *in-use* memory rather than leaked memory. Whereas leaky programs grind to a halt and crash, given sufficient disk space, Melt keeps several programs with real leaks running and maintains their performance as the leak grows.

Reclaiming leaked memory. Melt relies on available disk space to hold leaking objects. However, growing leaks will eventually exhaust disk space, and many embedded systems have no disk at all. *Leak pruning* predicts dead objects and reclaims them automatically, based on data structure sizes and usage patterns. It preserves program semantics because it waits until heap exhaustion before reclaiming objects and then *poisons* references to objects it reclaims. If the program later tries to access a poisoned referenced, the VM throws an error. In the worst case, leak pruning defers fatal errors. In the best case, programs with unbounded reachable memory growth execute indefinitely and correctly in bounded memory with consistent performance.

We evaluate Melt and leak pruning on nine long-running leaks in real programs and third-party microbenchmarks and show that both tolerate four leaks for at least 24 hours, help one program run significantly longer, and do not help three programs. Melt and leak pruning perform differently for one leak: Melt tolerates it for at least 24 hours, while leak pruning terminates sooner but allows it to run about 21 times longer than without leak tolerance.

1.3 Meaning and Impact

Our techniques use novel insights into programming languages to be orders of magnitude more efficient than prior approaches. The diagnosis techniques rely on innovative ways to represent data and control flow efficiently. The tolerance techniques effectively narrow the liveness-reachability gap inherent in garbage-collected languages by providing the illusion that dead but reachable objects consume no resources.

This work has immediate practical impact because it can be deployed today to improve software reliability. For example, Azul is considering adding origin tracking in their production VM [Cli08]. Nokia has expressed interest in leak pruning as a way to deal with leaks on cell phones [Her08]. Other researchers have used the Bell and PCC implementations in their work [JR08, TGQ08].

The broader impact of this work is that it demonstrates the viability of debugging and tolerating bugs during deployment. It points to a future where (1) developers use system support for finding bugs in software that users are running and (2) users embrace bug tolerance approaches since it improves their computing experience. Developers will increasingly need help diagnosing software as it becomes more complex and more difficult to reproduce errors

outside the deployment environment. As society's reliance on software grows, users will need more robust and available software, and they will increasingly need automated techniques that keep software running.

Chapter 2

Background

This chapter provides background information for the five diagnosis and tolerance approaches in this dissertation, which we have also published separately [BM06b, BM07, BM09, BM08, BNK$^+$07]. The chapter first describes the base system for our implementations and then presents the common parameters of our experimental methodology.

2.1 Implementations

We implemented the five bug diagnosis and tolerance tools described in this dissertation in Jikes RVM, a high-performance Java-in-Java virtual machine [AAB$^+$00, AFG$^+$00, Jika]. As of August 2008, the DaCapo benchmarks regression tests page shows that Jikes RVM performs the same as Sun Hotspot 1.5 and 15–20% worse than Sun 1.6, JRockit, and J9 1.9, all configured for high performance [DaC]. Our performance measurements are therefore relative to an excellent baseline.

All five implementations are publicly available on the Jikes RVM Research Archive [Jikb]:

- Probabilistic calling context, as a patch against Jikes RVM 2.4.6 (Section 3.3). Jones and Ryder used part of this patch, deterministic calling context profiling, in their object demographics study [JR08].

- Leak detection using Bell, as a patch against Jikes RVM 2.4.2 (Section 4.3). Tang et al. used our leak detector as the basis for their LeakSurvivor tool [TGQ08].
- Origin tracking of null pointer exceptions, as a patch against Jikes RVM 2.4.6 (Section 5.1). We have also made available the 12 null pointer exceptions we used to evaluate the implementation [KB08]. Our co-author Nethercote modified Valgrind's Memcheck tool [NS07, SN05] to track the origins of undefined values in C and C++ programs. That implementation is publicly available as a branch in the Valgrind source repository [BNK$^+$07].
- Melt, as a patch against Jikes RVM 2.9.2 (Section 6.2).
- Leak pruning will be available in January 2009 (Section 7.2).

2.2 Methodology

Jikes RVM uses just-in-time compilation and automatic memory management, and this section describes how we control these features for our experimental evaluation. It also describes other aspects of methodology common to all our experiments.

2.2.1 Compilers

Jikes RVM uses just-in-time compilation to produce machine code for each method at run time. Initially a *baseline* compiler generates machine code when the application first executes a method. The baseline compiler generates machine code directly from bytecode and does not use an internal representation (IR). If a method executes many times and becomes *hot*, the VM recompiles it with an *optimizing* compiler at successively higher optimization levels. This compiler uses an IR and includes standard optimizations such as inlining, constant propagation, and register allocation. The implementations in this dissertation modify both compilers to add instrumentation to generated application code.

2.2.2 Execution

Like many VMs, Jikes RVM by default dynamically identifies frequently-executed methods and recompiles them at higher optimization levels. We refer to experiments using this default execution model as using *adaptive* compilation. Because Jikes RVM uses timer-based sampling to identify hot methods, adaptive compilation is nondeterministic. Run-to-run performance variation is significant because different profiles lead to differently-optimized code, affecting application performance. In addition, the optimizing compiler affects performance since it runs in parallel with the application and allocates objects into the heap.

To eliminate this source of nondeterminism, performance experiments use a deterministic compilation model called *replay* compilation [HBM+04, OOK+06, SMB04]. Replay uses *advice files* produced by a previous well-performing adaptive run (best of five). The advice files specify (1) the highest optimization level reached by each method, (2) a dynamic call graph profile, and (3) an edge profile. Fixing these inputs, replay compilation executes two consecutive iterations of the application. During the first iteration, Jikes RVM optimizes code based on the advice files. The second iteration executes only the application, with a realistic mix of optimized and unoptimized code. We

use the first iteration to measure compilation time and the second iteration to measure application time.

For replay compilation, we typically report the minimum of five trials since it represents the run least perturbed by external effects. For adaptive compilation, we typically execute 25 trials because of high variability and report the median to discount outliers.

2.2.3 Memory Management

Jikes RVM's Memory Management Toolkit (MMTk) [BCM04] supports a variety of garbage collectors with most functionality residing in shared code. The memory management parts of our implementations reside almost entirely in this shared code, so they support a variety of collectors naturally. By default we use a generational mark-sweep collector, which has high performance. It allocates new objects into a *nursery*, then periodically moves surviving objects to a mark-sweep *mature space*. Two implementations run with other collectors: our leak detector that uses Bell, which requires a pure mark-sweep collector; and Melt, which works with many collectors but executes with a generational copying collector in our experiments to demonstrate Melt's support for moving in-use objects.

Heap size affects garbage collection frequency and workload, as well as application locality. To control this source of performance variation, our experiments fix the size of the heap for each benchmark. We typically execute each benchmark in a heap three times the minimum, a medium size that does not typically incur high GC overhead or poor application locality. We determine minimum heap sizes experimentally.

2.2.4 Benchmarks

We evaluate performance using the DaCapo benchmarks (version beta-050224 for Bell, 2006-10 for origin tracking and PCC, and 2006-10-MR1 for Melt and leak pruning), a fixed-workload version of SPECjbb2000 called pseudojbb, and SPECjvm98 [BGH+06, Sta99, Sta01].

2.2.5 Platform

Experiments execute on two Pentium 4 platforms and a Core 2 platform:

- Experiments for PCC, Bell, and origin tracking, execute on a 3.6 GHz Pentium 4 with a 64-byte L1 and L2 cache line size, a 16KB 8-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 2MB unified 8-way set associative L2 on-chip cache, and 2 GB main memory, running Linux 2.6.12.
- Experiments for Melt and leak pruning execute on a *dual-core* 3.2 GHz Pentium 4 system with 2 GB of main memory running Linux 2.6.20.3. Each core has a 64-byte L1 and L2 cache line size, a 16-KB 8-way set associative L1 data cache, a 12-Kμops L1 instruction trace cache, and a 1-MB unified 8-way set associative L2 on-chip cache.
- When evaluating Melt's ability to tolerate leaks, the top four leaks in Table 6.2 and SPECjbb2000 execute on a machine with 126 GB of free disk space. It is a Core 2 Quad 2.4 GHz system with 2 GB of main memory running Linux 2.6.20.3. Each core has a 64-byte L1 and L2 cache line size, an 8-way 32-KB L1 data/instruction cache, and each pair of cores shares a 4-MB 16-way L2 on-chip cache.

Chapter 3

Probabilistic Calling Context

Dynamic calling context is the sequence of active method invocations that lead to a program location. Calling context is powerful because it captures interprocedural behavior and yet is easy for programmers to understand. Previous work in testing [Bin97, CG06a, GKS05, HRS+00, Mye79, RKS05], debugging and error reporting [HRD+07, LYY+05, NS07, SN05], and security [FKF+03, Ino05] demonstrates its utility. For example, programmers frequently examine calling context, in the form of error stack traces, during debugging. Untested behavior such as unexercised calling contexts are called *residuals* [PY99]. Residual calling contexts observed in deployed software are clues to potential bugs. Anomalous sequences of calling contexts at system calls can reveal security vulnerabilities [FKF+03, Ino05].

Computing calling context cheaply is a challenge in non-object-oriented languages such as C, and it is even more challenging in object-oriented languages. Compared with C programs, Java programs generally express more control flow interprocedurally in the call graph, rather than intraprocedurally in the control flow graph. Our results show that Java has more distinct contexts than comparable C programs [ABL97, Spi04]. For example, large C programs such as GCC and 147.vortex have 57,777 and 257,710 distinct calling contexts respectively [ABL97, Spi04], but the remaining six SPEC CPU C programs in Ammons et al.'s workload have fewer than 5,000 contexts. In contrast, we find that 5 of 11 DaCapo Java benchmarks contain more than 1,000,000 distinct calling contexts, and 5 others contain more than 100,000 (Section 3.4).

The simplest method for capturing the current calling context is walking the stack. For example, Valgrind walks the stack at each memory allocation to record its context-sensitive program location, and reports this information in the event of a bug [NS07, SN05]. If the client of calling contexts very rarely needs to know the context, then the high overhead of stack-walking is easily tolerated. An alternative to walking the stack is to build a calling context tree (CCT) dynamically and to track continuously the program's position in the CCT [ABL97, Spi04]. Unfortunately, tracking the program's current position in a CCT adds a factor of 2 to 4 to program runtimes. These overheads are unacceptable for most deployed systems. Recent work samples hot calling contexts to reduce overhead for optimizations [ZSCC06]. However, sampling is not appropriate for testing, debugging, or checking security violations since these applications need coverage of both hot and cold contexts.

This chapter introduces an approach called *probabilistic calling context* (PCC) that continuously maintains a value that represents the current calling context with very low overhead. PCC computes this value by evaluating a function at each call site. To differentiate calling contexts that include the same methods in a different order, we require a function that is non-commutative. To optimize a sequence of inlined method calls into a single operation, we prefer a function whose composition is cheap to compute. We present a function that has these properties. In theory and practice, it produces a unique value for up to millions of contexts with relatively few conflicts (false negatives) using a 32-bit PCC value. When necessary, a 64-bit PCC value can probabilistically

differentiate billions of unique calling contexts.

PCC is well suited to adding context sensitivity to dynamic analyses that detect new or anomalous program behavior such as coverage testing, residual testing, invariant-based bug detection, and anomaly-based intrusion detection. These clients naturally have a *training* phase, which collects program behavior, and a *production* phase, which compares behavior against training behavior. Calling contexts across runs can be compared easily by comparing PCC values: two different PCC values definitely represent different contexts. Although a new PCC value indicates a new context, the context is not determinable from the value, so PCC walks the stack when it encounters anomalous behavior to report the calling context.

We demonstrate that continuously computing a 32-bit PCC value adds on average 3% overhead. Clients add additional overhead to query the PCC value at client-specific program points. We approximate the overhead of querying the PCC value by looking up the value in a hash table on each query. Querying at every call in the application increases execution times by an average of 49% and thus is probably only practical at testing time. In several interesting production scenarios, we demonstrate that querying the PCC value frequently is feasible: querying at every system call adds no measurable overhead, at every `java.util` call adds 3% overhead (6% total); and examining it at every Java API call adds 9% overhead (12% total). Computing the PCC value adds no space overhead, but clients add space overhead proportional to the number of distinct contexts they store (one word per context), which is often millions but still much smaller than statically possible contexts. To our knowledge, PCC is the first approach to achieve low-overhead and always-available calling context.

3.1 Motivation

This section motivates efficient tracking of calling context for improving testing, debugging, and security. Some previous work shows dynamic context sensitivity helps these tasks [Bin97, FKF$^+$03, LYY$^+$05, Ino05]. However, most prior work uses intraprocedural paths or no control-flow sensitivity for these tasks [AH02, HRD$^+$07, HRS$^+$00, IF02, VNC07, WS02] since paths are often good enough for capturing program behavior and calling context was too expensive to compute. Because developers more often now choose object-oriented, managed languages such as Java and C# [TIO07], calling context is growing in importance for these tasks. In essence, Java programs use more method invocations (i.e., interprocedural control flow) and fewer control flow paths (i.e., intraprocedural control flow) compared with C programs. This work seeks to help enable the switch to dynamic context-sensitivity analyses by making them efficient enough for deployed systems.

3.1.1 Testing

Half of application development time is spent in testing [Bal01, Mye79]. A key part of testing is coverage, and one metric of coverage is exercising unique statements, paths, calling contexts [Bin97], and calling sequences that include the order of calls and returns [HRS$^+$00, RKS05]. *Residual testing* identifies untested coverage, such as paths, that occur at production time but were not observed during testing [PY99, VNC07]. PCC is well suited to context-sensitive residual testing since it identifies new contexts with high probability while adding low enough overhead for deployed software.

3.1.2 Debugging

Identifying program behavior correlated with incorrect execution often helps programmers find bugs. Previous work in invariant-based bug detection tracks program behavior such as variables' values across multiple runs to identify behavior that is well-correlated with errors [EPG$^+$07, HL02, LNZ$^+$05, LTQZ06, ZLF$^+$04]. We are not aware of work that uses calling context for invariant-based bug detection, although the high time and space overhead may have been a factor. Some previous work uses a limited amount of calling context in features in bug detection. Liu et al. use *behavior graphs*, which include call relationships (essentially one level of context sensitivity), to help identify call chains correlated with bugs [LYY$^+$05]. *Clarify* uses *call-tree profiling*, which measures two levels of context sensitivity as well as the order of calls, to classify program executions for better error reporting, a task similar to bug-finding [HRD$^+$07].

Programmers already appreciate the usefulness of calling context in debugging tasks. For example, developers typically start with an error stack trace to diagnose a crash and *Valgrind*, a testing-time tool, reports context-sensitive allocation sites for heap blocks involved in errors [NS07].

Artemis provides a framework for selectively sampling bug detection instrumentation to keep overhead low [FM06]. The key idea is to track contexts and to avoid sampling contexts that have already been sampled. Artemis's definition of context includes values of local and global variables but does not include calling context. PCC makes it viable to add calling context to Artemis because of its low cost.

3.1.3 Security

Anomaly-based intrusion detection seeks to detect new attacks by identifying anomalous (i.e., previously unseen) program behavior [FKF+03, IF02, WS02]. Existing approaches typically keep track of system calls and flag system call sequences that deviate from previously-observed behavior and may indicate an attacker has hijacked the application. Wagner and Soto show that attackers can circumvent these approaches by *mimicking* normal application behavior while still accomplishing attacks [WS02]. Adding context sensitivity to the model of acceptable behavior constrains what attackers can do without getting caught, and recent work on intrusion detection uses calling context to identify program control-flow hijacking [FKF+03, Ino05, ZZPL05]. Inoue on page 109 in his dissertation writes the following [Ino05]:

> Adding context by increasing the number of observed stack frames can make some attacks significantly more difficult. So-called "mimicry" attacks take advantage of the inner workings of applications to attack while still behaving similarly to the attacked application. Adding context makes this more difficult because it restricts the attacker to using only methods usually invoked from within the enclosing method that the exploit attacks, instead of any method invoked by the entire application.

Zhang et al. show that k-length interprocedural paths gathered with hardware reveal possible security violations [ZZPL05]. Feng et al. utilize a single level of context sensitivity by including each system call's return address in the sequence of system calls, constraining possible attacks [FKF+03].

We show that the expense of walking the stack stands in the way of deployed use of context-sensitive system calls but that PCC permits cheap

computation of context sensitivity (Section 3.4). An intrusion detection system could use PCC to record the calling context for each system call in sequences of system calls. Because PCC is probabilistic, it may incur false negatives if it misses anomalous calling contexts that map to the same value as an already-seen calling context. However, the conflict rate is very low, 0.1% or less for up to 10 million contexts with 32-bit values, and 64-bit values provide even fewer conflicts. A determined attacker with knowledge of PCC could potentially engineer an attack using an anomalous calling context with a conflicting PCC value. We believe randomizing call site values on the host would make a "conflict attack" virtually impossible, although we do not prove it.

Existing work shows dynamic calling context is useful for residual testing, invariant-based bug detection, and anomaly-based intrusion detection. Trends toward managed languages and more complex applications are likely to make dynamic context sensitivity more essential, and PCC makes it feasible.

3.2 Probabilistic Calling Context

This section describes our approach for efficiently computing a value that represents the current calling context and is unique with high probability.

3.2.1 Calling Context

The current program location (method and line number) and the active call sites on the stack define dynamic calling context. For example, the first line below is the current program location, and the remaining lines are the active call sites:

```
at com.mckoi.database.jdbcserver.JDBCDatabaseInterface.execQuery():213
at com.mckoi.database.jdbc.MConnection.executeQuery():348
at com.mckoi.database.jdbc.MStatement.executeQuery():110
at com.mckoi.database.jdbc.MStatement.executeQuery():127
at Test.main():48
```

3.2.2 Probabilistic Approach

Probabilistic calling context (PCC) keeps track of an integer value, V, that represents the current calling context. Our goal is to compute random, independent values for each context. To determine the feasibility of this approach, we assume a random number generator and use the following formula to determine the number of expected conflicts given population size n and 32- or 64-bit values [MU05]:

$$E[\mathit{conflicts}] \equiv n - m + m\left(\frac{m-1}{m}\right)^n$$

where m is the size of the value range (e.g., $m = 2^{32}$ for 32-bit values). Table 3.1 shows the expected number of conflicts for populations ranging in size from one thousand to ten billion. For example, if we choose 10 million random 32-bit numbers, we can expect 11,632 conflicts on average. Applied to the calling context problem, if a program executes 10 million distinct calling contexts, we expect to miss contexts at a rate of just over over 0.1%, which is good enough for many clients.

The programs we evaluate execute fewer than 10 million distinct calling contexts (except eclipse with the *large* input; Section 3.4.1). For programs with many more distinct calling contexts, or for clients that need greater probability guarantees, 64-bit values should suffice. For example, one can expect only a handful of conflicts for as many as 10 billion distinct calling contexts.

Random	Expected conflicts	
values	32-bit values	64-bit values
1,000	0 (0.0%)	0 (0.0%)
10,000	0 (0.0%)	0 (0.0%)
100,000	1 (0.0%)	0 (0.0%)
1,000,000	116 (0.0%)	0 (0.0%)
10,000,000	11,632 (0.1%)	0 (0.0%)
100,000,000	1,155,170 (1.2%)	0 (0.0%)
1,000,000,000	107,882,641 (10.8%)	0 (0.0%)
10,000,000,000	6,123,623,065 (61.2%)	3 (0.0%)

Table 3.1: **Expected conflicts for various populations of random numbers using 32-bit and 64-bit values.**

3.2.3 Computing Calling Context Values

The previous section shows that assigning randomly-chosen PCC values results in an acceptably small level of conflicts (i.e., distinct calling contexts with the same value). This section introduces an online approach for computing a PCC value that has the following properties:

- PCC values must be distributed roughly randomly so that the number of value conflicts is close to the ideal.
- The PCC value must be deterministic, i.e., a given calling context always computes the same value.
- Computing the next PCC value from the current PCC value must be efficient.

We use a function

$$f(V, cs)$$

where V is the current calling context value and *cs* is the call site at which the function is evaluated. We add instrumentation that computes the new value of V at each call site by applying f as follows:

```
method() {
       int temp = V;      // ADDED: load PCC value
       ...
       V = f(temp, cs_1); // ADDED: compute new value
 cs_1: calleeA(...);      // call site 1
       ...
       V = f(temp, cs_2); // ADDED: compute new value
 cs_2: calleeB(...);      // call site 2
       ...
}
```

We have two requirements for this function: non-commutativity and efficient composability.

Non-commutativity. We have found that our benchmarks contain many distinct calling contexts that differ only in the order of call sites. For example, we want to differentiate calling context ABC from CAB. We therefore require a function that is *non-commutative* and thus computes a distinct value when call sites occur in different orders. This requirement rules out functions such as $f(V, cs) \equiv \times V \mathrm{XOR} cs$.

Efficient composability. We want to handle method inlining efficiently and gracefully because of its widespread use in high-performance static and dynamic compilers. For example, suppose method A calls B calls C calls D. If the compiler inlines B and C into A, now A calls D. We want to avoid evaluating f three times before the inlined call to D. By choosing a function whose composition can be computed efficiently ahead-of-time,

we can statically compute the *inlined* call site value that represents the sequence of call sites B, C, D.

We use the following non-commutative but efficiently composable function to compute PCC values:

$$f(V, cs) \equiv 3 \times V + cs$$

where $\times$ is multiplication (modulo 2^{32}), and $+$ is addition (modulo 2^{32}). We statically compute *cs* for a call site with a hash of the method and line number.

The function is non-commutative because evaluating call sites in different orders do not give the same value in general:

$$\begin{aligned} f(f(V, cs_A), cs_B) &= 9 \times V + (3 \times cs_A) + cs_B) \\ \neq \quad f(f(V, cs_B), cs_A) &= 9 \times V + (3 \times cs_B) + cs_A) \end{aligned}$$

since in general

$$(3 \times cs_A) + cs_B \quad \neq \quad (3 \times cs_B) + cs_A$$

Non-commutativity is a result of mixing addition and multiplication (which are commutative operations by themselves). At the same time, the function's composition is efficient because addition and multiplication are distributive:

$$\begin{aligned} & f(f(V, cs_A), cs_B) = \\ & \quad 3 \times (3 \times V + cs_A) + cs_B = \\ & \quad 9 \times V + (3 \times cs_A + cs_B) \end{aligned}$$

Note that $(3 \times cs_A + cs_B)$ is a compile-time constant, so the composition is as efficient to compute as f.

Gropp and Langou et al. use similar functions to compute hashes for Message Passing Interface (MPI) data types [Gro00, LBFD05]. We experimented with these and other related functions. For example, multiplying by 2 is attractive because it is equivalent to bitwise shift, but bits for methods low on the stack are lost as they are pushed off to the left. Circular shift (equivalent to multiplication by 2 modulo $2^{32} - 1$) solves this problem, but when combined with addition modulo $2^{32} - 1$ (necessary to keep efficient composability), it loses information about multiple consecutive recursive calls. That is, 32 consecutive recursive calls compute $f^{32}(V, cs)$, which for this function is simply V for any V and cs.

3.2.4 Querying Calling Context Values

This section describes how clients can query PCC values at program points. In any given method, V represents the current dynamic context, *except for the position in the currently executing method*. To check V at a given program point, we simply apply f to V using the value of cs for the current site (not necessary a call site), i.e., current local method and line number:

```
method() {
      ...
 cs: query(f(V, cs));       // ADDED: query PCC value
     statement_of_interest; // application code
      ...
}
```

PCC is most applicable to clients that detect new or anomalous behavior, which naturally tend to have two modes, *training* and *production*. In training, clients query PCC values and store them. In production, clients query PCC values and determine if they represent anomalous behavior; if so, PCC walks

the stack to determine the calling context represented by the anomalous PCC value. Many anomalous contexts in production could add high overhead because each new context requires walking the stack. However, this case should be uncommon for a well-trained application.

3.3 Implementation

PCC's approach is suitable for implementation in ahead-of-time or dynamic compilation systems. This section describes details of our implementation in Jikes RVM.

3.3.1 Computing the PCC value

PCC adds instrumentation to maintain V that computes $f(V, cs)$ at each call site, where cs is an integer representing the call site. PCC could assign each call site a random integer using a lookup table, but this approach adds space overhead and complicates comparing PCC values across runs. PCC computes a hash of the call site's method name, declaring class name, descriptor, and line number. This computation is efficient because it occurs once at compile time and produces the same results across multiple program executions.

V is a thread-local variable modified at each call site. Since multiple threads map to a single processor, each processor keeps track of the PCC value for the current thread. When a processor switches threads, it stores the PCC value to the outgoing thread and loads the PCC value from the incoming thread. Accessing the PCC value is efficient in Jikes RVM because it reserves a register for per-processor storage. In systems without efficient access to per-processor storage, an implementation could modify the calling conventions to add the PCC value as an implicit parameter to every method. While this alternative approach is elegant, we did not implement it because it would require pervasive changes to Jikes RVM.

To compute PCC values, the compiler adds instrumentation that (1) at the beginning of each method, loads V into a local variable, (2) at each call site, computes the next calling context with f and updates the global V, and (3) at the method return, stores the local copy back to the global V (this redundancy is helpful for correctly maintaining V in the face of exception control flow). At inlined call sites, the compiler combines multiple call site values ahead-of-time into a single value and inserts a function that is an efficient composition of multiple instances of f.

3.3.2 Querying the PCC value

Clients may query PCC values at different program points, and they may use PCC values differently. For example, an intrusion detection client might query the PCC value at each system call, recording sequences of consecutive context-sensitive program locations (in the form of PCC values) during training, then detecting anomalous sequences during production. A client performs work per query that is likely to be similar to hash table lookup, so our implementation looks up the PCC value in a global hash table at each query point. The hash table implements *open-address hashing* and *double hashing* [CLRS01] using an array of 2^k 32-bit slots. To look up a PCC value, the *query* indexes the array using the low k bits of V, and checks if the indexed slot contains V. In the common case, the slot contains V, and no further action is needed. In the uncommon case, either (1) the slot is empty (contains zero), in which case PCC stores V in the slot; or (2) the slot holds another

PCC value, in which case the *query* performs secondary hashing by advancing $s+1$ slots where s is the high $32-k$ bits of V. Secondary hashing tries three times to find a non-conflicting slot. If it fails, it stops trying to find a slot, and it increments a variable that counts hashing failures.

For efficiency, we inline the common case into hot, optimized code. For simplicity in our prototype implementation, we use a fixed-size array with $2^{20} = 1,048,576$ elements (4 MB of space), but a more flexible implementation would adjust the size to accommodate the number of stored PCC values collected during training (e.g., intrusion detection clients could use much less space since there are relatively few distinct contexts at system calls). Of our benchmarks, `pmd` queries the most distinct PCC values, over 800,000 at Java API calls (Table 3.3), for which a hash table with a million elements is probably not quite large enough for good performance. We also measure the overhead of querying the PCC value at every call as an upper bound for a PCC client (Figure 3.1); for several benchmarks with millions of distinct contexts, the hash table is not large enough, resulting in many hash table lookup failures, but the time overhead should still be a representative upper bound.

3.3.3 Defining Calling Context

Our implementation distinguishes between VM methods (defined in Jikes RVM classes), Java library methods (`java.*` classes), and application classes (all other classes). The implementation does not consider VM and library call sites to be part of calling context, since call sites in these methods are probably not interesting to developers and are often considered "black boxes." All application methods on the stack are considered part of the calling context, even if VM or library methods are above them. For example, container classes often access application-defined equals() and hashCode() methods:

```
at result.Value.equals():164
at java.util.LinkedList.indexOf():406
at java.util.LinkedList.contains():176
at option.BenchOption.getFormalName():80
at task.ManyTask.main():46
```

Our implementation considers this context to be simply

```
at result.Value.equals():164
at option.BenchOption.getFormalName():80
at task.ManyTask.main():46
```

Similarly, sometimes the application triggers the VM, which calls the application, such as for class initialization:

```
at dacapo.TestHarness.<clinit>():57
at com.ibm.JikesRVM.classloader.VM_Class.initialize():1689
at com.ibm.JikesRVM.VM_Runtime.initializeClassForDynamicLink():545
at com.ibm.JikesRVM.classloader.VM_TableBasedDynamicLinker.resolveMember():65
at com.ibm.JikesRVM.classloader.VM_TableBasedDynamicLinker.resolveMember():54
at Harness.main():5
```

Our implementation considers this context to be

```
at dacapo.TestHarness.<clinit>():57
at Harness.main():5
```

PCC implements this definition of calling context. PCC instruments application methods only, and in these methods it instruments call sites to application and library methods. In cases where the application calls the VM directly, and the VM then invokes the application (e.g., for class initialization), PCC walks the stack to determine the correct value of V, which is feasible because it happens infrequently.

3.4 Results

This section evaluates the performance and accuracy of probabilistic calling context (PCC). It first describes *deterministic* calling context profiling, which we use to measure the accuracy of PCC. Then we present the query points we evaluate, which correspond to potential clients of PCC. Next we evaluate PCC's accuracy at identifying new contexts at these query points, then measure PCC's time and space performance and compare it to walking the stack. Finally we evaluate PCC's ability to identify new contexts not observed in a previous run and the power of calling context to detect new program behavior not detectable with context-insensitive profiling.

3.4.1 Deterministic Calling Context Profiling

To evaluate the accuracy of PCC and to collect other statistics, we also implement *deterministic* calling context profiling. Our implementation constructs a calling context tree (CCT) and maintains the current position in the CCT throughout execution. Our implementation is probably less time and space efficient than the prior work (Section 8.2.6) because (1) it collects per-node statistics during execution, and (2) for simplicity, we modify only the non-optimizing baseline compiler and disable the optimizing compiler for these experiments only. Since we only use it to collect statistics, we are not concerned with its performance.

For all performance and statistics runs, we use *large* benchmark inputs, except we use *medium* for eclipse's statistics runs. With *large*, our deterministic calling context implementation runs out of memory because eclipse with this input executes at least 41 million distinct contexts.

3.4.2 Potential PCC Clients

PCC continuously keeps track of a probabilistically unique value that represents the current dynamic calling context. To evaluate PCC's use in several potential clients, we query the PCC value at various program points corresponding to these clients' needs.

System calls. Anomaly-based security intrusion detection typically collects sequences of system calls, and adding context-sensitivity can strengthen detection (Section 3.1). To explore this potential client, we add a call to PCC's *query* method before each system call, i.e., each call that can potentially throw a Java security exception. The callees roughly correspond to operations that can affect external state, e.g., file system I/O and network access. Our benchmarks range in behavior from very few to many system calls. Programs most prone to security intrusions, such as web servers, are likely to have many system calls.

Java utility calls. Residual testing of a software component at production time detects if the component is called from a new, untested context. While application developers often perform residual testing on a component of their own application, we use the Java *utility* libraries as a surrogate for exploring residual testing on a component library. These libraries provide functionality such as container classes, time and date conversions, and random numbers. At each call to a java.util.* method, instrumentation queries the PCC value.

Java API calls. We also explore residual testing using the Java *API* libraries as a surrogate by adding instrumentation at every call to a method in

java.*. This library is a superset of java.util. Using these methods simulates residual testing of a larger component, since calls to java.* methods, especially java.lang.* methods, are extremely frequent in most Java programs (e.g., all String operations are in the API libraries).

All calls. Finally, we evaluate querying the PCC value at every call site. This configuration would be useful for measuring code coverage and generating tests with good code coverage [Bin97, HRS$^+$00, RKS05], and it represents an upper bound on overhead for possible PCC clients. We find querying PCC values at every call is too expensive for deployed use but speeds up testing time compared with walking the stack.

3.4.3 PCC Accuracy

Table 3.2 shows calling context statistics for the first three potential clients from the previous section. *Dynamic* is the number of dynamic calls to *query*. For example, for system calls, *Dynamic* is the dynamic number of system calls. *Distinct* is the number of distinct calling contexts that occur at query points. *Conf* is the number of PCC value conflicts that occur for these calling contexts. Conflicts indicate when PCC maps two or more distinct calling contexts to the same value (k contexts mapping to the same value count as $k - 1$ conflicts). We summarize the dynamic and distinct counts using geometric mean.

The benchmarks show a wide range of behavior with respect to system calls. Seven benchmarks perform more than 1,000 *dynamic* system calls, and two benchmarks (antlr and jython) exercise more than 1,000 distinct contexts at system calls. No PCC value conflicts occur between contexts.

Program	System calls			Java utility calls			Java API calls		
	Dynamic	Distinct	Conf	Dynamic	Distinct	Conf	Dynamic	Distinct	Conf
antlr	211,490	1,567	0	698,810	8,010	0	24,422,013	128,627	3
bloat	12	10	0	1,030,955,346	143,587	3	1,159,281,573	600,947	40
chart	63	62	0	43,345,653	44,502	0	258,891,525	202,603	4
eclipse	14,110	197	0	3,958,510	54,175	0	132,507,343	226,020	5
fop	18	17	0	5,737,083	25,528	0	9,918,275	37,710	0
hsqldb	12	12	0	90,324	267	0	81,161,541	16,050	0
jython	5,929	4,289	0	76,150,625	131,992	2	543,845,772	628,048	48
luindex	2,615	14	0	5,437,548	1,024	0	39,733,214	102,556	0
lusearch	141	11	0	23,183,861	176	0	113,511,311	905	0
pmd	1,045	25	0	372,159,946	442,845	24	537,017,118	847,108	79
xalan	137,895	59	0	744,311,518	6,896	0	2,105,838,670	17,905	0
DaCapo geo	843	60		19,667,815	12,689		163,072,787	85,963	
pseudojbb	507,326	145	0	18,944,200	475	0	30,340,974	3,410	0
compress	7	5	0	1,018	682	0	8,138	1,081	0
jess	50	6	0	4,851,299	2,061	0	16,487,052	5,240	0
raytrace	7	5	0	1,078	684	0	5,331,338	3,383	0
db	7	5	0	65,911,710	767	0	90,130,132	1,439	0
javac	7	5	0	6,499,455	55,994	0	24,677,625	255,334	4
mpegaudio	7	5	0	874	682	0	7,575,084	1,668	0
mtrt	7	5	0	880	682	0	5,573,455	3,366	0
jack	7	5	0	14,987,342	14,718	0	21,771,285	29,461	0
SPEC geo	30	7		199,386	1,724		7,074,200	5,410	
Geomean	188	23		2,491,316	5,168		39,734,213	24,764	

Table 3.2: **Statistics for calling contexts at several subsets of call sites.** Dynamic and distinct contexts, and PCC value conflicts, for (1) system calls, (2) Java utility calls, and (3) Java API calls.

As expected, the programs make significantly more calls into the utility libraries and the entire Java API. For the utility libraries, *dynamic* calls range from about a thousand for several SPECjvm98 benchmarks to a billion for bloat, and the number of unique contexts ranges from 176 to 442,845. For the Java API, the *dynamic* calls are up to 2 billion for xalan, and distinct contexts range from 905 to 847,108. These potential clients will therefore require many PCC value queries, but as we show in the next section, PCC is efficient even with this high load. The numerous zero entries in the Conf columns show that PCC is completely accurate in many cases. The conflicts are low—at most 79 for pmd's 847,108 distinct contexts at API calls—and are consistent with the ideal values from Table 3.1.

Table 3.3 presents calling context statistics for *all* executed contexts, as well as average and maximum call depth. To profile every context, we query calling context at every call site, as well as every method prologue in order to capture leaf calls. The table shows six programs execute over one million distinct contexts and another five over one hundred thousand contexts. The last two columns show that programs spend a lot of time in fairly deep call chains: average call depth is almost 20, and maximum call depth is over 100 for several benchmarks due to recursive methods.

3.4.4 PCC Performance

This section evaluates PCC's run-time performance. We evaluate PCC alone without a client and also measure the additional cost of using PCC with four sets of query points corresponding to potential clients. These experiments report application time only using replay compilation, which produces a deterministic measurement.

Program	All contexts Dynamic	Distinct	Conf	Call depth Avg	Max
antlr	490,363,211	1,006,578	118	21.5	164
bloat	6,276,446,059	1,980,205	453	30.6	167
chart	908,459,469	845,432	91	16.6	29
eclipse	1,266,810,504	4,815,901	2,652	15.0	102
fop	44,200,446	174,955	2	22.4	49
hsqldb	877,680,667	110,795	1	19.3	36
jython	5,326,949,158	3,859,545	1,738	58.3	223
luindex	740,053,104	374,201	12	19.4	34
lusearch	1,439,034,336	6,039	0	15.2	24
pmd	2,726,876,957	8,043,096	7,653	28.9	416
xalan	10,083,858,546	163,205	6	19.4	63
DaCapo geo	1,321,327,982	562,992		22.3	78
pseudojbb	186,015,473	19,709	0	7.1	25
compress	451,867,672	1,518	0	13.6	17
jess	198,606,454	18,021	0	43.1	83
raytrace	557,951,542	21,047	0	6.7	18
db	91,794,359	2,118	0	13.0	18
javac	135,968,813	2,202,223	544	29.5	122
mpegaudio	218,003,466	7,576	0	21.9	26
mtrt	564,072,400	21,040	0	6.7	18
jack	35,879,204	82,514	1	22.3	49
SPEC geo	200,039,740	20,695		14.8	32
Geomean	565,012,654	127,324		18.6	52

Table 3.3: **Statistics for every calling context executed.** Dynamic and distinct contexts, PCC value conflicts, and average and maximum size (call depth) of dynamic contexts.

Figure 3.1 shows the run-time overhead of PCC, normalized to *Base*, which represents execution without any instrumentation. *PCC* is the execution time of PCC alone: instrumentation keeps track of the PCC value throughout execution but does not use it. The final four bars show the execution time of examining the PCC value at call sites correspond to potential clients: system calls, Java utility calls, Java API calls, and all calls. PCC actually improves chart's performance, but this anomaly is most likely because of architectural sensitivities due code modifications that may affect the trace cache and branch predictor.

PCC by itself adds only 3% on average and 9% at most (for hsqldb). Since system calls are relatively rare, checking the context at each one adds negligible overhead on average. PCC value checking at Java utility and API calls adds 2% and 9% on average over PCC tracking, respectively, which is interesting given the high frequency of these calls (Table 3.2). The highest overhead is 47%, for bloat's API calls.

Compilation overhead. By adding instrumentation to application code, PCC increases compilation time. We measure compilation overhead by measuring time spent in the baseline and optimizing compilers during the first iteration of replay compilation. Figure 3.2 shows compilation time overhead. PCC instrumentation alone adds 18% compilation overhead on average. Adding instrumentation to query the PCC value increases compilation time by an additional 0-31% for system, utility, and API calls, and up to 150% for all calls, although this overhead could be reduced by not inlining the query method. Per-phase compiler timings show that most of the compilation overhead comes from compiler phases downstream from PCC instrumentation, due

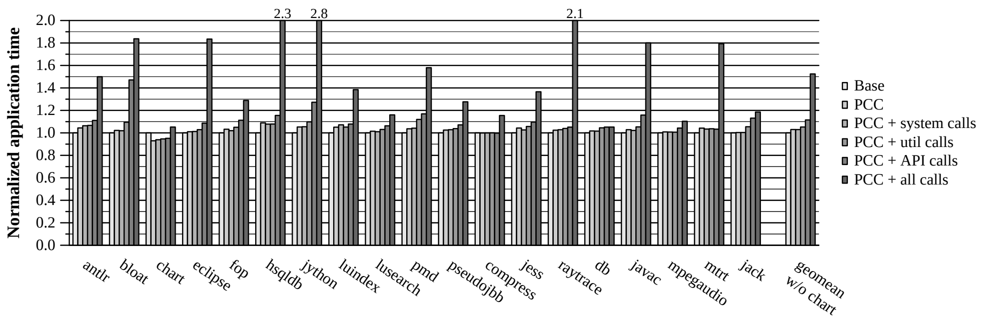


Figure 3.1: **Application execution time overhead of maintaining the PCC value and querying it.**

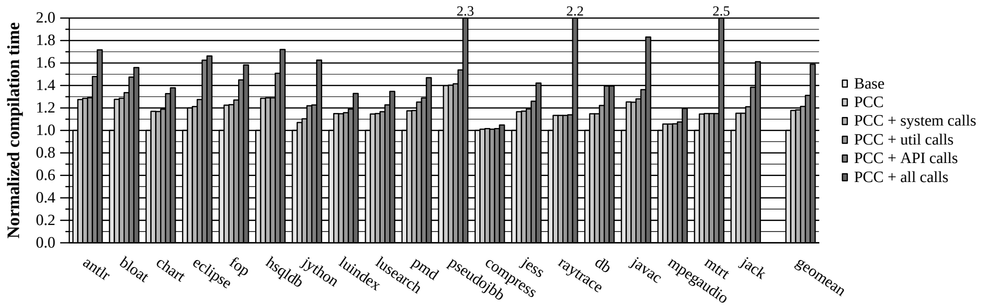


Figure 3.2: **Compilation time overhead due to adding instrumentation to maintain the PCC value and query it.**

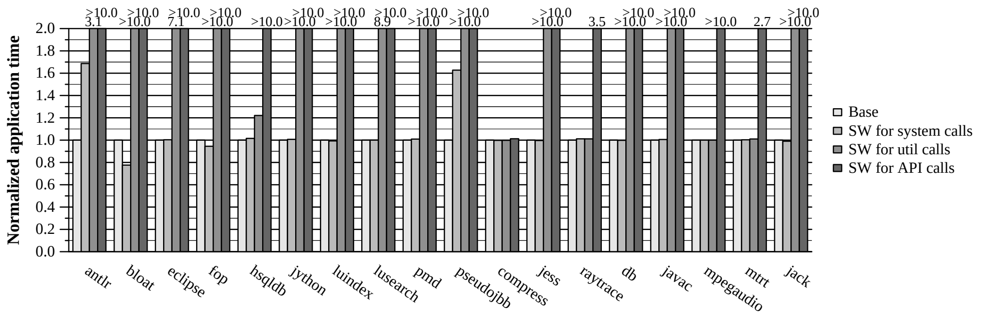


Figure 3.3: **Application execution time overhead of walking the stack.**

to the bloated intermediate representation (IR). By design, compilation time is a small fraction of overall execution time. In our experiments, the time spent in the application is on average 20 times greater than the time spent in compilation, for each of the PCC configurations shown in Figure 3.2.

Space overhead. Computing the PCC does not add space overhead to keep track of the PCC value, but of course the clients use space proportional to the number of PCC values they store. Our experiments that test potential clients simply use a fixed-size hash table with $2^{20} = 1,048,576$ slots (4 MB), as described in Section 3.3, but real clients would use space proportional to their needs. Clients storing PCC values in a large data structure could potentially hurt execution time due to poor access locality.

PCC adds space overhead and instruction cache pressure by increasing the size of generated machine code. We find that on average, PCC instrumentation adds 18% to code size. Adding instrumentation to query the PCC value at system calls, utility calls, API calls, and all calls adds an additional 0%, 2%, 6%, and 14%, respectively.

Comparison with stack-walking. An alternative to PCC is to walk the stack at each query point (Section 8.2.6). We evaluate here how well stack-walking performs for the call sites corresponding to potential clients. We implement stack-walking by calling a method that walks the entire stack at each query point; we do not add any PCC instrumentation for these runs. Stack-walking implementations would additionally look up a unique identifier for the current context [NS07], and they could save time by walking only the subset of calls occurring since the last walk [Wha00], but we do not implement these features here.

Figure 3.3 shows the execution time overhead of walking the stack at various points corresponding to three potential clients: system calls, Java utility calls, and Java API calls (we omit “all calls” because its overhead is greater than for Java API calls, which is very high). Since most benchmarks have few dynamic system calls, stack-walking adds negligible overhead at these calls. However, for the two benchmarks with more than 200,000 dynamic system calls, antlr and pseudojbb, stack-walking adds 67% and 62% overhead, respectively. These results show the substantial cost of walking the stack even for something as infrequent as system calls. Applications prone to security attacks such as web servers are likely to have many system calls.

3.4.5 Evaluating Context Sensitivity

Clients that detect new or anomalous behavior usually do not use dynamic context sensitivity because of its prior cost. This section compares calling context profiling to *call site profiling*, which is context *insensitive*, to evaluate whether calling context detects significantly more previously unobserved behavior than call sites alone. Table 3.4 compares calling contexts and call sites. The first two columns are counts of distinct calling contexts and call sites for calls to Java API methods (the calling context figures are the same as in Table 3.2). For most programs, there are many more calling contexts than call sites, which indicates that call sites are invoked from multiple calling contexts. The first two columns show that thousands of call sites generate hundreds of thousands of calling contexts.

Finally, we consider the power of residual calling context compared to residual call site profiling on the medium versus the large inputs. Columns in

	Large input		Large-medium diff		Contexts with
Program	Contexts	Call sites	Contexts	Call sites	new call sites
antlr	128,627	4,184	8	0	0
bloat	600,947	3,306	320,864	82	1,002
chart	202,603	2,335	139,599	379	9,112
eclipse*	226,020	9,611	121,939	1,240	46,206
fop	37,710	2,225	0	0	0
hsqldb	16,050	947	13	0	0
jython	628,048	1,830	59,202	1	1
luindex	102,556	654	7,398	0	0
lusearch	905	507	0	0	0
pmd	847,108	1,890	711,223	48	388
xalan	17,905	1,530	15	2	2
Dacapo geo	85,963	1,897			
pseudojbb	3,410	846	17	0	0
compress	1,081	1,017	0	0	0
jess	5,240	1,363	1,827	22	22
raytrace	3,383	1,215	25	2	5
db	1,439	1,105	72	4	4
javac	255,334	1,610	163,916	9	201
mpegaudio	1,668	1,072	32	1	4
mtrt	3,366	1,190	25	2	5
jack	29,461	2,173	0	0	0
SPEC geo	5,410	1,242			
Geomean	24,764	1,568			

Table 3.4: **Comparing call site profiles with calling context on Java API calls.** *Medium vs. small inputs for eclipse.

Table 3.4 under *Large-medium diff* count the distinct calling contexts and call sites seen in a large run but not a medium run. In several programs, many new distinct calling contexts occur, but many fewer new call sites occur, and, in particular, luindex executes 7,398 new contexts without executing any new call sites. The final column shows the number of new, distinct calling contexts that correspond to the new call sites in the large run. This column shows how well residual call site profiling would do at identifying new calling context behavior. If every new call site (i.e., call site seen in large but not medium run) triggered stack-walking, call site profiling would identify only a small fraction of the new calling contexts for most programs.

3.5 Conclusion and Interpretation

Complex object-oriented programs motivate calling context as a program behavior indicator in residual testing, invariant-based bug detection, and security intrusion detection. PCC provides dynamic context sensitivity for these clients with low overhead.

The bug diagnosis work in this dissertation tracks control and data flow and the relationship between them. PCC tracks control flow only, but it is directly useful for adding context sensitivity to invariant-based bug detection systems that correlate code and data [EPG+07, HL02, LNZ+05, LTQZ06, ZLF+04]. For example, PCC could improve the precision of DIDUCE and statistical bug isolation, which identify program locations where anomalous variable values occur [HL02, LNZ+05], by tracking each context-sensitive program location separately. Context sensitivity is fundamental for understanding and analyzing modern software since static program location is not enough in complex, object-oriented programs. By showing for the first time how to

represent dynamic context efficiently, this work promotes the use of context sensitivity in deployed debugging situations.

Chapter 4

Storing Per-Object Sites in a Bit with Bell

To find and fix bugs, programmers benefit from knowing which program locations (*sites*) are responsible for failures. Some bug detection and program understanding tools track sites for each object, such as the site that allocated the object or last accessed the object [CH04, LTQZ06]. Then when an error occurs, the tools report the program location(s) stored for objects associated with the error. Most programs written in managed languages have many small objects, and this approach adds space overhead too high for deployed systems.

This chapter introduces *Bell*, a novel approach for correlating object instances and sites with extremely low space overhead. Bell encodes the site for an object in a single bit using an *encoding* function $f(\mathit{site}, \mathit{object})$ that takes the site and the object address as input and returns zero or one. Bell thus loses information, but with sufficiently many objects and a known, finite set of sites, Bell can *decode* sites with high confidence. Decoding uses a brute-force application of the encoding function for all sites and a subset of objects. Bell can assist with a variety of tasks that can use statistical per-object information, both in managed and unmanaged languages. Bell can benefit debugging tasks that involve multiple errors, such as malformed data structures (e.g., a completely unbalanced binary search tree) [CG06b] and memory leaks. It can also help debugging of multiple running instances, either runs by multiple users or repeated runs by one user, by reconstructing culprit program locations

with confidence by combining information across runs.

We apply Bell to diagnosing memory leaks since they are an important and challenging bug type. Managed languages do not eliminate memory leaks since garbage collection cannot reclaim reachable but dead memory; leaks are hard reproduce since they often take days or longer to manifest; and they are hard to find and fix since they have no immediate symptoms [HJ92]. We use Bell as part of a new leak detector that identifies program sites responsible for memory leaks without adding any space overhead. It helps diagnose memory leaks in SPECjbb2000 and Eclipse [Eclb, Sta01], which have known memory leaks. The sites it reports are directly useful for fixing the leaks, although the programs need to run long enough to leak enough objects to be reported by Bell decoding. This work demonstrates that bug detection can in some cases store nontrivial information for every object *statistically*, reducing overhead drastically, while still recovering useful information.

4.1 Encoding and Decoding Per-Object Information

This section describes Bell, novel approach for encoding per-object information into a single bit.

4.1.1 Encoding

Bell *encodes* per-object information from a known, finite set in a single bit. In this work, we use Bell to encode *sites* such as source locations that allocate and use objects. A site can be a program counter (PC) value or a unique number that identifies a line in a source file. Bell's *encoding function*

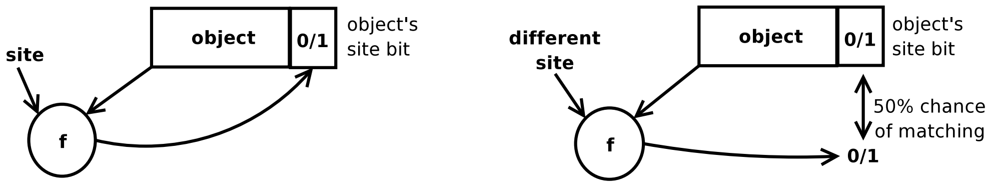


Figure 4.1: **Bell encoding.** (a) An object's encoded site is stored in its site bit. (b) A different site matches the object with $\frac{1}{2}$ probability.

takes two parameters, the site and object address, and returns zero or one:

$$f(\mathit{site}, \mathit{object}) = 0 \text{ or } 1$$

Bell computes $f(\mathit{site}, \mathit{object})$ and stores the result in the object's *site bit*, and we say the site was *encoded together with* the object. We say a site *matches* an object if $f(\mathit{site}, \mathit{object})$ equals the object's site bit. An object always matches the site it was encoded together with, but it may or may not match other sites. We choose f so it is *unbiased*: (1) with $\frac{1}{2}$ probability, a site matches an object encoded together with a different site, and (2) whether an object and site match is independent of whether another object matches the site. Figure 4.1 shows an example of the first property of an unbiased function. Section 4.1.3 presents several encoding functions that are unbiased and inexpensive to compute.

Since many sites (about half of all sites) may match an object, Bell loses information by encoding to a single bit. However, with enough objects, Bell can decode sites with high confidence.

4.1.2 Decoding

Bell *decodes* the sites for a subset of all objects. In this section, all mentions of objects refer to objects in this subset. In a leak detection tool, for example, Bell would decode the subset of objects the tool identified as potential leaks. Decoding reports sites encoded together with a significant number of objects, as well as the number of objects each site encodes (within a confidence interval). The key to decoding is as follows (recall that a site *matches* an object if $f(\mathit{site}, \mathit{object})$ equals the object's site bit).

> A site that *was not* encoded together with a significant number of objects will match about half the objects, whereas a site that *was* encoded together with a significant number of objects will match significantly more than half the objects.

In general, we expect a site encoded together with n_{site} objects (out of n objects in the subset) to match about $m_{site} = n_{site} + \frac{1}{2}(n - n_{site})$ objects, since the site matches (1) all of the n_{site} objects that *were* encoded together with it and (2) about half of the $n - n_{site}$ objects that were *not* encoded together with it. Solving for n_{site}, we find that about $n_{site} = 2m_{site} - n$ objects were encoded together with the site given that it matches m_{site} objects.

Bell decodes per-object sites using a brute-force approach that evaluates f for every object and every site:

```
foreach possible site
  $m_{site} \leftarrow 0$
  foreach object in the subset
    if $f(site, object) = object$'s site bit
      $m_{site} \leftarrow m_{site} + 1$
  print site has about $2m_{site} - n$ objects
```

	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$
10^3 sites	68	232	736	2,326
10^4 sites	72	248	784	2,480
10^5 sites	74	260	828	2,622
10^6 sites	78	272	868	2,752
10^7 sites	80	286	910	2,874

Table 4.1: **Values of n_{min} for example numbers of sites and objects (n).**

Because of statistical variability, $2m_{site} - n$ only approximates the number of objects encoded together with the site. Bell differentiates between sites that were actually encoded together with objects, and those that were not, by weeding out the latter with a *false positive threshold* m_{FP}:

```
**if** $m \geq m_{FP}$
  **print** *site* has about $2m_{site} - n$ objects
```

Section 4.2 describes how we compute m_{FP} so that decoding avoids false positives with high probability (99%). By weeding out sites, Bell misses sites that were encoded together with few but not many objects. We can compute the minimum number of objects n_{min} that need to be encoded together with a site, in order for Bell to report the site with very high probability (99.9%). Section 4.2 describes how we compute n_{min}.

Table 4.1 reports n_{min} for various numbers of sites and objects. It shows that n_{min} scales sublinearly with n (at a rate roughly proportional to $\sqrt{n}$). Thus, an increase in n requires more objects—but a *smaller fraction* of all objects—be encoded together with a site for Bell to report it. The table shows that n_{min} is not affected much by the number of sites, so Bell's precision scales well with program size.

4.1.3 Choosing the Encoding Function

This section presents the encoding functions we use. A practical encoding function should be both unbiased and inexpensive to compute, since applications of Bell will compute it at runtime. We find that taking a bit from the product of the site and the object address, meets both these criteria fairly well:

$$f_{singleMult}(site, object) \equiv \ bit_{31}(site \times object)$$

$f_{singleMult}$ returns the middle bit of the product of the site identifier and object address, assuming both are 32-bit integers. We find via simulation that for object addresses chosen randomly with few constraints, this function is unbiased (i.e., decoding does not report false positives or negatives more than expected). However, our leak detection implementation uses a segregated free list allocator, yielding non-arbitrary object addresses. Using $f_{singleMult}$ causes decoding to report a few more false positives than expected.

We find that the following encoding function eliminates unexpected false positives because the extra multiply permutes the bits enough to randomize away the regularity of object addresses allocated using a segregated free list:

$$f_{doubleMult}(site, object) \equiv bit_{31}(site \times object \times object)$$

We also experimented with

$$f_{parity}(site, object) \equiv parity(site \wedge object)$$

which returns the parity of the bitwise *AND* of the site and object address. While f_{parity} is unbiased if we choose object addresses randomly, site decoding returns many false positives if a segregated free list allocates objects since f_{parity} does not permute the bits of its inputs.

4.2 Avoiding False Positives and Negatives

Section 4.1.2 describes how Bell avoids false positives by not reporting sites that match less than m_{FP} objects, and how weeding out some sites requires that a site have been encoded together with at least n_{min} objects to be almost certainly reported. This section describes how we compute m_{FP} and n_{min}.

To compute m_{FP}, we use the fact that m_{site} (the number of objects that match a site) for a site encoded together with no objects, can be represented with a binomially-distributed random variable X with n trials and $\frac{1}{2}$ probability of success. (X is binomially distributed since whether a particular object matches the site is an independent event.) Solving for m_{FP} in the following equation gives the threshold needed to avoid reporting a single site as a false positive with high probability (99%):

$$1 - Pr(X \geq m_{FP}) \geq 99\%$$

We want to avoid reporting *any* false positive sites, so we solve for m_{FP} in the following equation:

$$[1 - Pr(X \geq m_{FP})]^{|sites|} \geq 99\%$$

where $|sites|$ is the number of possible sites.

Using m_{FP}, we compute n_{min} as follows. Given a site encoded together with n_{min} objects, we model the number of matches for the site as a binomially-distributed random variable Y with n trials and probability of success $\frac{1}{2}(n + n_{min})/n$ (because the expected value is $m_{site} = n_{min} + \frac{1}{2}(n - n_{min}) = \frac{1}{2}(n + n_{min})$). We solve for n_{min} in the following equation (note that m_{FP} is fixed, and n_{min} is implicitly in the equation as part of Y's probability of success):

$$1 - Pr(Y \geq m_{FP}) \geq 99.9\%$$

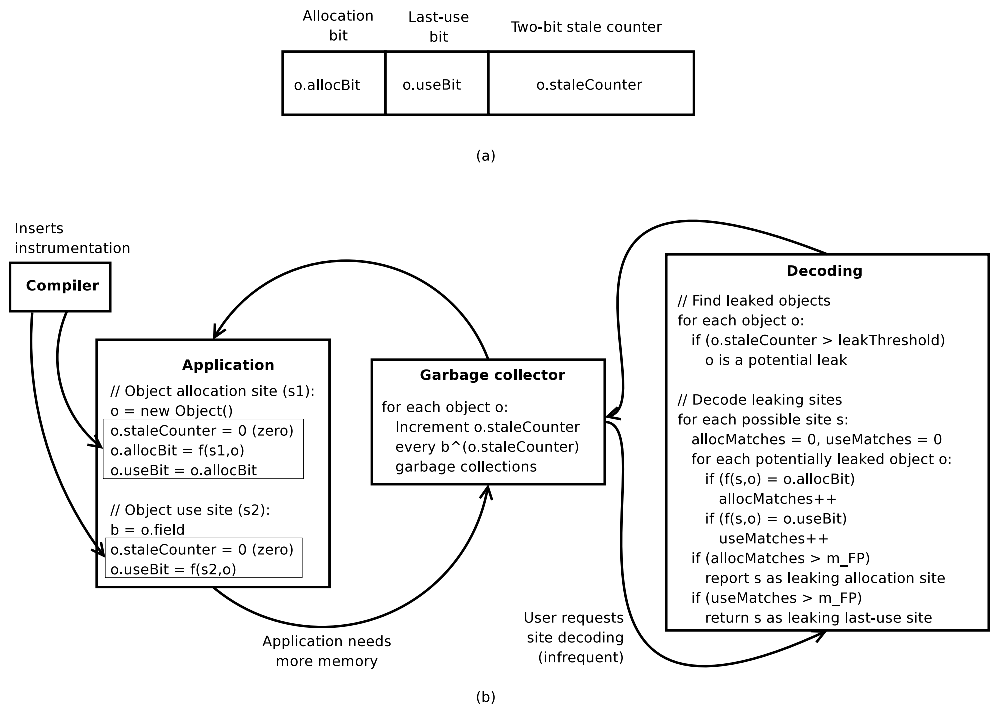
Table 4.1 showed example values of n_{min} for various numbers of sites and objects. Before decoding, our implementation solves for m_{FP} and m_{min} using the *Commons-Math* library [Apa].

4.3 Implementation

The section describes a memory leak detector we wrote that uses Bell to store and report per-object sites responsible for memory leaks. *Sleigh* encodes sites that allocated and last accessed each object, and it decodes these sites for highly *stale* objects.

4.3.1 Overview

Sleigh finds memory leaks in Java programs and reports the allocation and last-use sites of leaked objects, using just four bits per object. It inserts Bell instrumentation to encode object allocation and last-use sites in a single bit each, tracks object *staleness* (time since last use) in two bits using a logarithmic counter, and occasionally decodes the sites for stale objects. Sleigh borrows four unused bits in the object header in our implementation, so it adds no per-object space overhead. Other VMs such as IBM's J9 [GKS$^+$04] have free header bits. Without free header bits, Sleigh could store its bits outside the heap, efficiently mapping every two words (assuming objects are at least two words long) to four bits of metadata, resulting in 6.25% space overhead.



```
Application
// Object allocation site (s1):
o = new Object()
o.staleCounter = 0 (zero)
o.allocBit = f(s1,o)
o.useBit = o.allocBit

// Object use site (s2):
b = o.field
o.staleCounter = 0 (zero)
o.useBit = f(s2,o)
```

```
Garbage collector
for each object o:
  Increment o.staleCounter
  every b^(o.staleCounter)
  garbage collections
```

```
Decoding
// Find leaked objects
for each object o:
  if (o.staleCounter > leakThreshold)
    o is a potential leak

// Decode leaking sites
for each possible site s:
  allocMatches = 0, useMatches = 0
  for each potentially leaked object o:
    if (f(s,o) = o.allocBit)
      allocMatches++
    if (f(s,o) = o.useBit)
      useMatches++
  if (allocMatches > m_FP)
    report s as leaking allocation site
  if (useMatches > m_FP)
    return s as leaking last-use site
```

Figure 4.2: **Sleigh's components.** (a) Sleigh uses four bits per object. (b) Sleigh has several components that live in different parts of the VM.

Figure 4.2(a) shows the four bits that Sleigh uses in each object's header. Figure 4.2(b) shows the components that Sleigh adds to the VM. Sleigh uses the compiler to insert instrumentation in the application at object allocations (calls to `new`) and object uses (field and array element reads). It uses the garbage collector to increment each object's stale counter at a logarithmic rate. The garbage collector invokes decoding periodically or on demand. Decoding identifies allocation and last-use sites of potentially leaked objects.

4.3.2 Encoding Allocation and Last-Use Sites

Sleigh uses Bell to encode the allocation and last-use sites for each object using a single bit each. Sleigh adds instrumentation at object allocation that computes $f(\textit{site}, \textit{object})$ and stores the result in both the allocation bit and the last-use bit. If an object is never used, its last use is just its allocation site. Similarly, Sleigh adds instrumentation at object uses (field and array element reads) that computes $f(\textit{site}, \textit{object})$ and stores the result in the last-use bit. Figure 4.2(b) shows how the compiler inserts this instrumentation into application code.

Sleigh defines a site to be a calling context consisting of methods and line numbers (from source files), much like an exception stack trace in Java. For efficiency, Sleigh uses only the *inlined* portion of the calling context, which is known at compile time, whereas the rest of the calling context is not known until runtime. The following is an example site (the leaf callee comes first):

```
spec.jbb.infra.Factory.Container.deallocObject():352
  spec.jbb.infra.Factory.Factory.deleteEntity():659
    spec.jbb.District.removeOldestOrder():285
```

Sleigh assigns a unique random identifier to each unique site and maintains a mapping from sites to identifiers.

4.3.3 Tracking Staleness Using Two Bits

In addition to inserting instrumentation to maintain per-object allocation and last-use sites, Sleigh inserts instrumentation at each site that tracks object staleness using a two-bit saturating *stale counter*. The stale counter is *logarithmic*: its value is approximately the logarithm of the time since the application last used the object. A logarithmic counter saves space without losing much accuracy by representing low stale values with high precision and high stale values with low precision.

Sleigh resets an object's stale counter to zero at allocation and at each object use. Periodically, during garbage collection (GC), Sleigh updates all stale counters (Figure 4.2(b)). Sleigh updates stale counters by incrementing a counter from k to $k + 1$ only if the current GC number divides b^k evenly, where b is the base of the logarithmic counter (we use $b = 4$). k saturates at 3 because the stale counter is two bits. Stale counters implicitly divide objects into four groups: not stale, slightly stale, moderately stale, and highly stale. In our experiments, we consider the highly stale objects to be potential leaks. We find Sleigh is not very sensitive to the definition of highly stale objects since most objects are stale briefly or for a long time. Our Sleigh implementation fixes the logarithm base b at 4, but a more flexible solution could increase b over time to adjust to a widening range of object staleness values.

Sleigh updates objects' stale counters at GC time for efficiency and convenience. It measures staleness in terms of number of GCs but could measure staleness in terms of execution time instead by using elapsed time to determine whether and how much to increment stale counters.

4.3.4 Decoding

Sleigh occasionally performs Bell decoding to identify the site(s) that allocated and last used highly stale objects. Users can configure Sleigh to trigger decoding periodically (e.g., every hour or every thousand GCs) or on demand via a signal (not currently implemented). Decoding occurs during the next GC after being triggered. Figure 4.2(b) shows how GC occasionally invokes decod-

ing, and it shows pseudocode for decoding based on the decoding algorithm from Section 4.1.2. Decoding computes the number of objects that match each possible site, for both the object's allocation and last-use bits. It reports allocation and last-use sites that match more than m_{FP} objects (Section 4.1.2), and it reports the number of objects for each site, within a confidence interval.

Decoding is potentially expensive because its execution time is proportional to both the number of possible sites and number of highly stale objects. However, several factors mitigate this potential cost. First, we expect decoding to be an infrequent process, occurring only occasionally as needed on runs that last hours, days, or weeks and take as long to manifest significant memory leaks. Second, the vast majority of decoding's work can occur in parallel with the VM and application, on a different CPU or machine (currently unimplemented). The VM would need to provide the highly stale object addresses and the possible sites (or a delta since the last decoding) to a separate execution context, which would perform the brute-force application of the encoding function. Third, it is not necessary to perform decoding on all stale objects: a random sample of them suffices, although using fewer objects increases n_{min} and widens confidence intervals. Fourth, decoding could use type constraints (e.g., an object can only encode allocation sites that allocate the object's type) to significantly decrease the number of times Sleigh computes $f(\mathit{site}, \mathit{object})$ (currently unimplemented). Decoding runs in reasonable time in our experiments, and occasionally paying for decoding offers memory efficiency as compared with the all-the-time space overhead from storing un-encoded per-object sites.

Sleigh decodes allocation and last-use sites separately, but it could find and report allocation and last-use sites correlated with each other, as suggested by an anonymous reviewer.

4.3.5 Decreasing Instrumentation Costs

The instrumentation Sleigh adds at object uses (field and array reads) can be costly because it executes frequently. Sleigh removes redundant instrumentation and uses adaptive profiling [CH04] to reduce instrumentation overhead.

Removing redundant instrumentation. Instrumentation at object uses is required only at the *last* use of any object because the instrumentation at each use clears the stale counter and computes a new last-use bit. Sleigh can thus eliminate instrumentation at a use if it can determine that the use is followed by another use of the same object. A use is *fully redundant* if the same object is used later on every path. A use is *partially redundant* if the program uses the same object on some path. We use a backward, non-SSA, intraprocedural data-flow analysis to find partially redundant and fully redundant uses. Our analysis is similar to partial redundancy elimination (PRE) analysis [BC94], but is simpler because it computes redundant uses rather than redundant expressions.

We do *not* add instrumentation at fully redundant uses because they do not need it. We *do* add instrumentation at partially redundant uses, although we could remove it and add instrumentation along each path that does not use the object again. We have not implemented this optimization, but Section 4.5.4 evaluates an upper bound on its benefit.

Removing redundant instrumentation may cause Sleigh to report some in-use objects as stale if a long time passes between an uninstrumented use and an instrumented use. However, this effect can happen only to an object

referenced by a local (stack) variable continuously between the uninstrumented use to the instrumented use. We do not see inaccuracy in practice.

Adaptive profiling. Sleigh as described so far adds no per-object space overhead, but it does add 29% time overhead on average (Section 4.5.4). This time overhead is low compared to other memory leak detection tools (Section 8.2.4) but may be too expensive for online production use. To reduce this overhead, we borrow *adaptive profiling* from Chilimbi and Hauswirth [CH04], which samples instrumented code at a rate inversely proportional to its execution frequency. This approach maintains bug coverage while reducing overhead by relying on the hypothesis that cold code contributes disproportionately to bugs.

Sleigh optionally uses adaptive profiling to sample instrumentation at object uses. Since Bell decoding needs a significant number of objects to report a site, Sleigh uses all-the-time instrumentation at a site until it takes 10,000 samples. It progressively lowers the sampling rate by 10x every 10,000 samples until reaching the minimum sampling rate of 0.1%.

4.3.6 Memory Management

Since Bell's encoding function takes the object address as input, objects cannot move, or decoding will not work correctly. We use Jikes RVM's mark-sweep collector [BCM04], which allocates using a segregated free list and does not move heap objects.

Mark-sweep is not among the best-performing collectors. Sleigh could be modified to use a high-performance *generational* mark-sweep (GenMS) collector, which allocates objects in a small *nursery* and moves them to a mark-sweep *older space* if they survive a nursery collection. A GenMS-compatible Sleigh would (1) store *un-encoded* allocation and last-use sites (as extra header words) for nursery objects, (2) store *encoded* sites for older objects, and (3) when promoting objects from the nursery to the older space, encode each object's allocation and last-use sites using the object's new address in the older space and the object's un-encoded sites from the nursery. If the nursery were bounded, the space overhead added by un-encoded sites would be bounded.

Bell is incompatible with compacting collectors, which are popular in commercial VMs (e.g., JRockit [Oraa]) because they increase locality and decrease fragmentation. However, in some production environments it might be worthwhile to switch to generational mark-sweep in order to take advantage of Bell's space-saving benefits. Bell is compatible with C and C++ memory managers, since they do not move objects.

4.3.7 Miscellaneous Implementation Issues

Sleigh adds instrumentation to both application methods and library methods (the Java API) to reset objects' stale counters. Sleigh encodes allocation and last-use sites in application methods, but not in library methods since these sites are probably not helpful to the user and may obscure Sleigh's report. Sleigh *does* encode sites for library methods when they are inlined into application methods.

Because Jikes RVM is written in Java, the VM allocates its own objects in the heap together with the application's objects. These VM objects are not of interest to application developers, and thus Sleigh differentiates VM and application objects at allocation time using a fifth bit in the object header (a more elegant solution would put application and VM objects in separate heap

spaces). Bell decoding then ignores these VM objects.

4.4 Finding Leaks

This section evaluates Sleigh's ability to find leaks and help developers fix leaks.

4.4.1 Methodology

Execution. We execute Sleigh by running a production build of Jikes RVM (*FastAdaptive*) for two hours. We use a variable-sized heap (Jikes RVM automatically and dynamically adjusts the heap size) since leaks cause live memory to grow over time. In Sections 4.4.2 and 4.4.3, Sleigh inserts all-the-time instrumentation at object uses and removes instrumentation from fully but not partially redundant uses (this configuration is called *Sleigh default* in Section 4.5). In Section 4.4.4, Sleigh samples object uses using adaptive profiling (*Sleigh AP* in Section 4.5). We show just one trial per experiment since averaging Sleigh's statistical output over multiple runs makes its accuracy seem unfairly high, but we have verified that the presented results are typical from run to run.

Decoding. Decoding can process every (highly) stale object in the heap. However, we have found that many stale objects are pointed at by only other stale objects, i.e., they are just interior members of stale data structures. Sleigh's staleness-based approach implicitly divides the heap into two parts: in-use and stale objects. Figure 4.3 shows in-use and stale objects in a cross-section of the heap. Conceptually, an *in-use/stale border* divides the in-use and stale objects; this border consists of references from in-use to stale objects.

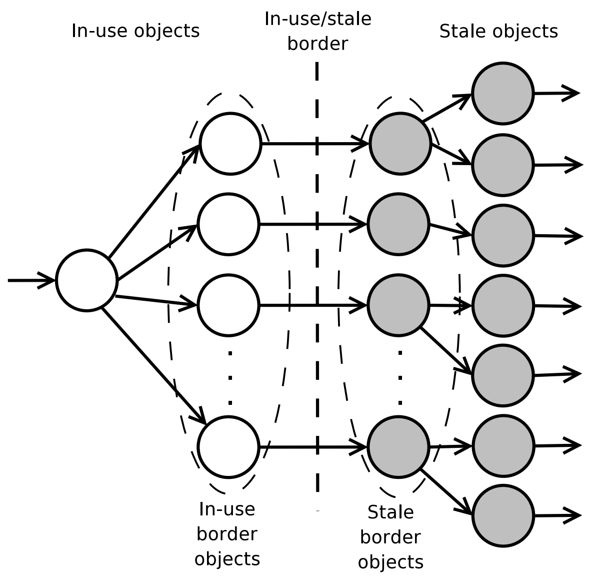


Figure 4.3: **Sleigh implicitly divides the heap into in-use and stale objects.**

We define a stale object pointed at by an in-use object as a *stale border object*, and an in-use object that points to a stale object as an *in-use border object*. Stale border objects are effectively the "roots" of stale data structures, and decoding these objects gives the allocation and last-use sites for these data structures. In-use border objects point to stale data structures, so decoding their sites may help answer the question, "Why is the stale data structure not being used anymore?" We note we had the idea to investigate stale and in-use border objects *after* examining the output from decoding all stale objects and fixing the Eclipse leak. Limiting decoding to border objects may be more important in Java since data structures typically consist of many objects, whereas Chilimbi and Hauswirth report success using sites for all stale objects in C [CH04].

We configure Sleigh to execute decoding every 20 minutes. Decoding processes and reports sites for three different subsets of objects: (1) all stale objects, (2) stale border objects, and (3) in-use border objects. Whenever one of these subsets has more than 100,000 objects, decoding processes a sample of 100,000 of them.

We plot reported object counts for reported sites with respect to time, which shows the sites that are growing. (Identifying growing sites is currently a manual process, but Sleigh could automatically find growing sites by analyzing the plotted data.) In this section, we are primarily interested in growing sites, since they will eventually crash programs. However, program developers might also be interested in non-growing sites, since unused memory may indicate poor memory usage.

Benchmarks. We evaluate Sleigh on two leaks in SPECjbb2000 and Eclipse 3.1.2 [Eclb, Sta01].

4.4.2 SPECjbb2000

SPECjbb2000 simulates an order processing system and is intended for evaluating server-side Java performance [Sta01]. SPECjbb2000 contains a known, growing memory leak that manifests when it runs for a long time without changing warehouses. The leak occurs because SPECjbb2000 adds but does not correctly remove orders from an order list that is supposed to have zero net growth.

We use Sleigh to find and help fix the leak. Table 4.2 presents statistics from running Sleigh on SPECjbb2000 for three subsets of stale and in-use objects. The first three labeled columns give the size of the object subset, the number of program sites, and decoding's execution time; the data are ranges over the six times decoding executes during a two-hour run. As expected, the number of stale objects grows over time as the leak grows (the number of stale objects starts high due to unused `String` and `char[]` objects that appear to be SPECjbb2000's "data"). The number of sites increases as dynamic compilation adds more sites. The last two columns show how many allocation and last-use sites decoding reports, and how many of these sites' object counts grow over time (based on manual inspection of plots with respect to time).

Figures 4.4 and 4.5 plot the sites for stale border objects (the dashed line is the minimum object count n_{min}). In general, we expect the plots for stale border objects to be most useful because they show site(s) where the roots of stale data structures were allocated and last used. Figure 4.4 reports one growing and one non-growing allocation site; the growing site is the generic `Class.newInstance()`, which is not very useful information. Last-use sites are more useful in this case, and we expect them to be more useful in general for pinpointing an unintentional leak's cause. Figure 4.5 shows two growing and two non-growing last-use sites with enough stale objects to be reported by decoding. One of the two growing sites Sleigh reports is the following:

```
spec.jbb.infra.Factory.Container.deallocObject():352
  spec.jbb.infra.Factory.Factory.deleteEntity():659
    spec.jbb.District.removeOldestOrder():285
```

This site is the key to fixing SPECjbb2000's leak: the fix replaces SPECjbb-2000's only call to removeOldestOrder() with two different lines that properly remove orders from SPECjbb2000's order list. Thus the three lines of inlined calling context that Sleigh provides are enough to pinpoint the exact line responsible for the leak. We believe a SPECjbb2000 developer could quickly

Instru-mentation	Object subset	Objects	Possible sites	Decoding time (s)	Growing (all) reported sites: Allocation	Growing (all) reported sites: Last use
All-the-time	All stale	60,610–73,175	4,412–4,476	2.0–2.5	3 (8)	3 (10)
	Stale border	24,454–28,639	4,412–4,476	0.8–1.0	1 (2)	2 (4)
	In-use border	239,603—420,128*	4,412–4,476	3.4–3.4	3 (6)	3 (14)
Adaptive profiling	All stale	103,228–127,917*	4,302–4,384	3.2–3.2	1 (7)	3 (14)
	Stale border	50,905–60,008	4,302–4,384	1.6–2.0	0 (4)	3 (10)
	In-use border	225,876–459,393*	4,302–4,384	3.2–3.2	3 (6)	2 (11)

Table 4.2: **Decoding statistics for Sleigh running SPECjbb2000.** *Decoding processes at most 100,000 objects.

Instru-mentation	Object subset	Objects	Possible sites	Decoding time (s)	Growing (all) reported sites: Allocation	Growing (all) reported sites: Last use
All-the-time	All stale	1,616,736–8,936,357*	31,733–32,574	24.2–24.9	7 (14)	10 (17)
	Stale border	40,492–43,360	31,733–32,574	10.0–10.9	1 (3)	2 (3)
	In-use border	40,572–454,975*	31,733–32,574	10.3–24.7	1 (7)	0 (10)
Adaptive profiling	All stale	1,683,898–9,022,732*	31,151–32,000	23.1–23.8	7 (7)	7 (12)
	Stale border	34,093–36,241	31,151–32,000	8.0–8.6	1 (3)	1 (2)
	In-use border	37,440–361,703*	31,151–32,000	9.0–23.5	0 (7)	0 (5)

Table 4.3: **Decoding statistics for Sleigh running Eclipse.** *Decoding processes at most 100,000 objects.

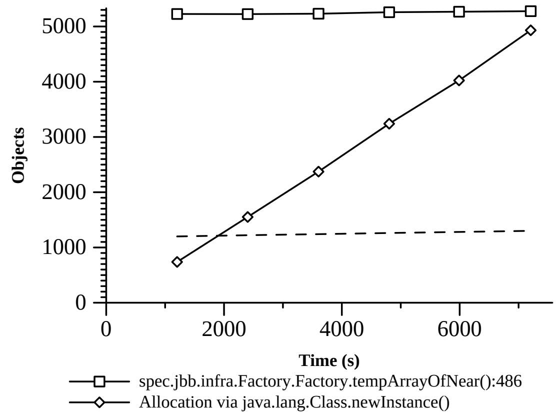


Figure 4.4: **Reported *allocation sites* for SPECjbb2000 when decoding processes *stale border objects* only.**

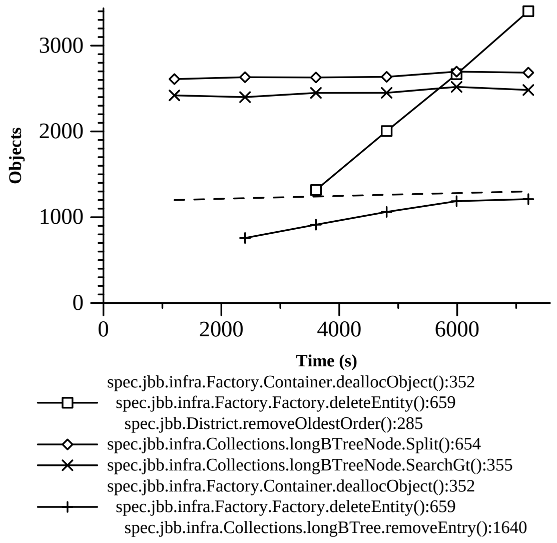


Figure 4.5: **Reported *last-use sites* for SPECjbb2000 when decoding processes *stale border objects* only.**

fix the leak based on Figure 4.5. The key site takes some time (about an hour) to manifest since decoding requires about $n_{min} = 1200$ objects (dashed line) to report the site. The last-use plot for all stale objects (not shown) also includes the key site, as well as several other sites, including two growing sites for *non-border* stale objects. The key site takes longer to manifest in this case since n_{min} increases with n (Section 4.1.2). The last-use plot for in-use border objects (not shown) does not show the key site above, which is not surprising since decoding operates on an entirely different subset of objects. At this time we do not understand SPECjbb2000 well enough to know if the plot for in-use objects is useful for fixing the leak.

SPECjbb2000's heap growth is due to both stale and in-use objects: `Order`s grow in number but are used, whereas `Container`s become stale. The fix described above eliminates only heap growth due to in-use objects, which contribute the vast majority (or perhaps all) of the heap growth in terms of bytes. Sleigh reports the offending last-use site because the in-use and stale objects are related (orders point to containers). At this time we do not understand SPECjbb2000 well enough to determine if the stale container objects are a leak or how to fix this potential leak, although the fix described above appears to eliminate all sustained heap growth.

4.4.3 Eclipse

Eclipse 3.1.2 is a popular integrated development environment (IDE) written in Java [Eclb]. Eclipse is a good target because it is a large, complex program (over 2 million lines of source code). The Eclipse bug repository reports several unfixed memory leaks. We pick unfixed bug #115789, which reports that repeatedly performing a structural (recursive) *diff* leaks memory that eventually exhausts available memory. We automate the GUI behavior that performs a repeated structural diff on MMTk source code [BCM04] before and after implementing Sleigh (17 of 250 files differ; textual diff is 350 lines).

The leak occurs in Eclipse's `NavigationHistory` component, which allows a user to step backward and forward through browsed editor windows. This component keeps a list of `NavigationHistoryEntry` (`Entry`) objects, each of which points to a `NavigationHistoryEditorInfo` (`EditorInfo`) object. In our test case, each `EditorInfo` points to a `CompareEditorInput` object, which is the root of a data structure that holds the results of the structural diff. The `NavigationHistory` component maintains the number of `Entry` objects that point to each `EditorInfo` object. If an `EditorInfo`'s count drops to zero, `NavigationHistory` removes the object. However, `NavigationHistory` erroneously omits the decrement in some cases, maintaining unnecessary pointers to `EditorInfo` objects. Because `NavigationHistory` regularly iterates through all `EditorInfo` objects but not pointed-to `CompareEditorInput` objects, the former are in-use border objects, and the latter are stale border objects.

Table 4.3 shows information about running Eclipse using Sleigh, in the same format as Table 4.2. Decoding all objects returns seven growing allocation and 10 growing last-use sites (plot not shown), most of which are for stale descendants of `CompareEditorInput` objects (i.e., the data for the structural diff).

Decoding *stale border* objects gives one growing allocation and two growing last-use sites. Figure 4.6 shows the last-use sites. The first growing last-use site, from `ElementTree`, is a red herring: this site's count grows and shrinks over time. It does not cause the sustained growing leak, but

it may be of interest to developers. The second growing last-use site, from `CompareEditorInput`, is in fact the last-use site for leaking `CompareEditorInput` objects. Unfortunately, the last-use site for these objects is not in or related to the `NavigationHistory` component.

We next try decoding sites for *in-use* border objects. Figure 4.7 plots the last-use sites for in-use border objects. It is not clear to us why the object counts of most reported sites decrease over time; perhaps Eclipse performs clean-up of pointers to unused objects as time passes. Almost two hours pass before Sleigh reports two sites from `NavigationHistory`, both of which are involved with `NavigationHistory`'s iteration through the list of `EditorInfo` objects. These sites do not have time to grow since the experiment ends after two hours, but a longer run shows that these sites do in fact grow. The plot of *allocation sites* for in-use border objects (not shown) also reports a site within `NavigationHistory` (the allocation site of `EditorInfo` objects) shortly before two hours pass.

Fixing the leak requires modifying a single line of code inside `NavigationHistory.java` to correctly decrement the reference count of each `EditorInfo` object. After determining that the `NavigationHistory` component was causing the leak by holding on to `EditorInfo` objects, we fixed the leak within an hour. Thus we believe Sleigh's output would help an Eclipse developer fix the leak quickly, although enough in-use border objects must leak first. We posted the leak's fix as an update to the bug report.

4.4.4 Adaptive Profiling

The results so far use all-the-time instrumentation at object uses. This section evaluates Sleigh's accuracy using adaptive profiling at object uses (Sec-

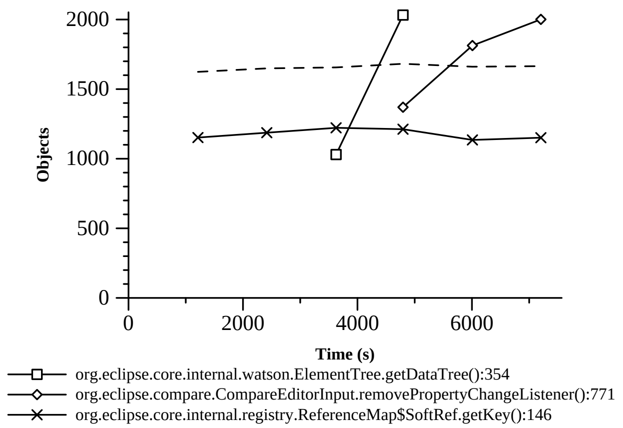


Figure 4.6: **Reported *last-use sites* for Eclipse when decoding processes *stale border objects* only.**

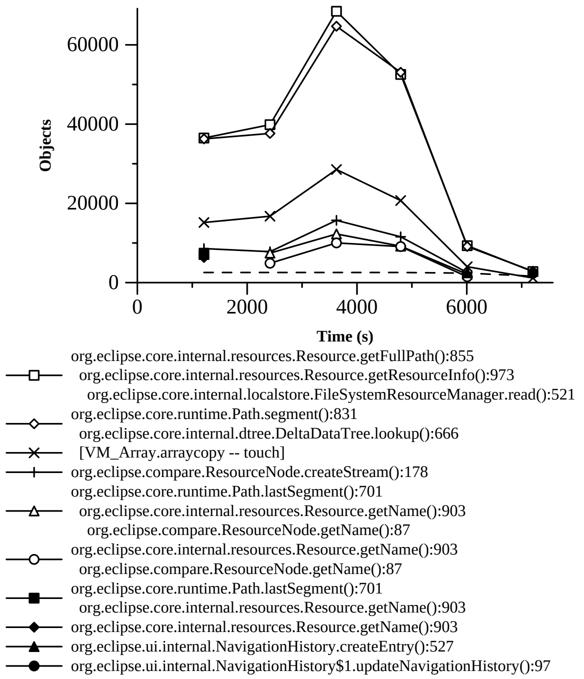


Figure 4.7: **Reported *last-use sites* for Eclipse when decoding processes *in-use border objects* only.**

tion 4.3.5). Adaptive profiling affects Sleigh's accuracy by (1) identifying some in-use objects as stale if it samples all the use sites of an in-use object at a too-low sampling rate and (2) reporting false positive or negative last-use sites if it samples a leaking last-use site at a too-low sampling rate. Tables 4.2 and 4.3 show results for adaptive profiling (lower three rows). Adaptive profiling causes Sleigh to identify more stale objects and to report more sites than all-the-time instrumentation. Figure 4.8 shows last-use sites for stale border objects from SPECjbb2000. This plot is noisier than Figure 4.5, which shows the same data collected using all-the-time instrumentation. However, the adaptive profiling graph shows the key leaking site, `removeOldestOrder()`, which appears in both graphs after about an hour and grows after that.

Sleigh with adaptive profiling *does* report the key leaking sites for SPECjbb2000 and Eclipse since these sites' execution rates are comparable with the rates they leak objects. We believe developers could fix the leaks using Sleigh's output from adaptive profiling.

4.4.5 Discussion

This section discusses Sleigh's benefits and drawbacks as a leak detection tool. Allocation and last-use sites help us find leaks, which agrees with Chilimbi and Hauswirth's experience that these sites are useful [CH04]. Last-use sites are particularly useful for pinpointing leaks, although allocation sites may be useful to developers, who understand their own code well. Limiting decoding to objects on the in-use/stale border is particularly useful for reporting sites directly involved in leaks.

At the same time, border objects may be few in number compared with all stale objects. For example, each structural diff performed in Eclipse yields

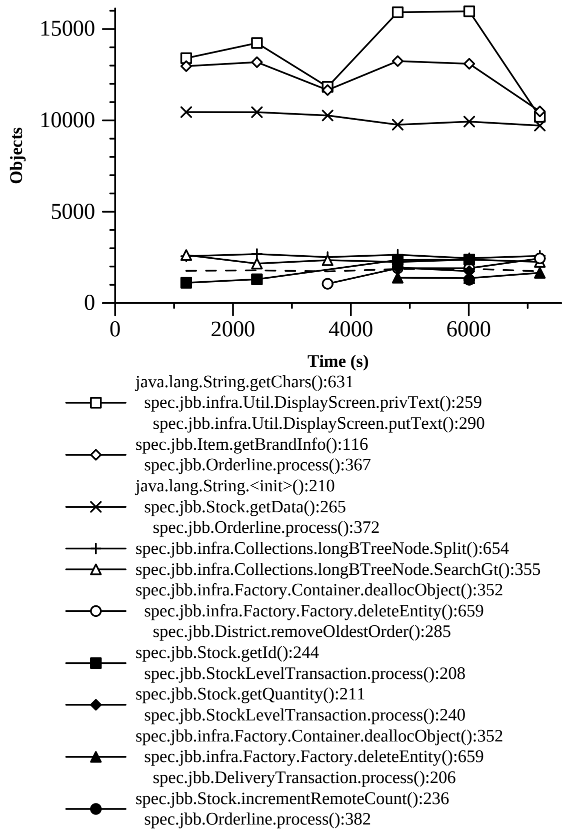


Figure 4.8: **Reported *last-use sites* for SPECjbb2000 when decoding processes *stale border objects* only, using *adaptive profiling*.**

one in-use border object and one stale border object—as well as a stale data structure whose size is dependent on the size of the diff. Bell needs hundreds or thousands of these objects to definitely report the leaking site (Section 4.1.2). By decoding all stale objects, Sleigh can generally report leaking sites for any nontrivial leak, but it is unclear if sites for non-border stale objects are useful in general. Thus, Sleigh may not be able to find some leaks in other programs, but we have not encountered such leaks (SPECjbb2000 and Eclipse are the only programs for which we have tried to find leaks due to time constraints and a lack of available long-running Java programs). While Sleigh may fail to find some leaks, it is unlikely to report erroneous leaks (false positives) since (1) its staleness approach precisely identifies memory not being used by the application, and (2) the false positive threshold m_{FP} (Section 4.1.2) avoids reporting incorrect sites for stale objects.

Another drawback of Sleigh's sites, and per-object sites in general, is that calling context is limited to the inlined portion, which may not be enough to understand the behavior of the code causing the leak. Eclipse in particular is a complex, highly object-oriented program with deep calling contexts. Unfortunately, efficiently maintaining and representing *dynamic* calling context is an unsolved problem.

4.5 Performance

This section evaluates Sleigh's space and time overheads.

4.5.1 Methodology

We execute each benchmark with a heap size fixed at two times the minimum possible for that benchmark. Because decoding is infrequent and not part of steady-state performance, we do not evaluate decoding's performance here (Section 4.4 evaluates decoding's performance).

4.5.2 Space Overhead

Sleigh uses four bits per object to maintain staleness and encode allocation and last-use sites (Section 4.3.1). It commandeers four available bits in the object header, so it effectively adds no per-object space overhead. Sleigh adds some space overhead to keep track of the mapping from sites to unique identifiers. The mapping's size is equal to the number of unique sites, which is proportional to program size. Sleigh could forego this mapping by using program counters (PCs) for sites (Jikes RVM supports obtaining source locations from the PC).

4.5.3 Compilation Overhead

Sleigh adds compilation overhead because it inserts instrumentation at object allocations and uses, increasing compilation load. Adaptive profiling duplicates code, so it also adds significant compilation overhead. We measure compilation overhead by extracting compilation time from the first run of replay compilation. Sleigh with all-the-time instrumentation and with adaptive profiling add 43% and 122% average compilation overhead, respectively, although an adaptive VM might respond to these increases by optimizing less code and by scaling back bloating optimizations such as inlining. Compilation overhead is not a primary concern because Sleigh targets long-running programs, for which compilation time represents a small fraction of execution time.

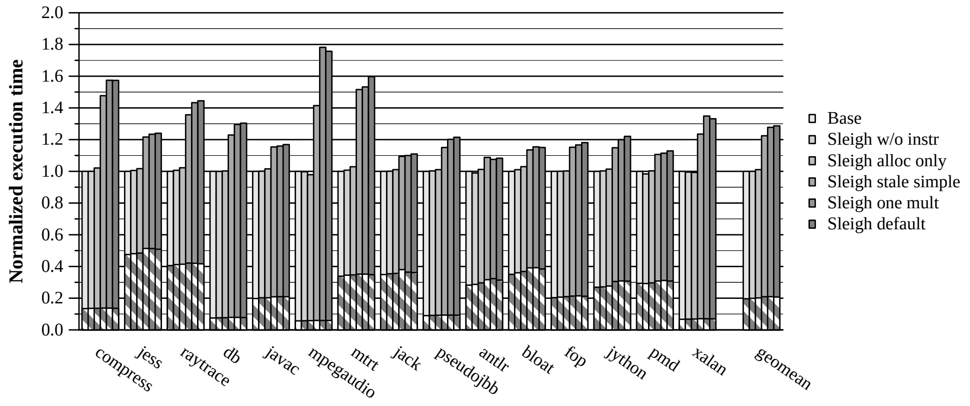


Figure 4.9: **Components of Sleigh runtime overhead.**

4.5.4 Time Overhead

Sleigh adds time overhead to maintain objects' stale counters and to encode objects' allocation and last-use site bits. Figure 4.9 presents the execution time overhead added by Sleigh. We use the second iteration of replay compilation, which measures only the application (not the compiler). Each bar is the minimum of five trials. We take the minimum because it represents the run least perturbed by external effects. The striped bars represent the portion of time spent in garbage collection (GC). *Base* is execution time without Sleigh; the bars are normalized to *Base*. The following configurations add Sleigh features monotonically:

- *Sleigh w/o instr* is execution time including updating stale counters during GC and marking VM objects at allocation time (Section 4.3.7) but without any instrumentation. This configuration adds no detectable overhead.

- *Sleigh alloc only* adds instrumentation at each allocation to initialize the stale counter and encode and set the allocation and last-use bits, incurring only 1% overhead on average.

- *Sleigh stale simple* adds simple instrumentation at object uses that resets the stale counter but does not encode the last-use site. This instrumentation occurs frequently and reads and writes the object header, and it adds 22% overhead over *Sleigh alloc only*.

- *Sleigh one mult* adds instrumentation that computes $f_{singleMult}$ (Section 4.1.3) at object uses and encodes the result in the object's last-use bit. This configuration adds just 5% over *Sleigh stale simple*, demonstrating that computing the encoding function itself is not a large source of overhead in Sleigh.

- *Sleigh default* uses the more robust $f_{doubleMult}$, which adds 1% over the single-multiply encoding function, for total average overhead of 29%.

Adaptive profiling. Sleigh uses adaptive profiling to lower its instrumentation overhead at object uses (Section 4.3.5). Figure 4.10 shows the overhead of Sleigh with adaptive profiling. *Base* and *Sleigh default* are the same as in Figure 4.9. *Sleigh AP min* is the execution overhead of Sleigh using adaptive profiling, but configured so control flow never enters the instrumented code. This configuration measures just the switching code, which adds 10% overhead. This overhead is higher than the 4% switching code overhead that Chilimbi and Hauswirth report [CH04], which is apparently a platform and implementation difference (e.g., C vs. Java). *Sleigh AP* is the overhead of Sleigh using fully functional adaptive profiling; it adds just 1% on average over *Sleigh AP min* since adaptive profiling executes instrumented code infrequently, for a total of 11% overhead.

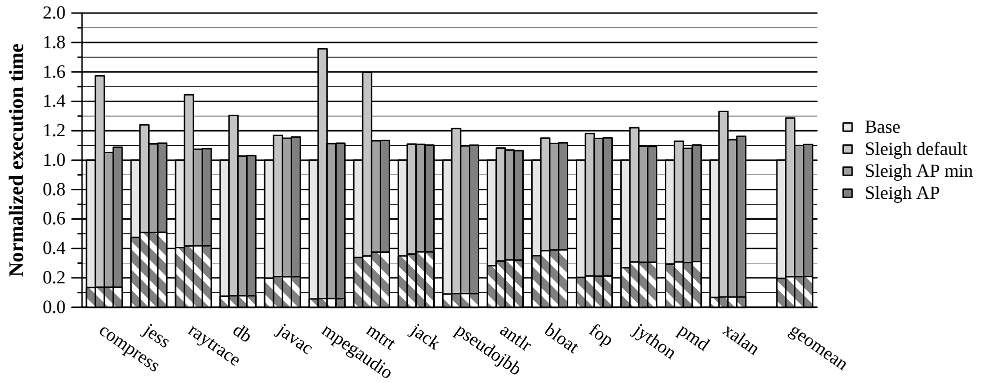


Figure 4.10: **Sleigh runtime overhead with adaptive profiling.**

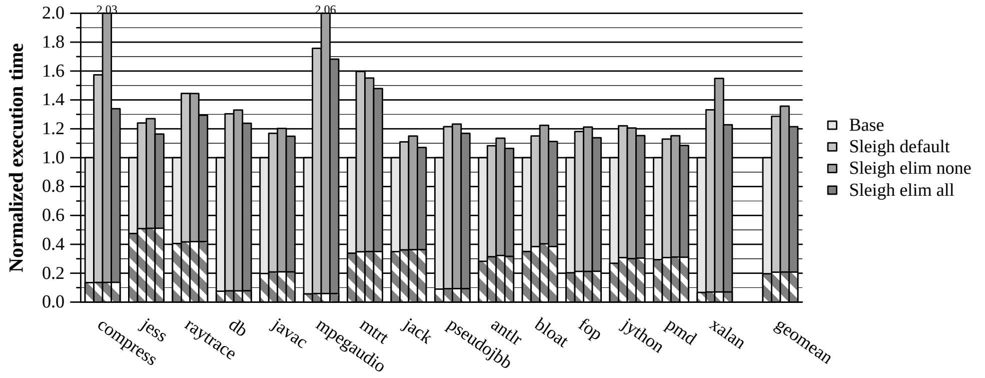


Figure 4.11: **Sleigh runtime overhead with and without redundant instrumentation optimizations.**

Redundant instrumentation. All Sleigh configurations presented so far remove fully redundant but not partially redundant instrumentation (Section 4.3.5). Figure 4.11 shows the overhead of Sleigh with various redundant instrumentation optimizations. *Base* and *Sleigh default* are the same as in

Figure 4.9. *Sleigh elim none* is execution time including both fully and partially redundant instrumentation (i.e., no redundant instrumentation removal). *Sleigh default* saves 7% of total execution time on average by removing fully redundant instrumentation. *Sleigh elim all* removes both fully and partially redundant instrumentation, providing an optimistic lower bound of 22% average overhead for redundant instrumentation removal.

4.6 Conclusion and Interpretation

Bell is a novel approach for encoding per-object information from a known, finite set in a single bit and decoding the information accurately and with statistical guarantees, given enough objects. We apply Bell to leak detection to store and report statistical per-object sites responsible for potentially leaked objects, using no space overhead. Drawbacks include high encoding and decoding time and a steep accuracy-time trade-off. Future work could decrease overhead with more efficient instrumentation and increase accuracy and decrease decoding time by considering dynamic type.

Similar to the other diagnosis approaches in this dissertation, Bell tracks a relationship between control and data flow. It maintains any number of program locations specific to all objects in the heap. We show Bell is directly useful for diagnosing leaks. It is applicable to other bugs that involve multiple erroneous objects, such as malformed data structures [CG06b], or that involve multiple running instances since it can reconstruct information from across the instances. From a research perspective, Bell demonstrates that it is possible to store relational information for all program data—but statistically to save space—and still be able to reconstruct information corresponding to a subset of the data.

Chapter 5

Tracking the Origins of Unusable Values

Finding the causes of bugs is hard, both during testing and after deployment. One reason is that a bug's effect is often far from its cause. Liblit et al. examined bug symptoms for various programs and found that inspecting the methods in a stack trace did not identify the method containing the error for 50% of the bugs [LNZ+05].

This chapter offers help for a class of bugs due to *unusable values* that either cause a failure directly or result in erroneous behavior. In managed languages such as Java and C#, the *null value* is unusable and causes a null pointer exception when dereferenced. In languages like C and C++, *undefined values*—those that are uninitialized, or derived from undefined values—are unusable, and their use can cause various problems such as silent data corruption, altered control flow, or a segmentation fault.

Failures due to unusable values are difficult to debug because (1) the origin of the unusable value may be far from the point of failure, having been propagated through assignments, operations, and parameter passing; and (2) unusable values themselves yield no useful debugging information. At best, the programmer sees a stack trace from the crash point that identifies the effect of the unused value, not its source. This problem is particularly bad for deployed software since the bug may be difficult to reproduce.

Null pointer exceptions are a well-known problem for Java programmers. Eric Allen writes the following on IBM developerWorks [All01]:

> Of all the exceptions a Java programmer might encounter, the null-pointer exception is among the most dreaded, and for good reason: it is one of the least informative exceptions that a program can signal. Unlike, for example, a class-cast exception, a null-pointer exception says nothing about what was expected instead of the null pointer. Furthermore, it says nothing about where in the code the null pointer was actually assigned. In many null-pointer exceptions, the true bug occurs where the variable is actually assigned to null. To find the bug, we have to trace back through the flow of control to find out where the variable was assigned and determine whether doing so was incorrect. This process can be particularly frustrating when the assignment occurs in a package other than the one in which the error was signaled.

Our goal is to provide this information automatically and at a very low cost.

Unused value errors are similarly difficult to debug for programs written in unmanaged languages such as C and C++. For example, Memcheck [SN05] is a memory checking tool built with the dynamic binary instrumentation framework Valgrind [NS07]. Memcheck can detect dangerous uses of undefined values, but prior to this work gave no origin information about those values. Requests for such origin information from Memcheck users were common enough that the FAQ explained the reason for this shortcoming [NS07].

The key question is: why does this variable contain an unusable value? We answer this question and solve this problem by introducing *origin tracking*. Origin tracking records program locations where unusable values are assigned, so they can be reported at the time of failure. We leverage the property that

unusable values are difficult to debug because they contain no useful information and store the origin information *in place of the unusable values themselves*, a form of *value piggybacking*. Value piggybacking requires no additional space, making origin tracking efficient. With some modifications to program execution, origin values flow freely through the program: they are copied, stored in the heap, or passed as parameters. They thus act like normal unusable values until the programs uses them inappropriately, whereupon we report the origin, which is often exactly what the programmer needs to diagnose the defect.

We present an approach and report implementation results for tracking the origins of null pointers in Java. Our results show that origin tracking is effective at reporting the origins of Java null pointer exceptions and adds minimal overhead to overall execution time (4% on average), making it suitable for deployed programs. We collected a test suite of 12 null pointer exceptions from publicly available Java programs with documented, reproducible bugs. Based on examining the stack trace, origin, source code, and bug report, we determine that origin tracking correctly reports the origin for all of the 12 bugs in this test suite, provides information not available from the stack trace for 8 bugs, and is useful for debugging in 7 of those 8 cases.

Tracking origins via value piggybacking has other applications in addition to null pointers in managed languages. As part of this research, our collaborator Nicholas Nethercote implemented origin tracking for undefined values in C and C++ programs executing in Valgrind's Memcheck tool [NS07, SN05]. Origin tracking adds negligible overhead to Memcheck, and identifies the origin of most 32-bit undefined values, but it cannot handle origins of smaller undefined values since they are too small to hold origins. Our conference paper describes origin tracking of undefined values in C and C++ in detail [BNK$^+$07].

Tracking origins is notable because previous value piggybacking applications conveyed only one or two bits of information per value [KLP88], whereas we show it can convey more useful information. Since our approach requires modest changes to the Java virtual machine and incurs very low overhead, commercial JVMs could rapidly deploy it to help programmers find and fix bugs.

5.1 Origin Tracking in Java

This section describes our implementation for tracking the origins of null references in Java programs. In each subsection, we first describe the general strategy used to piggyback information on null references, and then describe the details specific to our Jikes RVM implementation.

5.1.1 Supporting Nonzero Null References

Java virtual machines typically use the value zero to represent a null reference. This choice allows operations on null references, such as comparisons and detection of null pointer exceptions, to be simple and efficient. However, the Java VM specification does not require the use of a particular concrete value for encoding null [LY99b]. We modify the VM to instead represent null using a range of reserved addresses. Objects may not be allocated to this range, allowing null and object references to be differentiated easily.

Our implementation reserves addresses in the lowest 32nd of the address space: 0x00000000–0x07ffffff. That is, a reference is null if and only if its highest five bits are zero. The remaining 27 bits encode a program location as described in the next section.

As an alternative to a contiguous range of null values, null could be represented as any value with its lowest bit set. Object references and null could be differentiated easily since object references are word-aligned. VMs such as Jikes RVM that implement null pointer exceptions using hardware traps could instead use alignment traps. We did not implement this approach since unmodified Jikes RVM compiled for IA32 performs many unaligned accesses already (alignment checking is disabled by default on IA32).

5.1.2 Encoding Program Locations

The strategy described above provides 27 bits for encoding an origin in a null reference. We use one of these bits to distinguish between two cases: nulls that originate in a method body (the common case) and nulls that result from uninitialized static fields. In the latter case, the remaining 26 bits identify the particular field. In the former case, we encode the program location as a <method, line number> pair using one more bit to choose from among the following two layouts for the remaining 25 bits:

1. The default layout uses 13 bits for method ID and 12 bits for bytecode index, which is easily translated to a line number.
2. The alternate layout uses 8 bits for method ID and 17 bits for bytecode index, and is used only when the bytecode index does not fit in 12 bits. The few methods that fall into this category are assigned separate 8-bit identifiers.

We find these layouts handle all the programs in our test suite for origin tracking. Alternatively, one could assign a unique 27-bit value to each program location that assigns null via a lookup table. This approach would use space proportional to the size of the program to store the mapping. Our implementation adds no space since Jikes RVM already contains per-method IDs.

5.1.3 Redefining Program Operations

Our implementation redefines Java operations to accommodate representing null using a range of values.

Null assignment. Instead of assigning zero at null assignments, our modified VM assigns the 27-bit value corresponding to the current program location (method and line number). The dynamic compiler computes this value at compile time. Table 5.1, row (a) shows the null assignment in Java, its corresponding semantics for an unmodified VM, and semantics for a VM implementing origin tracking.

Object allocation. When a program allocates a new object, whether scalar or array, its reference slots are initialized to null by default. VMs implement this efficiently by allocating objects into mass-zeroed memory. Since origin tracking uses nonzero values to represent null, our modified VM adds code at object allocations that initializes each reference slot to the program location, as shown in Table 5.1, row (b). These values identify the allocation site as the origin of the null.

Since reference slot locations are known at allocation time, we can modify the compiler to optimize the code inserted at allocation sites. For hot sites (determined from profiles, which are collected by Jikes RVM and other VMs), the compiler inlines the code shown in the last column of Table 5.1,

	Java semantics	Standard VM	Origin tracking
(a) Assignment of null constant	`obj = null;`	`obj = 0;`	`obj = this_location;`

(b) Object allocation	`obj = new Object();`	... *allocate object* ...
		`foreach ref slot i`
		  `obj[i] = 0;`
		... *call constructor* ...
			... *allocate object* ...
			`foreach ref slot i`
			  `obj[i] = this_location;`
			... *call constructor* ...

(c) Null reference comparison	`b = (obj == null);`	`b = (obj == 0);`	`b = ((obj & 0xf8000000) == 0)`

(d) General reference comparison	`b = (obj1 == obj2);`	`b = (obj1 == obj2);`	`if (((obj1 & 0xf8000000) == 0))`
			  `b = ((obj2 & 0xf8000000) == 0);`
			`else`
			  `b = (obj1 == obj2);`

Table 5.1: **How origin tracking handles uses of null in Java code.** Column 2 shows example code involving null. Column 3 shows typical semantics for an unmodified VM. Column 4 shows semantics in a VM implementing origin tracking.

row (b). If the number of slots is a small, known constant (true for all small scalars, as well as small arrays where the size is known at compile time), the compiler flattens the loop.

Static fields are also initialized to null, but during class initialization. We modify the VM's class initialization to fill each static reference field with a value representing the static field (the VM's internal ID for the field). Of the 12 bugs we evaluate in Section 5.2, one manifests as a null assigned at class initialization time.

Null comparison. To implement checking whether a reference is null, VMs compare the reference to zero. Origin tracking requires a more complex comparison since null may have any value from a range. Our implementation uses the range 0x00000000–0x07ffffff for null, and it implements the null test using a bitwise AND with 0xf8000000, as shown in Table 5.1, row (c).

General reference comparison. A more complex case is when a program compares two references. With origin tracking, two references may have different underlying values even though both represent null. To handle this case, the origin tracking implementation uses the following test: two references are the same if and only if (1) they are both nonzero null values or (2) their values are the same. Table 5.1, row (d) shows the modified VM implementation.

We optimize this test for the common case: non-null references. The instrumentation first tests if the first reference is null; if so, it jumps to an out-of-line basic block that checks the second reference. Otherwise, the common case performs a simple check for reference equality, which is sufficient since both references are now known to be non-null.

5.1.4 Implementation

Our implementation in Jikes RVM stores program locations instead of zero (Table 5.1, rows (a) and (b)) only in application code, not in VM code or in the Java standard libraries. This choice reflects developers' overriding interest in source locations in their application code. Since the VM is not implemented entirely in pure Java—it needs C-style memory access for low-level runtime features and garbage collection—generating nonzero values for null in the VM and libraries would require more pervasive changes to the VM because it assumes null is zero. Since null references generated by the application sometimes make their way into the VM and libraries, our implementation modifies all reference comparisons to handle nonzero null references (Table 5.1, rows (c) and (d)) in the application, libraries, and VM.

Some VMs, including Jikes RVM, catch null pointer exceptions using a hardware trap handler: since low memory is protected, dereferencing a null pointer generates the signal SIGSEGV. The VM's custom hardware trap handler detects this signal and returns control to the VM, which throws a null pointer exception. The origin tracking implementation protects the address range 0x00000000–0x07ffffff so null dereferences will result in a trap. For origin tracking, we modify the trap handler to identify and record the culprit base address, which is the value of the null reference. When control is returned to the VM, it decodes the value into a program location and reports it together with the null pointer exception. VMs that use explicit null checks to detect null pointer exceptions could simply use modified null checks as described in Section 5.1.3.

The Java Native Interface (JNI) allows unmanaged languages such as C and C++ to access Java objects. Unmanaged code assumes that null is zero. Our modified VM uses zero for null parameters passed to JNI methods. This approach loses origin information for these parameters but ensures correct execution.

5.2 Finding and Fixing Bugs

This section describes a case study using origin tracking to identify the causes of 12 failures in eight programs. These results are summarized in Table 5.2, which contains the lines of code measured with the Unix wc command; whether the origin was identified; whether the origin was identifiable trivially; and how useful we found the origin report (these criteria are explained in detail later in this section). We describe three of the most interesting cases (Cases 1, 2, and 3) in detail below. The other nine are in our conference paper [BNK$^+$07]. In summary, our experience shows:

- The usefulness of origin information depends heavily on the complexity of the underlying defect. In some cases, it is critical for diagnosing a bug. Given the extremely low cost of origin tracking (see Section 5.3), there is little reason not to provide this extra information, which speeds debugging even when a defect is relatively trivial.
- Bug reports often do not contain sufficient information for developers to diagnose or reproduce a bug. Origin tracking provides extra information for users to put in bug reports to help developers without access to the deployment environment.
- It is not always clear whether the defect lies in the code producing the null value, or in the code dereferencing it (e.g., the dereferencing code

Case	Program	Lines	Exception description	Origin?	Trivial?	Useful?
1	Mckoi SQL DB	94,681	Access closed connection	Yes	Nontrivial	Definitely
2	FreeMarker	64,442	JUnit test crashes unexpectedly	Yes	Nontrivial	Definitely
3	JFreeChart	223,869	Plot without x-axis	Yes	Nontrivial	Definitely
4	JRefactory	231,338	Invalid class name	Yes	Nontrivial	Definitely
5	Eclipse	2,425,709	Malformed XML document	Yes	Nontrivial	Most likely
6	Checkstyle	47,871	Empty default case	Yes	Nontrivial	Most likely
7	JODE	44,937	Exception decompiling class	Yes	Nontrivial	Most likely
8	Jython	144,739	Use built-in class as variable	Yes	Nontrivial	Potentially
9	JFreeChart	223,869	Stacked XY plot with lines	Yes	SWNT	Marginally
10	Jython	144,739	Problem accessing *__doc__* attribute	Yes	SWNT	Marginally
11	JRefactory	231,338	Package and import on same line	Yes	Trivial	No
12	Eclipse	2,425,709	Close Eclipse while deleting project	Yes	Trivial	No

Table 5.2: **The diagnostic utility of origins returned by origin tracking in Java.** Cases 1, 2, and 3 are described in detail in Section 5.2. *SWNT* means "somewhat nontrivial." Bug repositories are on SourceForge [Sou] except for Eclipse [Eclb] and Mckoi SQL Database [Mck].

should add a null check). A stack trace alone only provides information about the dereferencing code. Origin tracking allows programmers to consider both options when formulating a bug fix.

- Null pointer exceptions often involve a null value flowing between different software components, such as application code and library code. Therefore, even when the origin and dereference occur close together it can be difficult to evaluate the failure without a full understanding of both components. For example, a programmer might trigger a null pointer exception in a library method by passing it an object with a field that is unexpectedly null. Origin tracking indicates which null store in the application code is responsible, without requiring extra knowledge or source code for the library.

5.2.1 Evaluation Criteria

For each error, we evaluate how well origin tracking performs using three criteria:

Origin identification. Does origin tracking correctly return the method and line number that assigned the null responsible for the exception?

Triviality. Is the stack trace alone, along with the source code, sufficient to identify the origin? In 8 of 12 null pointer exceptions, the origin is not easy to find via inspection of the source location identified by the stack trace.

Usefulness. Does knowing the origin help with understanding and fixing the defect? Although we are not the developers of these programs, we examined the source code and also looked at bug fixes when available. We believe that the origin report is not useful for one-third of the cases; probably useful for another third; and definitely useful for the remaining third of the cases.

5.2.2 Origin Tracking Case Studies

We now describe three bugs that highlight origin tracking's role in discovering the program defect.

Case 1: Mckoi SQL Database: Access Closed Connection

The first case highlights an important benefit of origin tracking: it identifies a null store far away from the point of failure (possibly in another thread). The location of the store indicates a potential cause of the failure.

The bug report comes from a user's message on the mailing list for Mckoi SQL Database version 0.93, a database management system for Java (Message 02079). The user reports that the database throws a null pointer exception when the user's code attempts to execute a query. The bug report contains only the statement `dbStatement.executeUpdate(dbQuery);` and a stack trace, so we use information from the developer's responses to construct a test case. Our code artificially induces the failure but captures the essence of the problem.

Figure 5.1(a) shows the stack trace. This information presents two problems for the application developer. First, the failure is in the library code, so it cannot be easily debugged. Second, it indicates simply that the query failed, with no error message or exception indicating why.

Our origin information, shown in Figure 5.1(b), reveals the reason

```
java.lang.NullPointerException:
  at com.mckoi.database.jdbcserver.JDBCDatabaseInterface.execQuery():213
  at com.mckoi.database.jdbc.MConnection.executeQuery():348
  at com.mckoi.database.jdbc.MStatement.executeQuery():110
  at com.mckoi.database.jdbc.MStatement.executeQuery():127
  at Test.main():48
```

(a)

```
Origin:
 com.mckoi.database.jdbcserver.AbstractJDBCDatabaseInterface.internalDispose():298
```

(b)

Figure 5.1: **Case 1: VM output for Mckoi SQL Database bug.** (a) The stack trace shows the query failed inside the library. (b) Origin tracking suggests that the failure is due to a closed connection.

for the failure: a null store occurred in AbstractJDBCDatabaseInterface.internalDispose() at line 298. This method is part of closing a connection; line 298 assigns null to the connection object reference. The cause of the failure is that the query attempts to use a connection that has already been closed.

The origin information may be useful to both the application user and the database library developers. Users can probably guess from the name of the method AbstractJDBCDatabaseInterface.internalDispose() that the problem is a closed connection, and can plan for this possibility in their application logic. The developers can also modify the execQuery() method to check for a closed connection and to throw a useful SQLException that reports the reason, as noted in an existing response on the mailing list.

Case 2: FreeMarker: JUnit Test Crashes Unexpectedly

The second case illustrates how origin tracking helps diagnose errors when the null reference passes from variable to variable by assignment. The

```
java.lang.NullPointerException:
  at freemarker.template.WrappingTemplateModel.wrap():131
  at freemarker.template.SimpleHash.get():197
  at freemarker.core.Environment.getVariable():959
  at freemarker.core.Identifier._getAsTemplateModel():70
  at freemarker.core.Expression.getAsTemplateModel():89
  ...
  at junit.textui.TestRunner.main():138
```

(a)

```
Origin: freemarker.template.DefaultObjectWrapper.instance
```

(b)

Figure 5.2: **Case 2: VM output for FreeMarker bug.** (a) The stack trace shows the fault location. (b) The null's origin is an uninitialized static field.

case is also interesting because at first glance the initial assignment appears to be non-null, but it is in fact null because of a static initialization ordering issue.

FreeMarker 2.3.4 is a Java library that generates output such as HTML and source code using user-defined templates. We reproduced an exception in the library using test code posted by a user (Bug 1354173). Figure 5.2 shows the exception stack trace.

The exception occurs at line 131 of wrap(), which tries to dereference defaultObjectWrapper, which is null. Previously, defaultObjectWrapper was assigned the value of the static, final field DefaultObjectWrapper.instance. At first glance, it appears that DefaultObjectWrapper.instance is properly initialized:

```
static final DefaultObjectWrapper instance =
  new DefaultObjectWrapper();
```

However, due to a circular initialization dependency between WrappingTemplate-

Model and DefaultObjectWrapper, instance is in fact initialized to null. Origin tracking helps diagnose this error by reporting the uninitialized static field instance as the origin of the offending null. The origin is quite useful for diagnosing the bug since (1) the null passes through a variable, and (2) it is not intuitive that the original assignment assigns null. A responder to the bug report also came to the conclusion that the exception is a result of static class initialization ordering, but to our knowledge it has not been fixed in any version of FreeMarker.

Case 3: JFreeChart: Plot Without X-Axis

This case involves a small test program provided by a bug reporter that causes a null pointer exception inside JFreeChart 1.0.2, a graphing library (Bug 1593150). This case, like the first case in this section, represents an important class of failures for which origin tracking is useful: the failure is induced by the application, but since it occurs inside the library the programmer has no easy way to interpret the stack trace or to debug the library code.

The following is code provided by the user, annotated with line numbers:

```
12:   float[][] data = {{1.0f,2.0f},{3.0f,4.0f}};
13:   FastScatterPlot plot = new FastScatterPlot(data, null, null);
14:   Button aButton = new Button();
15:   Graphics2D graphics = (Graphics2D)(aButton.getGraphics());
16:   plot.draw(graphics, new Rectangle2D.Float(),
                new Point2D.Float(), null, null);
```

Figure 5.3(a) shows the exception stack trace. The method FastScatterPlot.-draw(), called from line 16 of the user code, throws a null pointer exception.

```
java.lang.NullPointerException:
  at org.jfree.chart.plot.FastScatterPlot.draw():447
  at Bug2.test():16
  at Bug2.main():9
```

(a)

```
Origin: Bug2.test():13
```

(b)

Figure 5.3: **Case 3: VM output for JFreeChart bug.** (a) The stack trace shows a failure inside the library. (b) Origin tracking identifies the error as caused by a null from user code.

This stack trace is not very helpful to the library user, who may not have access to or be familiar with the JFreeChart source code.

On the other hand, origin tracking provides information that is directly useful to the user: the origin is line 13 of the user's test() (Figure 5.3). The user can quickly understand that the exception occurs because the code passes null as the x-axis parameter to the FastScatterPlot constructor.

While the origin allows a frustrated user to modify his or her code immediately, it also suggests a better long-term fix: for JFreeChart to return a helpful error message. The developers diagnosed this bug separate from us, and their solution, implemented in version 1.0.3, causes the constructor to fail with an error message if the x-axis parameter is null.

5.3 Performance Summary

This section summarizes the performance impact of tracking origins of null references. Using adaptive compilation, we find that origin tracking adds 4% to overall execution time. Replay compilation shows that about 3% comes from instrumentation in the application and 1% comes from compilation overhead. The conference paper contains a detailed performance evaluation [BNK+07].

5.4 Conclusion and Interpretation

Developers need all the debugging help they can get. Our lightweight approach reports errors due to unusable values by storing program locations *in place* of the values. Its minimal footprint and diagnosis benefits make it suitable for commercial VMs.

The diagnosis approaches in this dissertation generally track control and data flow and the relationship between them. This chapter presented an approach for efficiently storing information about control flow through arbitrary data flow for a class of values that are particularly difficult to debug. With modest effort, commercial VMs could implement origin tracking and use it during deployment. There it would provide immediate benefits by enhancing user error reports, which often contain only a stack trace. Azul is considering adding origin tracking to its production JVM [Cli08]. More broadly, origin tracking's success motivates developers to build other debugging features intended for the deployed setting.

Chapter 6

Tolerating Memory Leaks with Melt

Chapter 4 applied Bell to the problem of detecting leaks in order to save space. As part of the Bell work, we manually diagnosed and fixed a memory leak in Eclipse, but several months passed before developers applied the fix and released a new version [Ecla]. While waiting for developers to find and fix a leak, users have to deal with the effects of the leak. Performance suffers as garbage collection workload and frequency grows and as application locality decreases. Eventually the program grinds to a halt if it outgrows physical memory, or it crashes with an out-of-memory error when it runs out of virtual memory or reaches the maximum heap size.

While deployed software is full of many types of bugs, we expect memory leaks to be particularly difficult to find and fix in the testing stage. They are environment sensitive and may take days or weeks or longer to manifest. Since they have no immediate symptoms but instead eventually result in slowdowns and crashes at arbitrary program points, they are especially hard to reproduce, find, and fix [HJ92]. Leaks may occur unexpectedly, and they are thus excellent candidates for automatic approaches that tolerate leaks by eliminating performance degradations and out-of-memory errors. Although it is undecidable in general whether an object is *dead* (will not be used again), we can estimate *likely leaks* due to dead objects by identifying objects that the program has not used in a while.

This chapter presents a new leak tolerance approach called *Melt* that transfers likely leaked objects to disk. By offloading leaks to disk and freeing up physical and virtual memory, Melt significantly delays memory exhaustion since disks are typically orders of magnitude larger than main memory. Melt is analogous to virtual memory paging since both conserve memory by transferring stale memory to disk. However, standard paging is insufficient for managed languages since (1) pages that mix in-use and leaked objects cannot be paged to disk, and (2) garbage collection thrashes since its working set is all objects. Melt effectively provides *fine-grained* paging by using object instead of page granularity and by restricting the collector accessing only objects in memory. Determining whether an object is live (will be used again) is undecidable in general, so Melt predicts that *stale* objects (objects the program has not used in a while) are likely leaks and moves them to disk. Melt maintains program semantics: if the application tries to access an object on disk, Melt *activates* it by moving it from disk back to main memory.

Melt keeps programs performing well by guaranteeing time and space remain proportional to *in-use* (non-leaked) memory. It restricts the application and garbage collector accesses to in-use objects in memory. It prohibits accesses to stale objects on disk and keeps metadata proportional to in-use memory. Other leak tolerance approaches for garbage-collected languages do not provide this guarantee [BGH$^+$07, GSW07, TGQ08]. *Bookmarking collection* is similar to Melt in that it restricts collection to in-use pages, but it operates at page granularity [HFB05], whereas Melt seeks to tolerate leaks with fine-grained object tracking and reorganization.

We implement Melt in a copying generational collector, which demonstrates Melt's support for collectors that move objects. Our design works with any tracing copying or non-copying collector. Our results show that Melt generally adds overhead only when the program is close to running out of memory. For simplicity, our implementation inserts instrumentation into the application that helps identify stale objects, adding on average 6% overhead, but a future implementation would insert instrumentation only in response to memory pressure. We apply Melt to 10 leaks, including 2 leaks in Eclipse and a leak in a MySQL database client application. Melt successfully tolerates five of these leaks: throughput does not degrade over time, and the programs run until they exhaust disk space or exceed a 24-hour limit. It helps two other leaks but adds high overhead activating objects that are temporarily stale but not dead. Of the other three, two exhibit *in-use* growth since most of the heap growth is memory that is inadvertently in-use: the application continues to access objects it is not using. Melt cannot tolerate the third leak because of a shortcoming in the implementation.

As a whole, our results indicate that Melt is a viable approach for increasing program reliability with low overhead while preserving sematnics, and it is a compelling feature for production VMs.

6.1 How Melt Identifies and Tolerates Leaks

Melt's primary objective is to give the illusion there is no leak: performance does not degrade as the leak grows, the program does not crash, and it runs correctly. To achieve this objective, Melt meets the following design goals:

1. Time and space overheads are proportional to the in-use memory, not leaked memory.

2. Melt preserves semantics by maintaining and, if needed, activating stale objects.

Furthermore, Melt adheres to the following *invariants*:

- *Stale* objects are isolated from in-use objects in a separate *stale space*, which resides on disk.
- The collector never accesses objects in the stale space, except when moving objects to the stale space.
- The application never accesses objects in the stale space, except when activating objects from the stale space.

We satisfy these invariants as follows: (1) Melt identifies stale objects (Section 6.1.1); (2) it segregates stale objects from in-use objects by moving stale objects to disk, and it uses double indirection for references from stale to in-use objects (Section 6.1.2); and (3) it intercepts program attempts to access objects in stale space and immediately moves the object into in-use space. (Section 6.1.3). Section 6.1.4 presents how Melt decides when and which stale objects to move to disk, based on how close the program is to running out of memory.

6.1.1 Identifying Stale Objects

We classify reachable objects that the program has not referenced in a while as *stale*. If the program never accesses them again, they are true leaks. As we show later, some leaks manifest as in-use (live) objects. For example, the program forgets to delete objects from a hash table, keeps adding objects,

and then rehashes all elements every time it grows beyond the current limit. Staleness thus under-approximates leaks of in-use objects.

To identify stale objects, Melt modifies both the garbage collector and the dynamic compiler. At a high level, the modified collector *marks* objects as stale on each collection, and the modified compiler adds instrumentation to the application to *unmark* objects at each use. At each collection, objects the program has *not* accessed since the last collection will still be marked stale, while accessed objects will be unmarked. For efficiency, the collector actually marks both references and objects as stale. It marks *references* by setting the lowest (least significant) bit of the pointer. The lowest bit is available for marking since object references are word-aligned in most VMs. In Melt, the collector marks *objects* as stale by setting a bit in the object header.

The compiler adds instrumentation called a *read barrier* [BH04] to every load of an object reference. The barrier checks whether the reference is stale. If it is stale, the barrier unmarks the referenced object and the reference. The following pseudocode shows the barrier:

```
b = a.f;                       // Application code
if (b & 0x1) {                 // Conditional barrier
  t = b;                       // Backup ref
  b &= ~0x1;                   // Unmark ref
  a.f = b; [iff a.f == t]      // Atomic update
  b.staleHeaderBit = 0x0;      // Unmark object
}
```

This *conditional* barrier reduces overhead since it performs stores only the first time the application loads each reference in each mutator epoch.[1] Checking

[1]The mutator is the application alone, not the collector. A mutator epoch is the period between two collections.

for a marked *reference*, rather than a marked *object*, reduces overhead since it avoids an extra memory load.

For thread safety, the barrier updates the accessed slot atomically with respect to the read. Otherwise another thread's write to a.f may be lost. The pseudocode [iff a.f == t] indicates the store is dependent on a.f being unchanged. We implement the atomic update using a compare-and-swap (CAS) instruction that succeeds only if a.f still contains the original value of b. If the atomic update fails, the read barrier simply continues; it is semantically correct to proceed with (unmarked) b while a.f holds the update from the other thread. Similarly, clearing the stale header bit (b.staleHeaderBit = 0x0) must be atomic if another thread can update other bits in the header. In our implementation, these atomic updates add negligible overhead since the body of the conditional barrier executes infrequently.

At the next collection, each object will be marked stale if and only if the application did not load a reference to it since the previous collection. Figure 6.1 shows an example heap with stale (shaded gray) objects C and D. They have not been accessed since the last collection, because all their incoming references are stale, marked with S. Although B has incoming stale references, B is in-use because the reference A → B is in-use.

6.1.2 The Stale Space

When the garbage collection traces the heap, it moves stale objects to the *stale space*, which resides on disk. For example, the collector moves stale objects C and D from Figure 6.1 to the stale space, as illustrated by Figure 6.2.

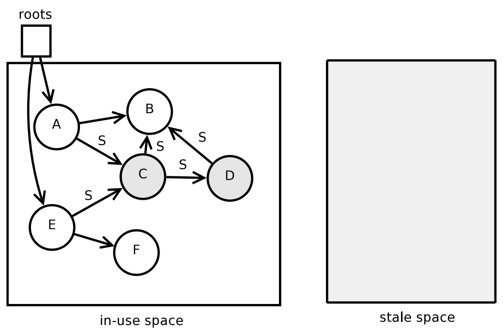


Figure 6.1: **Stale Objects and References**

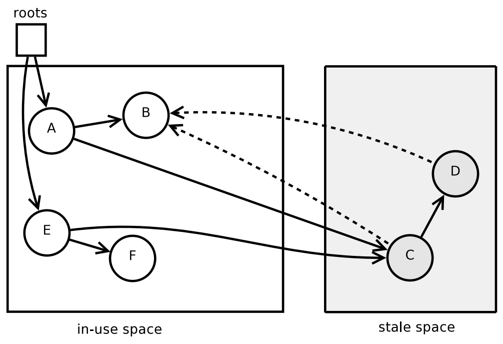


Figure 6.2: **Segregation of In-Use and Stale Objects**

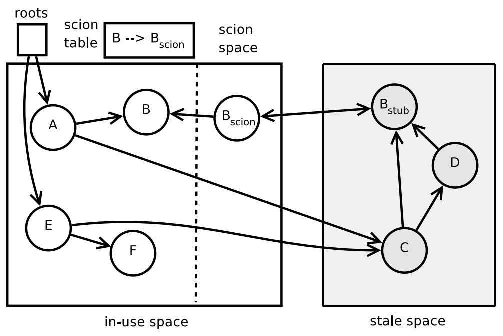


Figure 6.3: **Stub-Scion Pairs**

Stub-Scion Pairs

References from stale objects to in-use objects are problematic because moving collectors such as copying and compacting collectors move in-use objects. For example, consider moving B, which has references from C and D. If B moves, we do not want to touch stale objects to update their outgoing references, which would violate the invariants. Solutions such as remembered sets and card marking [JL96] will not work because either space is proportional to the number of references, or GC must access the source reference slot.

We solve this problem by using *stub-scion pairs*, borrowed from distributed garbage collection [Pla94]. Stub-scion pairs provide two levels of indirection. Melt creates a *stub* object in the stale space and a *scion* object in the in-use space for each in-use object that is referenced by one or more stale object(s). The collector avoids touching stubs and stale objects by referencing and updating the scion. The stub has a single field that points to the scion.

The scion has two fields: one points to the in-use object and the other points back to its stub. We modify references in the stale space that refer to an in-use object to refer instead to the stub. Figure 6.3 shows B_{stub} and B_{scion} providing two levels of indirection for references from C and D to B. Scions may not move. The collector treats scions as roots, retaining in-use objects referenced by stale objects. If the collector moves an object referenced by a scion, it updates the scion to point to the moved object.

To ensure each in-use object has only one stub-scion pair, we use a *scion lookup table* that maps from an in-use object to its scion, if it has one. This data structure is proportional to the number of scions, which is proportional to the number of in-use objects in the worst case, but is usually much smaller in practice. The collector processes the scions at the beginning of collection.

Returning to Figure 6.2, when the collector copies C to the stale space, B initially has no entry in the scion lookup table, so Melt adds a mapping $B \rightarrow B_{scion}$ to the table when it creates B_{stub} and B_{scion}. Next, when it copies D to the stale space, it finds the mapping $B \rightarrow B_{scion}$ in the table and re-uses the existing stub-scion pair. The resulting system snapshot is shown in Figure 6.3.

It may seem at first that we need scions but not necessarily stubs, i.e., stale objects could point directly to the scion. However, we need both because an in-use object referenced by a scion may become stale later. For example, consider the case of B becoming stale on the next garbage collection in Figure 6.3. In order to eliminate the scion without a stub (to avoid using in-use memory for stale-to-stale references), we would need to find all the stale pointers to the scion, which violates the stale space invariant to never visit stale objects after instantiation. Instead, Melt copies B to the stale space, looks up the stub location in the scion, and points the stub to stale B. Note that Melt accesses the disk both to modify the stub and to move the new stale object. These accesses do not violate invariants since they are part of moving an object to the stale space. Melt then deletes the scion and removes the entry in the scion lookup table. Figure 6.4 shows the result.

6.1.3 Activating Stale Objects

Melt prevents the application from directly accessing the stale space since (1) these accesses would violate the invariant that the stale space is not part of the application's working set, and (2) object references in the stale space may refer to stubs and scions. Melt intercepts application access to stale objects by modifying the read barrier to check for references to the stale space:

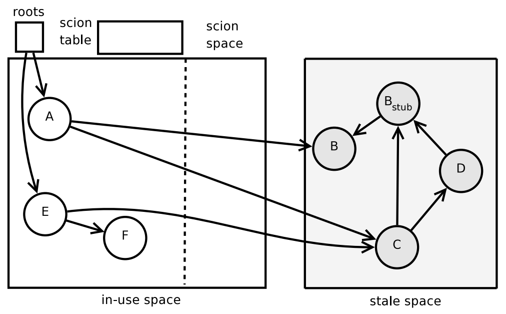


Figure 6.4: **Scion-Referenced Object Becomes Stale**

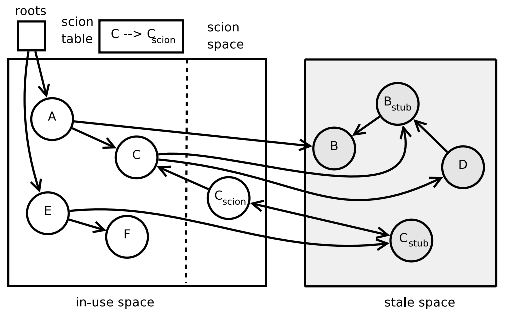


Figure 6.5: **Stale Object Activation**

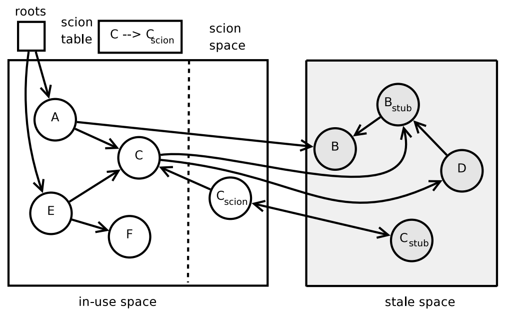


Figure 6.6: **Reference Updates Following Activation**

```
b = a.f;        // Application code
if (b & 0x1) { // Read barrier
  t = b;
  b &= ~0x1;
  // Check if in stale space
  if (inStaleSpace(b)) {
    b = activateStaleObject(b);
  }
  a.f = b; [iff a.f == t]
  b.staleHeaderBit = 0x0;
}
```

The VM method activateStaleObject() copies the stale object to the in-use space. Since other references may still point to the stale version, activateStaleObject() creates a stub-scion pair for the activated object as follows: (1) it converts the stale space object version into a stub, and (2) it creates a scion and points the stub at the scion. The scion points to the activated object. The store to a.f must be atomic with respect to the original value of b, i.e., [iff a.f == t].

Consider activating C from Figure 6.4. First, activateStaleObject() copies C to the in-use space. Then it replaces stale C with a stub, allocates a scion, and links them all together, as shown in Figure 6.5. Note that C retains its references to D and B_{stub}, and E retains its reference to the old version of C, which is now C_{stub}.

If the application later follows a different reference to the previously stale object in the stale space, activateStaleObject() finds the stub in the object's place, which it follows to the scion, which in turn points to the activated object. The first access of such a reference will update the reference to point to the activated version and any subsequent accesses will go directly to the in-use object. For example, if the application accesses a reference from E to C_{stub} in

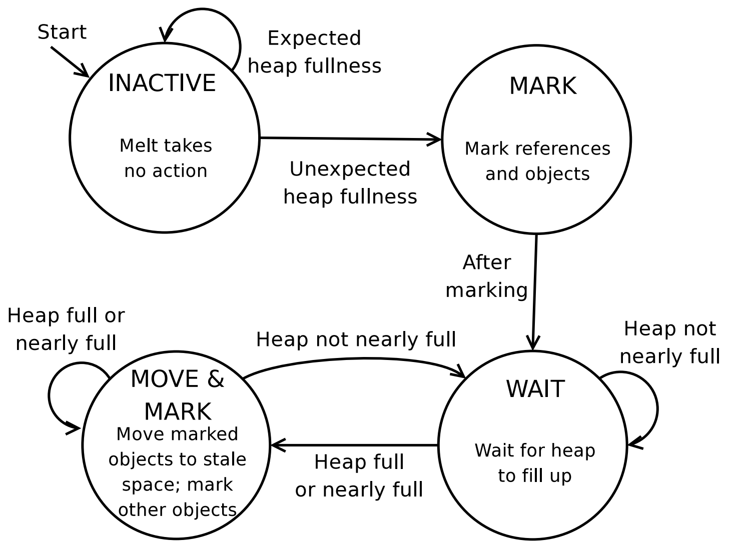


Figure 6.7: **State diagram for when Melt marks objects as stale and moves objects to the stale space.**

Figure 6.5, activateStaleObject() follows C_{stub} to C_{scion} to C in the in-use space and updates the reference, as shown in Figure 6.6.

6.1.4 When to Move Objects to the Stale Space

Melt can mark objects as stale and/or move objects to the stale space on any full-heap garbage collection. However, it does not make sense to incur this overhead if the application is not leaking memory. Furthermore, Melt could potentially fill the disk for a non-leaking application, producing an error where none existed. Thus, Melt decides whether to mark and move based on *how full the heap is* as shown in Figure 6.7.

Initially Melt is INACTIVE: it does not mark or move objects. It also does not need read barriers if the VM supports adding them later via recompilation or code patching (we did not implement this feature). The heap *fullness* is the ratio of reachable memory to maximum heap size at the end of a full-heap collection. Since users typically run applications in heaps at least twice the minimum needed to keep GC overhead low, by default we use 50% fullness as the "unexpected" heap fullness. If the heap fullness exceeds this expected amount, Melt moves to the MARK state, where the GC marks all objects and references during the next full-heap GC.

After GC marks all objects and references, Melt enters the WAIT state. It remains in the WAIT state until the program is close to memory exhaustion; then it enters the MOVE & MARK state. By default this threshold is 80% heap fullness. Users could specify 100% heap fullness, which would wait until complete heap exhaustion before using the stale space. However, coming close to running out of memory brings the application to a virtual halt because garbage collection becomes extremely frequent. In MOVE & MARK, Melt moves all objects still marked to the stale space. It marks all objects that remain in the in-use space, so they can be moved to the stale space later if still marked. If the heap is still nearly full (e.g., for fast-growing leaks), Melt remains in MOVE & MARK for another full-heap GC. Otherwise, it returns to WAIT until the heap fills again, and then it returns to MOVE & MARK, and so on. Melt could return to INACTIVE if memory usage decreased to expected levels (not shown or implemented).

6.2 Implementation Details

This section presents details specific to our implementation in Jikes RVM. Our approach is suitable for garbage-collected, type-safe languages using tracing garbage collectors.

6.2.1 VM Issues

Identifying stale objects. To identify stale objects, Melt modifies (1) the compilers to add read barriers to the application and (2) the collector to mark heap references and objects stale. For efficiency and simplicity, we exclude VM objects and objects directly pointed to by roots (registers, stacks, and statics) as candidates for the stale space.

Moving large objects. Like most VM memory managers, MMTk allocates large objects (8 KB or larger) into a special non-moving *large object space* (LOS). Since we need to copy large objects to disk, we modify the LOS to handle copying. During collection, when Melt first encounters a stale large object, Melt moves it to the stale space, updates the reference, and installs a forwarding pointer that corrects any other references to this object. At the end of the collection, it reclaims the space for any large objects it moves. Activation works in the same way as for other object sizes.

Activating stale objects. When a read barrier intercepts an application read to the stale space, Melt immediately copies the object to an in-use space and updates the reference. Since activation allocates into the in-use part of the heap, it may trigger a garbage collection (GC). Application reads are not necessarily *GC-safe points*. GC-safe points require the VM to enumerate all the pointers into the heap, i.e., to produce a stack map of the local, global, and temporary variables in registers. If an activation triggers a GC, Melt defers collection by requesting an *asynchronous collection*, which causes collection to occur at the next GC-safe point. The VM may thus exceed its page budget. The number of operations until the next GC-safe point is small and bounded

since these points are frequent and occur and occur at allocations, method entries, method exits, and loop back edges in Jikes RVM and many other VMs.

6.2.2 Stale Space on Disk

64-bit on-disk addressing. Melt uses an on-disk stale space with 64-bit addressing, even though memory is 32-bit addressed. When it moves a stale object to disk, it uses a 64-bit address and expands the object's reference slots to 64 bits. Similarly, it uses 64-bit stubs. Most stale objects refer to other stale objects. For stale objects referenced by in-use objects, we use a level of indirection to handle the translation from 32- to 64-bit addresses. These *mapping stubs* reside in memory, but reference 64-bit on-disk objects. The GC traces mapping stubs, which reside in in-use memory, and collects unreachable mapping stubs. The number of mapping stubs is bounded by the number of references from in-use to stale memory, which is small in practice, and is at worst proportional to in-use memory.

Figures 6.8 and 6.9 show the 64-bit on-disk stale space representation for Figures 6.5 and 6.6. The main difference is the mapping stub space, which provides indirection for references from the in-use space to the stale space. Three types of references are 64 bits: mapping stubs, references in stale objects, and pointers from scions to their stubs. If a stale object references an in-memory object, e.g., $\mathsf{C}_{\mathsf{stub}} \rightarrow \mathsf{C}_{\mathsf{scion}}$ in Figure 6.9, the reference uses only the lower 32 bits.

We use *swizzling* [Mos92, Wil91] to convert references between 32-bit in-memory and 64-bit on-disk addresses. When the collector moves an object to the stale space, it *unswizzles* outgoing reference slots. If a slot references a

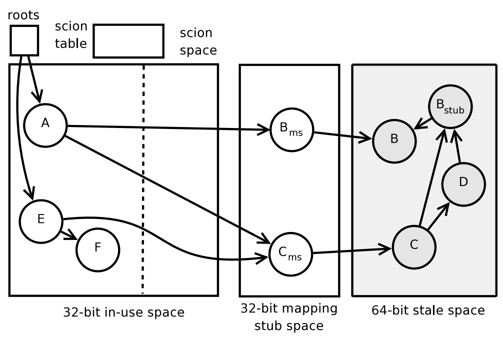


Figure 6.8: **Figure 6.4 with On-Disk Stale Space**

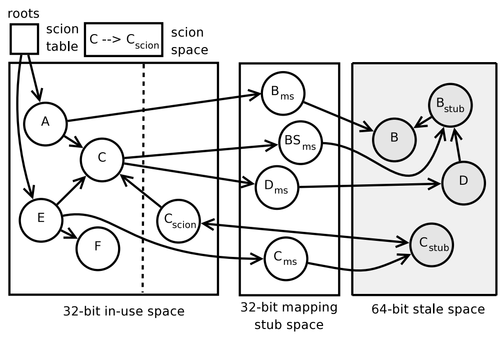


Figure 6.9: **Figure 6.5 with On-Disk Stale Space**

mapping stub, the collector stores the target of the mapping stub in the slot, in order to avoid using in-use memory (the mapping stub) for an intra-disk reference. When a read barrier activates an object in the stale space, it *swizzles* outgoing references by creating the mapping stub for each slot that references a 64-bit object. When the application activates C in Figure 6.8, Melt swizzles its references to B_{stub} and D by creating mapping stubs BS_{ms} (mapping stub of B_{stub}) and D_{ms}, and redirecting the references through them, as shown in Figure 6.9.

Buffering stale objects. Melt initially moves stale objects into in-memory buffers that each correspond to a 64-bit on-disk address range. Buffering enables object scanning and object expansion (from 32- to 64-bit reference slots) to occur in memory, and it avoids performing a native `read()` call for every object moved to the stale space. Furthermore, Melt flushes these buffers to disk gradually throughout application execution time, avoiding increased collection pause times.

Our implementation fills available disk space and crashes when there is no disk space left. A future implementation would limit its disk usage by reserving some space for other applications; it would exit gracefully after reaching this limit; and it would delete files corresponding to the stale space before exiting.

6.2.3 Multithreading

Melt supports multiple application and garbage collection threads by synchronizing shared accesses in read barriers, the scion lookup table, and the stale space. The scion lookup table is a shared, global hash table used during garbage collection to find existing scions for in-use objects referenced by the stale space. For simplicity, table accesses use global synchronization, but for better scalability, a future implementation would use fine-grained synchronization or a lock-free hash table.

Stale space accesses occur when the collector moves an object to the stale space or the application activates a stale object. Melt increases parallelism by using one file per collector thread (there is one collector thread per processor) and by using thread-local buffers for stale objects before flushing them to disk. Each thread allocates stale objects to a different part of the

64-bit stale address range: the high 8 bits of the address specify the thread ID.

An application thread may activate an object allocated by the collector thread on another processor. In this case, the read barrier acquires a per-collector thread lock when accessing the collector thread's buffers and file. When Melt flushes stale buffers in parallel with application execution, it acquires the appropriate collector thread's lock.

6.2.4 Saving Stale Space

This section discusses approaches for reducing the size of the stale space that we have not implemented. With Melt as described, garbage collection is *incomplete* because it does not collect the stale space. Stale objects may become unreachable after they are moved to the stale space and furthermore, they may refer to in-use objects. These uncollectible in-use objects will eventually move to the stale space since they are inherently stale. For example, even if C in Figure 6.6 becomes unreachable, the scion will keep it alive and it will eventually move to the stale space. One solution would be to reference-count the stale space, but reference counting cannot collect cycles. Alternatively, Melt could occasionally trace all memory including the stale space. An orthogonal approach would be to compress the stale space [CKV+03]. The stale space is especially suitable for compression compared with a regular heap because the stale space is accessed infrequently.

6.3 Results

This section evaluates Melt's performance and its ability to tolerate leaks in several real programs and third-party microbenchmarks.

6.3.1 Melt's Overhead

Application overhead. Figure 6.10 presents the run-time overhead of Melt. We run each benchmark in a single medium heap size, two times the minimum in which it can execute. Each bar is normalized to *Base* (an unmodified VM) and includes application and collection time, but not compilation time. Each bar is the median of five trials; the thin error bars show the range of the five trials. For all experiments, except for some bloat experiments, run-to-run variation is quite low since replay methodology eliminates almost all nondeterminism. The variation in bloat is high in general and not related to these configurations. The bottom sub-bars are the fraction of time spent in garbage collection, which is always under 12% in these configurations.

Barriers includes only Melt's read barrier; the barrier's condition is never true since the collector does not mark references stale. *Marking* performs marking of references and objects *on every full-heap GC*, i.e., Melt is always in the MARK state. *Melt memory* performs marking and moving to the stale space on every full-heap GC (i.e., Melt is always in the MOVE & MARK state), but the stale space is in memory rather than on disk. This configuration is analogous to adding a third generation based on object usage in a generational collector. Finally, *Melt* marks objects and moves objects to the on-disk stale space on every full-heap GC.

The graph shows that the read barrier alone costs 6% on average, and adding *Marking* adds no noticeable overhead. The *Melt memory* configuration, which divides the heap into in-use and in-memory stale spaces, has a negligible effect on overall performance. In fact, it sometimes improves collector performance (see below). Storing stale objects on disk (*Melt*) adds 1% to average execution time because of the extra costs of swizzling between 32-

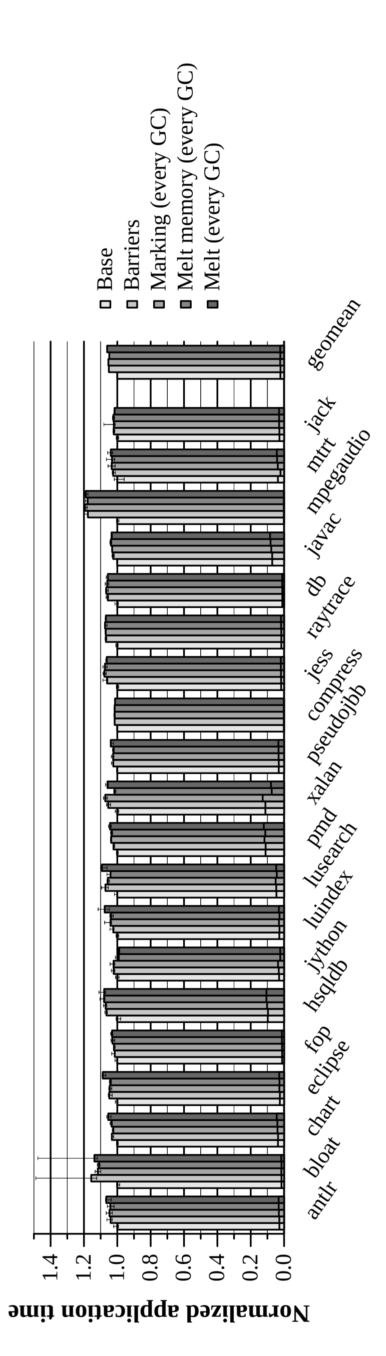


Figure 6.10: **Application execution time overhead of Melt configurations.** Sub-bars are GC time.

and 64-bit references and transferring objects to and from disk. Melt improves the performance of a few programs relative to barrier overhead. This improvement comes from better program locality (jython and lusearch) and lower GC overhead (xalan).

Melt's 6% read barrier overhead is comparable to read barrier overheads for concurrent, incremental, and real-time collectors [BCR03, DLM+78, PFPS07]. With the increasing importance of concurrent software and the potential introduction of transactional memory hardware, future general-purpose hardware is likely to provide read barriers with no overhead. Azul hardware has them already [CTW05]. Melt achieves low overhead because the common case is just two IA32 instructions in optimized code: a register comparison and a branch. An alternative to all-the-time read barriers would be to start without read barriers and recompile to add them to all methods only when the program entered the MARK state. This approach would require full support for on-stack-replacement [SK06].

Collection overhead. Figure 6.11 shows the geometric mean of the time spent in garbage collection as a function of heap size for all our benchmarks using Melt. We measure GC times at 1.5x, 2x, 3x, and 5x the minimum heap size for each benchmark. Times are normalized to *Base* with 5x min heap. Note that the y-axis starts at 1 and not 0.

The graph shows *Marking* slows collection by up to 7% for the smaller heap sizes. The other configurations measure both the overhead and benefits of using the stale space. *Melt memory*, which enjoys the benefits of reduced GC workload and frequency due to stale space discounting, speeds collection 15% over *Marking* and 8% over *Base* for the 1.5x heap. *Melt* adds up to

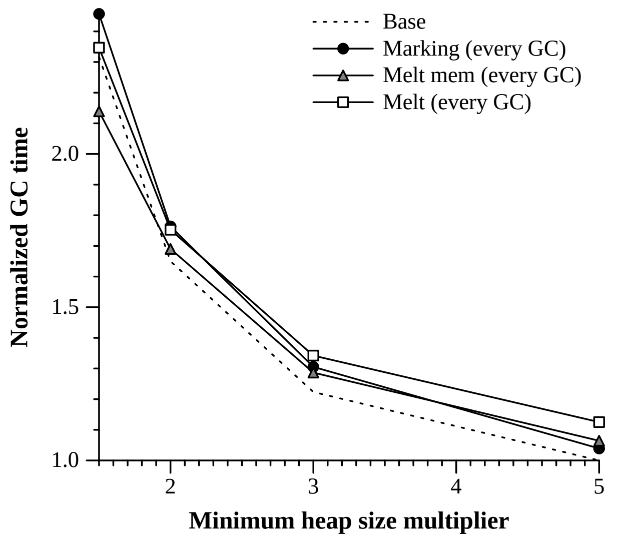


Figure 6.11: **Normalized GC times for Melt configurations across heap sizes.**

10% GC overhead over an in-memory stale space due to pointer swizzling and transferring objects to and from disk. This configuration adds up to 10% over the baseline in large heaps, but benefits and costs are roughly equal at the smallest heap size, where Melt nets just 1% over the baseline.

Compilation overhead. We also measure the compile-time overheads of increased code size and slowing downstream optimizations due to read barriers. Adding read barriers increases generated code size by 10% and compilation time by 16% on average. Because compilation accounts for just 4% on average of overall execution time, the effect of compilation on overall performance is modest.

Melt statistics. Table 6.1 presents statistics for a Melt stress test marks and moves objects every full-heap GC (i.e., the *Melt (every GC)* configuration used in Figures 6.10 and 6.11), for the DaCapo benchmarks. We run with a small heap, 1.5 times the minimum heap size for each benchmark, in order to trigger frequent collections and thus exercise Melt more heavily. The table presents the total number of objects moved to the stale space and activated by the program. It also shows the number of objects in the in-use and stale spaces, pointers from in-use to stale and from stale to in-use, and scions, averaged over each full-heap GC except the first, which we exclude since it does not move any objects to the stale space. The final column is the number of full-heap GCs. The table excludes fop and hsqldb since they execute fewer than two full-heap GCs.

The table shows that Melt moves 9–151 MB to the stale space, and the program activates 0–17 MB of this memory. Some benchmarks activate a significant fraction of stale memory, for example, more than 10% for bloat, eclipse, and lusearch due to this experiment's aggressive policy of moving objects to the stale space on every GC. The next two columns of Table 6.1 show that often more than half of the heap is stale for a long time, which explains the reductions in collection time observed in Figure 6.10. Leak tolerance can improve the performance of applications that do not have leaks *per se* but only use a small portion of a larger working set for significant periods of time. Used this way, leak tolerance is analogous to a fine-grained virtual memory manager for managed languages.

The *In*→*St* column shows the average number of references from in-use to stale objects. These references require a mapping stub to redirect from 32-bit memory to 64-bit disk, but there are usually signficantly fewer mapping

	Total		Average per GC					
	Moved to stale	Activated	In-use	Stale	In→St	St→In	Scions	GCs
antlr	157,486 (10 MB)	28 (0 MB)	68,331 (7 MB)	104,877 (6 MB)	1,592	6,250	2,692	4
bloat	337,126 (19 MB)	51,970 (2 MB)	127,335 (9 MB)	238,167 (14 MB)	13,196	29,701	10,358	6
chart	192,810 (10 MB)	107 (0 MB)	95,139 (14 MB)	153,928 (8 MB)	22,443	23,440	4,079	6
eclipse	1,789,252 (102 MB)	478,518 (17 MB)	258,823 (18 MB)	1,096,570 (65 MB)	48,590	607,619	81,852	24
jython	215,807 (14 MB)	16,842 (1 MB)	47,253 (6 MB)	193,691 (12 MB)	21,028	45,467	30,818	15
luindex	157,814 (9 MB)	308 (0 MB)	55,033 (7 MB)	118,091 (7 MB)	17,718	21,281	1,333	5
lusearch	249,709 (16 MB)	20,287 (2 MB)	92,224 (43 MB)	205,606 (13 MB)	7,593	10,622	9,493	9
pmd	475,714 (26 MB)	28,064 (1 MB)	125,625 (8 MB)	337,292 (19 MB)	39,811	18,787	10,419	16
xalan	701,733 (151 MB)	18,892 (1 MB)	43,538 (13 MB)	461,928 (87 MB)	26,741	77,565	5,173	101

Table 6.1: **Statistics for the DaCapo benchmarks running *Melt (every GC)* with 1.5 times the minimum heap size.**

stubs than in-use objects. The *St→In* and *Scions* columns show the number of references from stale to in-use objects and the number of scions, respectively. The next section shows that for growing leaks, the number of scions stays small and proportional to in-use memory, while references from stale to in-use grow with the leak, motivating Melt's use of stub-scion pairs.

6.3.2 Tolerating Leaks

This section evaluates how well Melt tolerates growing leaks in the wild by running programs with leaks longer and maintaining program performance. Table 6.2 shows all 10 leaks we found and could reproduce: two leaks in Eclipse, EclipseDiff and EclipseCP; a leak in a MySQL client application; a leak in Delaunay, a scientific computing application; a real leak in SPECjbb2000 and an injected leak in SPECjbb2000 called JbbMod; a leak in Mckoi, a database application; and three third-party microbenchmark leaks: ListLeak and SwapLeak from Sun Developer Network, and DualLeak from IBM developerWorks. Melt tolerates 5 of these 10 leaks well; it tolerates 2 leaks but adds high overhead by activating many stale objects; and it does not significantly help 3 leaks.

Melt cannot tolerate leaks in SPECjbb2000 and DualLeak because they are *live* leaks: the programs periodically access the objects they access. For example, DualLeak repeatedly adds String objects to a HashSet. It does not remove or use these objects again. However, when the HashSet grows, it re-hashes all the elements and accesses the String objects, so the String cannot remain in the stale space permanently. It seems challenging in general to determine that an object being accessed is nonetheless useless. However, future work could design *leak-tolerant data structures* that avoid inadvertently accessing objects that the application has not accessed in a while. At least two

Leak	(LOC)	Melt's effect	Reason
EclipseDiff	(2.4M)	Runs until 24-hr limit (1,000X longer)	Virtually all stale
EclipseCP	(2.4M)	Runs until 24-hr limit (194X longer)	All stale?
JbbMod	(34K)	Runs until crash at 20 hours (19X longer)	All stale?
ListLeak	(9)	Runs until disk full (200X longer)	All stale
SwapLeak	(33)	Runs until disk full (1,000X longer)	All stale
MySQL	(75K)	Runs until crash (74X longer; HAO)	Almost all stale
Delaunay	(1.9K)	Some help (HAO)	Short-running
SPECjbb2000	(34K)	Runs 2.2X longer	Most stale memory in use
DualLeak	(55)	Runs 2.0X longer	Almost all stale memory in use
Mckoi	(95K)	Runs 2.2X longer	Threads' stacks leak

Table 6.2: **Ten leaks and Melt's ability to tolerate them.** *HAO* means "high activation overhead."

other leaky programs, EclipseDiff and MySQL, also have live leaks, although they leak significantly more dead than live memory, so Melt can still improve their longevity and performance significantly.

We run the following experiments in maximum heap sizes chosen to be about twice what each program would need if it were not leaking. All the programs except Delaunay have growing leaks, so their behavior with and without Melt is not very sensitive to maximum heap size. All programs have a memory ceiling, which may be heap size, physical memory, or virtual memory, although physical memory is always a ceiling since it causes GC to thrash [HFB05, YBKM06]. Melt extends a program's memory ceiling to include all available disk space, substantially postponing a crash. We run Jikes RVM in uniprocessor mode because in multiprocessor mode, the VM often crashes before completing runs lasting many hours, apparently due to bugs in either Melt or Jikes RVM.

EclipseDiff. Eclipse is an integrated development environment (IDE) written in Java with over 2 millions lines of source code [Eclb]. We reproduce Eclipse bug #115789, which reports that repeatedly performing a structural (recursive) *diff*, or compare, slowly leaks memory that eventually leads to memory exhaustion. The leak occurs because a data structure for navigation history maintains references it should not. It exists in Eclipse 3.1.2 but was fixed by developers for Eclipse 3.2 after we reported a fix when working on Bell.

We automate repeated structural differences via an Eclipse plug-in that reports the wall clock time for each iteration of the difference. Figure 6.12 shows the time each iteration takes for vanilla Jikes RVM 2.9.2, the Sun JVM 1.5.0, and Jikes RVM with Melt. We use iterations as the x-axis. This figure

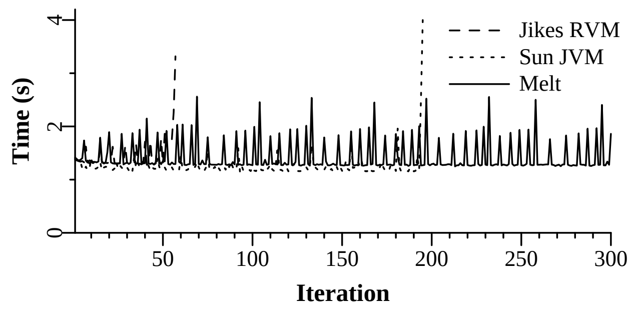


Figure 6.12: **Performance comparison of *Jikes RVM*, *Sun JVM*, and *Melt* for the first 300 iterations of EclipseDiff.**

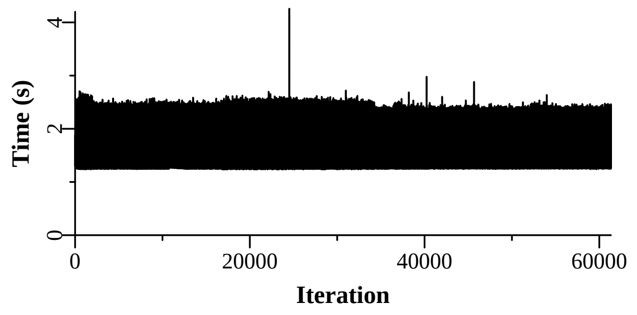


Figure 6.13: **Performance of *Melt* running EclipseDiff leak for 24 hours.**

shows the first 300 iterations in order to compare the three VMs, and Figure 6.13 shows the performance of just Melt for its entire run (terminated by us after 24 hours). Unmodified Jikes RVM slows and crashes after about 50 iterations when its heap fills. Sun JVM, which uses a more space-efficient collector than the generational copying collector used by Jikes in our experiments, runs almost 200 iterations before slowing down and crashing.

Melt's performance stays steady in the long term with variations in the short term. All VMs' performance varies per iteration because iterations interrupted by a full-heap GC take longer. Melt's performance varies more because

full-heap GCs that move objects to the stale space take longer: Melt moves objects to the stale space, unswizzles their references, and creates stub-scion pairs and mapping stubs. Melt buffers new stale objects in memory during these GCs, and it flushes these buffers to disk gradually during application execution. Without this gradual flushing, performance varies more. When we terminate Melt at 24 hours, it has written over 80 GB to the on-disk stale space.

Figures 6.14 and 6.15 show reachable memory, as reported at the end of the last full-heap GC, for the same VMs at each iteration. Unmodified Jikes RVM and Sun JVM fill the heap as the leak grows, while Melt starts moving stale objects to the disk when the heap reaches 80% full, and it keeps memory usage fairly constant in the long term. The figures show that memory usage oscillates gradually between about 100 and 130 MB: (1) Melt moves objects to *buffers* for the stale space when usage reaches 130 MB; (2) it then slowly flushes these buffers to disk over time; and (3) in the meantime, the leak continues to increase heap size until it reaches 130 MB again and triggers Melt to repeat the cycle.

Figures 6.16 and 6.17 report numbers of objects and references at each iteration of EclipseDiff. We divide the data between two graphs since the magnitudes vary greatly. Figure 6.16 shows that references from stale to in-use and objects in the stale space both grow linearly over iterations and have large magnitudes. This result motivates avoiding a solution that uses time or space proportional to stale objects or references from stale to in-use objects. Figure 6.17 shows that Melt holds in-use objects relatively constant over iterations. The number of scions grows linearly over time, although it stays small in magnitude: roughly one scion per iteration. This growth occurs because a

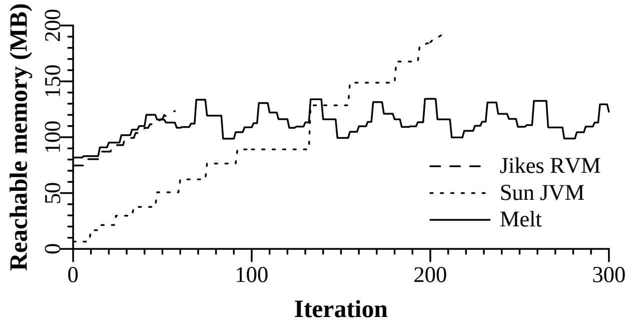


Figure 6.14: **Comparison of reachable memory for the first 300 iterations of EclipseDiff.**

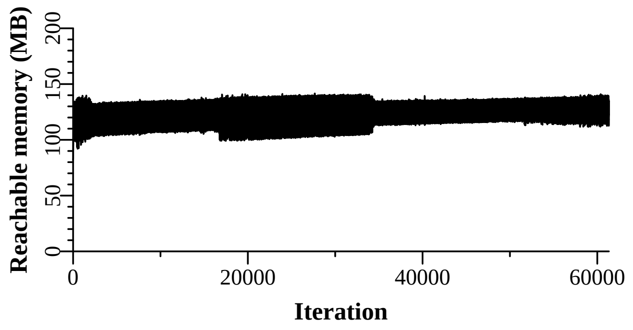


Figure 6.15: **Reachable memory running EclipseDiff with Melt for 24 hours.**

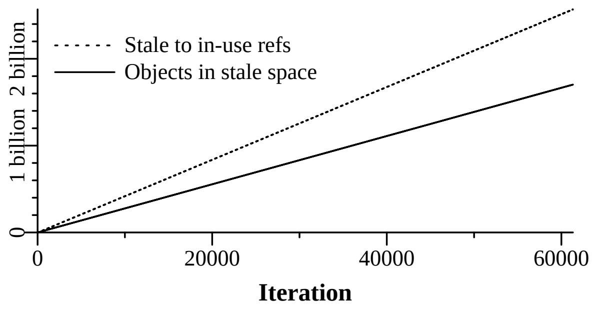


Figure 6.16: **EclipseDiff leak with Melt: stale objects and references from stale to in-use.**

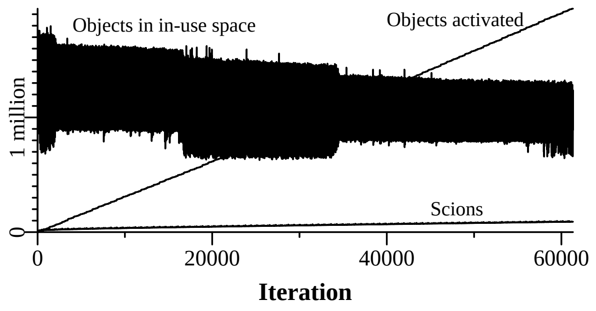


Figure 6.17: **EclipseDiff leak with Melt: in-use objects, objects activated, and scions.**

very small part of the leak is *live*. Each iteration leaks a large data structure, and the root object of this structure remains live, and this object uses an extra scion.

The graph shows that the number of objects activated increases linearly but its magnitude is still small compared with objects in the stale space, i.e., just a few stale objects are activated. Each activated object needs a scion, and many more objects are activated than there are scions, which shows that the application activates the same objects over and over again. A future implementation could consider a different policy for objects that have been activated, as LeakSurvivor does [TGQ08]. The fact that scions stay relatively small while stale-to-in-use references grow significantly, motivates Melt's use of stub-scion pairs to maintain references from stale to in-use objects. In contrast, LeakSurvivor's *swap-out table*, which maintains references from disk to memory, does not guarantee time or space proportional to in-use memory.

For the other leaks in this section that Melt tolerates, we observe similar ratios for in-use and stale objects and references between them.

EclipseCP. We reproduce Eclipse bug #155889, which reports a growing leak when the user repeatedly cuts text, saves, pastes the same text, and saves again. We wrote an Eclipse plug-in that exercises the GUI and performs this cut-paste behavior. Figure 6.18 shows the run time of each iteration of a cut-save-paste-save of a large block of text, using a logarithmic x-axis since unmodified Jikes runs for a short time before running out of memory. We do not present data for Sun JVM since we could not reproduce the leak with it. The figure shows that Melt adds some overhead to EclipseCP, but it is able to execute with fairly constant long-term performance for nearly 200 times as many iterations as without Melt. We terminate Melt after 24 hours, at which point it has used 39 GB of disk space. We note that the performance fluctuations are due to the application, not Melt, since they occur with unmodified Jikes RVM. Figure 6.19 shows memory usage over time, with and without Melt. Melt holds memory fairly steady in the long term. The short-term fluctuations are due to Melt moving objects gradually to the stale space each time the heap reaches 80%.

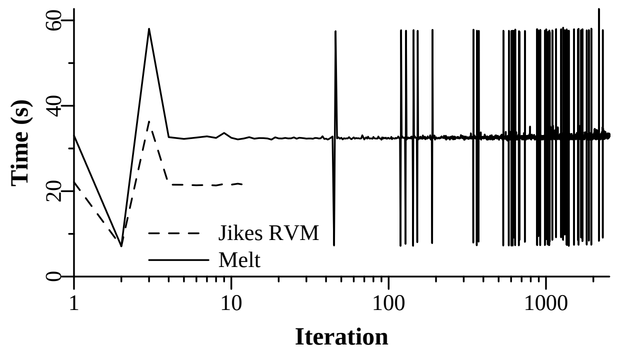


Figure 6.18: **EclipseCP performance over time, with and without Melt.** X-axis is logarithmic x-axis to show behavior of both VMs.

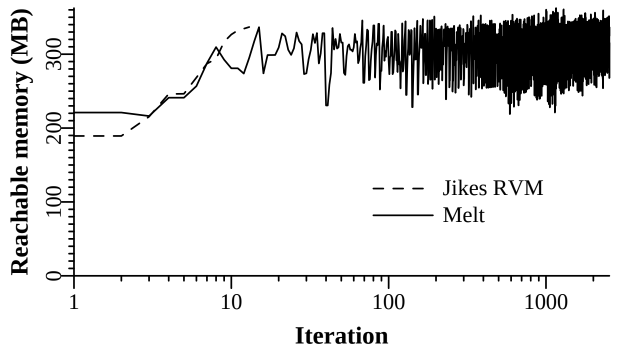


Figure 6.19: **EclipseCP reachable memory over time, with and without Melt.** X-axis is logarithmic x-axis to show behavior of both VMs.

JbbMod. SPECjbb2000 (see below) has significant *live* heap growth. Tang et al. modified SPECjbb2000 by injecting a leak of *dead* (permanently stale) objects [TGQ08], which we call JbbMod. Tang et al. execute JbbMod using LeakSurvivor for two hours before terminating the experiment. Melt runs almost 21 hours (almost 20 times more iterations than without Melt) before crashing with an apparent heap corruption error, likely due to a bug in Melt. During this time, it keeps performance and memory usage fairly constant. We thus believe that all heap growth is dead and, in lieu of crashing, Melt would run the program as long as disk space allowed.

ListLeak. The first microbenchmark leak is from a post on the Sun Developer Network [Sun03b]. It is a very simple and fast-growing leak:

```
List list = new LinkedList();
while (true) list.add(new Object());
```

Clearly this leak grows very quickly. Whereas unmodified Jikes RVM and Sun JVM crash in seconds, Melt keeps ListLeak running until it fills 126 GB of disk, which takes about 100 minutes.

SwapLeak. This leak also comes from a message posted on the Sun Developer Network [Sun03a]. The message asks for help understanding why an attached program runs out of memory. The program first initializes an array of 1,000,000 SObjects, which each contain an inner class Rep. The program then swaps out each SObject's Rep object with a new SObject's Rep object. Intuitively it seems that the second operation should have no net effect on reachable memory. However, as explained by a response to the message, the VM keeps a reference from an inner class object back to its containing object, which causes

the swapped-out Rep object and the new SObject to remain reachable. The fix is to make the inner class static. Melt provides the illusion of a fix until the developer applies the correction.

The swapping operation leaks only about 64 MB, so we add a loop around this operation to create a growing leak. SwapLeak grows nearly as quickly as ListLeak, and unmodified Jikes RVM and Sun JVM survive fewer than five iterations. Melt runs for 2,341 iterations (7 hours) and then terminates when it fills the available 126 GB of disk space.

MySQL. The MySQL leak is a simplified version of a JDBC application from a colleague. The program exhausts memory unless it acquires a new connection periodically. The leak, which is in the JDBC library, occurs because SQL statements executed on a connection remain reachable unless the connection is closed or the statements are explicitly closed. The MySQL leak repeatedly creates a SQL statement and executes it on a JDBC connection. We count 1,000 statements as an iteration. The application stores the statement objects in a hash table. The program periodically accesses them when the hash table grows, re-hashing the statement objects. However, in terms of bytes, objects referenced by the statement objects contribute much more to the leak, i.e., the vast majority of objects are permanently stale.

Melt tolerates this leak but periodically suffers a huge pause when the hash table grows and re-hashes its elements, which activates all statement objects. Figures 6.20 and 6.21 show the performance (logarithmic y-axis) and memory usage of MySQL over time, with and without Melt. Unmodified Jikes RVM and Sun JVM quickly run out of memory, but Melt keeps the program running for 74 times as many iterations as Jikes RVM. When the

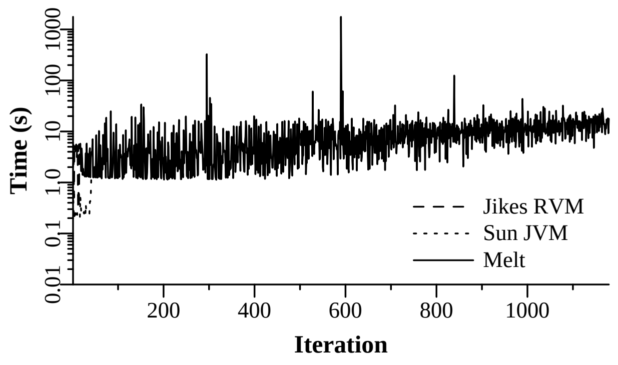


Figure 6.20: **MySQL performance over time, with and without Melt.** The y-axis is logarithmic because some pause times are quite high.

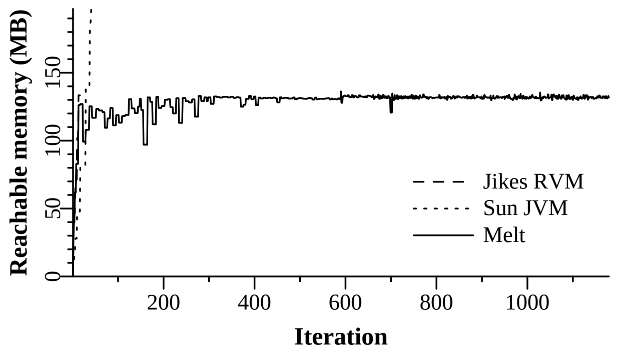


Figure 6.21: **MySQL reachable memory over time, with and without Melt.**

hash table of statements grows and re-hashes its elements, e.g., at iterations 300 and 600, pause times rise to 30 minutes. Our implementation of Melt is not optimized for activation performance since it does not consider locality when moving objects to disk or activating objects. Alternatively, we could attempt to recognize that statement objects are highly stale but live, not dead.

Melt terminates with an unrelated corruption error, mostly likely due

Input	Jikes	Melt memory	Melt		
size	Time	Time	Time	Stale	Activated
15,000	7 s	7 s	7 s	0 MB	0 MB
20,000	11 s	10 s	12 s	0 MB	0 MB
21,000	12 s	14 s	15 s	0 MB	0 MB
22,000	OOM	18 s	45 s	90 MB	14 MB
25,000	OOM	19 s	98 s	94 MB	18 MB
30,000	OOM	27 s	166 s	118 MB	25 MB

Table 6.3: **Delaunay run times, stale memory, and activated memory for various input sizes.** *OOM* means out of memory.

to a bug in Melt or perhaps Jikes RVM, after 4 hours and 20 minutes. Melt could tolerate the leak longer if the VM did not crash, albeit with periodic pauses to activate all statement objects.

Delaunay. Next we present a leak in Delaunay, an application that performs a Delaunay triangulation, which generates a triangle mesh, for a set of points, that meets a set of constraints [GKS90]. We obtained the program from colleagues who added a *history directed acyclic graph* (DAG) to reduce algorithmic complexity of the triangulation, but the change inadvertently caused graph components no longer in the graph to remain reachable.

Delaunay is not a growing leak in a long-running program. Rather, this leak degrades program performance and prevents the program from running input sizes and heap sizes that would work without the leak. To highlight this problem, we execute the program with a variety of input sizes, comparing Jikes RVM to Melt's memory and disk configurations.

Table 6.3 shows run times for all configurations and how much memory is transferred to and from disk using a maximum heap size of 256 MB and a variety of input sizes with a focus on 20,000-22,000 iterations, when the program exhausts memory. We set the threshold for moving objects to the stale space at 95% to avoid moving objects to the stale space too aggressively. For input sizes $\leq$21,000 iterations, all VMs perform similarly since the program has enough memory. Starting with 22,000 iterations, Melt tolerates the leak while the unmodified VM runs out of memory. The performance of Melt with an in-memory stale space scales well with input size. The on-disk stale space's performance does not scale well because Melt activates many objects from disk, which becomes expensive when the working set of accesses exceeds disk buffering. At some point, Melt is going beyond tolerating the leak, i.e., the heap would not be large enough even if the leak were fixed, as indicated by the increasing amount of activated memory.

These results show that Melt can help somewhat with short-running leaks, but it can add significant overhead if it incorrectly moves many live objects to the stale space, since activation overhead will be high.

SPECjbb2000. SPECjbb2000 simulates an order processing system and is intended for evaluating server-side Java performance [Sta01]. It contains a known, growing memory leak that manifests when it runs for a long time without changing warehouses. It leaks because it adds orders to an order list and does not correctly remove some of them when the orders complete.

Although SPECjbb2000 experiences unbounded heap growth over time, it uses almost all the objects. The program periodically accesses all orders in the order list. It seems unlikely that any system will be able to differentiate useful from useless memory accesses. We note our staleness-based leak detector using Bell diagnoses this leak because a small part of each order's data structure is stale (Section 4.4). Melt executes SPECjbb2000 about twice as long

as without Melt (1166 vs. 540 iterations) since it finds some stale memory to move to disk. However, performance suffers beginning at about 650 iterations because Melt starts moving many objects that are not permanently stale to disk in order to avoid running out of memory, resulting in significant activation overhead.

DualLeak. This leak comes from an example in an IBM developerWorks column [GP05]. We call it DualLeak since its 55 sources lines contain two different leaks. The program executes in iterations and exercises both leaks during each iteration. The first leak is slow-growing and occurs because of an off-by-one error that leads to an Integer object not being removed from a Vector on each iteration. The other leak grows more quickly by adding multiple String objects to a HashSet on each iteration.

Melt cannot tolerate either leak since the program accesses all of the Vector and HashSet periodically. The Vector leak accesses all slots in the Vector every iteration, since it removes elements from the middle of the vector, causing all leaked elements to the right to be moved one slot to the left. The HashSet repeatedly adds String objects that are accessed during re-hashing.

Melt executes twice as many iterations of DualLeak as unmodified Jikes RVM by swapping out the HashSet elements when they are not in use. But this approach is not sustainable. When the HashSet grows, Melt activates its elements, hurting performance and eventually running out of memory.

Mckoi. We reproduce a memory leak reported on a message board for Mckoi SQL Database, a database management system written in Java [Mck02]. The leak occurs if a program repeatedly opens a database connection, uses the connection, and closes the connection. Mckoi does not properly dispose of the connection thread, leading to a growing number of unused threads. These threads leak memory; most of the leaked bytes are for each thread's stack.

Melt cannot tolerate this leak because stacks are VM objects in Jikes RVM, so they may not become stale. Also, program code accesses the stack directly, so read barriers cannot intercept accesses to stale objects. However, we could modify Melt to detect stale threads (threads not scheduled for a while) and make their stacks stale and also allow objects directly referenced by the stack to become stale. If the scheduler scheduled a stale thread, Melt would activate the stack and all objects referenced by the stack.

Melt runs the leak for about twice as long as unmodified Jikes RVM because Melt still finds some memory to swap out that is not in use, but soon the leaked stacks dominate memory usage and exhaust memory.

6.4 Conclusion and Interpretation

While managed languages largely insulate programmers from memory management, programmers still need to ensure unused objects become unreachable. Otherwise they will produce programs that may use bounded live memory but have unbounded memory requirements. This work provides some protection for these programs by utilizing available disk space. Melt moves all highly stale objects to disk, which overestimates dead objects but works fairly well since Melt can recover live objects from disk. MySQL adds high space overhead since it activates many highly stale but live objects. The next chapter presents an approach that eliminates likely leaked memory, so it necessarily uses a more precise leak prediction heuristic. Melt could benefit from this more precise heuristic to reduce high activation overhead.

Melt is well suited to the deployed setting. Leaks are input and environment sensitive, and they can grow slowly and take days, weeks, or longer to have effects. For non-leaking programs, Melt adds low overhead, or no overhead if it adds instrumentation only when the program starts to run out of memory. If a leak does occur in deployed software, the system can activate Melt and reap its benefits. Melt is a bug tolerance approach that can be built into commercial VMs, deployed with current software, and automatically improve robustness and increase user happiness. It points to a future where systems provide, and users use, a variety of bug tolerance approaches that on the whole improve system reliability dramatically.

Chapter 7

Leak Pruning

The previous chapter presented an approach that relegates highly stale objects to disk. Although disks are typically orders of magnitude larger than main memory, a growing leak will eventually exhaust available disk space. Another concern is embedded devices that have no disk. This chapter introduces an approach orthogonal to Melt called *leak pruning* that uses *bounded resources*. It defers out-of-memory errors by predicting which objects are dead and reclaiming them when the program is about to run out of memory. As long as the program does not attempt to access reclaimed objects, it may run indefinitely. Otherwise, a leak pruning-enabled VM intercepts the access and throws an error. This behavior preserves semantics since the program already ran out of memory. In the worst case, leak pruning only defers out-of-memory errors. In the best case, it enables leaky programs with unbounded reachable memory growth to run indefinitely with bounded resources.

The key to leak pruning is precisely predicting which reachable objects are dead. In contrast, Melt simply predicts that highly stale objects are leaked since it can recover from mistakes by activating live objects from disk. Our dynamic prediction algorithm for leak pruning identifies stale *data structures*, i.e., stale subgraphs, during tracing garbage collection. It records the source and target classes of the first reference and the number of stale bytes in the stale data structure. When the VM runs out of memory, leak pruning *poisons* references to instances of the data structure type consuming the most bytes. Poisoning invalidates and specially marks references. The collector then reclaims objects that were only unreachable from these references. If the program subsequently accesses a poisoned reference, the VM throws an error.

We show leak pruning, which uses read barriers similar to Melt's, adds low enough overhead for deployed systems. We evaluate it on the same 10 leaks as in the previous chapter. Leak pruning is about as effective as Melt with an infinite disk for all leaks but two. Melt seems to run `JbbMod` indefinitely, but leak pruning eventually runs out of memory since it fails to identify some small part of the leak. Leak pruning extends `MySQL`'s lifetime for about as long as Melt but avoids Melt's slowdowns due to high activation overhead since leak pruning's more accurate identification algorithm does not select a small in-use part of the leak.

Modern software is never bug-free. In the event of an unexpected memory leak in a deployed system, leak pruning offers an alternative to running out of memory: reclaim some memory and keep going. Our evaluation shows that leak pruning often picks well and keeps real leaks going much longer or indefinitely.

7.1 Approach and Semantics

This section describes the high-level approach and semantics of leak pruning. Section 7.2 presents the details of our algorithm and implementation.

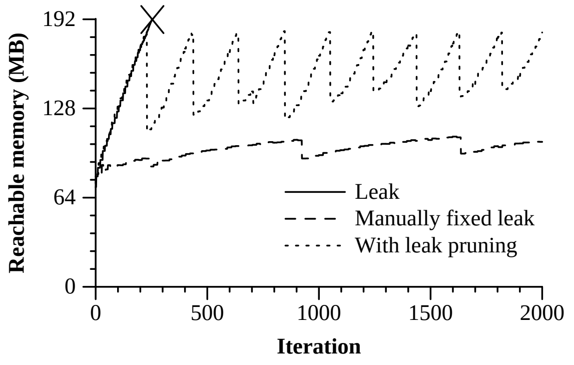


Figure 7.1: **Reachable heap memory for the EclipseDiff leak.** An unmodified VM running the leak, a manually fixed version, and the leak running with leak pruning.

7.1.1 Motivation

Figure 7.1 shows the memory consumption over time measured in *iterations* (fixed amounts of program work) for EclipseDiff, the growing leak described in the previous chapter. The graph shows reachable memory at the end of each full-heap collection. The solid line shows that the leak causes reachable memory to grow without bound until it overflows the heap. At 200 MB for this experiment, the VM throws an out-of-memory error (the plotted line reaches only 192 MB since the program did not complete another iteration before exhausting memory).

The dashed line shows reachable memory if we modify the Eclipse source to fix the leak. Reachable memory stays fairly constant over time, and Eclipse does not run out of memory. The dotted line shows reachable memory with leak pruning. When the program is about to run out of memory, leak pruning reclaims objects that it predicts are dead. It cannot reclaim all dead objects promptly because objects need time to become stale. Section 7.4 shows that leak pruning keeps EclipseDiff from running out of memory

for over 50,000 iterations (24 hours).

Leak pruning seeks to close the gap between liveness and reachability. When a program starts to run out of memory, leak pruning observes program execution to predict which reachable objects are dead and therefore will not be used again. When the program actually runs out of memory, it *poisons* references to these objects and reclaims them. If the application subsequently attempts to read a poisoned reference, the VM throws an internal error, giving the original out-of-memory error as the cause. Since the program has *executed beyond* an out-of-memory error, throwing an internal error does not violate semantics. The goal of leak pruning is to defer out-of-memory errors indefinitely by eliminating the space and time overheads due to leaks.

7.1.2 Triggering Leak Pruning

Figure 7.2 shows a high-level state diagram for leak pruning. Leak pruning's state is based on how close the program is to running out of memory. Leak pruning performs most of its work during full-heap garbage collections, and it changes state after each full-heap GC. State changes depend on how full the heap is at the end of GC.

Initially, leak pruning is INACTIVE and does not observe program behavior. This state has two purposes. First, it avoids the overhead of leak pruning's analysis when the program is not running out of memory. Second, it collects potentially better information by focusing on program behavior that appears to be leaking. Leak pruning remains INACTIVE until reachable memory exceeds "expected memory use," a user-configurable threshold. By default our implementation sets this threshold to 50% since users typically execute programs in heaps at least twice as large as maximum reachable memory. Leak

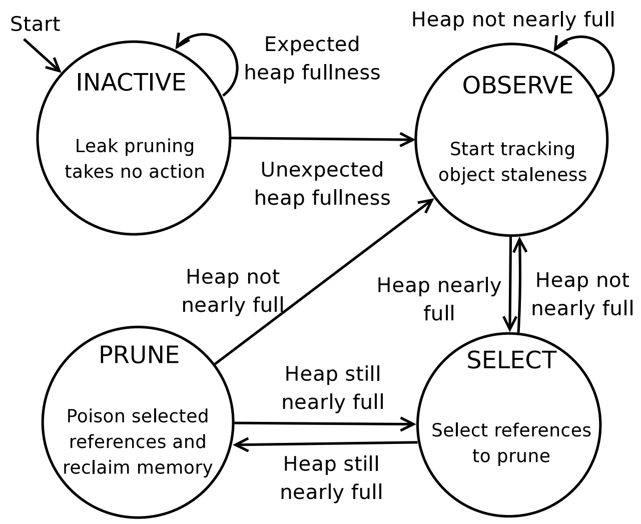


Figure 7.2: **State diagram for leak pruning.**

pruning is not very sensitive to the exact value of this threshold. If set too low, leak pruning may incur some overhead when the program is not leaking; if set too high, it will have less time to observe program behavior before selecting memory to reclaim.

When memory usage crosses this threshold, leak pruning enters the OBSERVE state, in which it analyzes program reference patterns to choose pruning candidates (described in detail in Section 7.2). Once leak pruning enters the OBSERVE state, it never returns to the INACTIVE state because it considers the application to be in a permanent unexpected state.

Leak pruning moves from OBSERVE to SELECT when the program has nearly run out of memory, which is user-configurable and 90% of available memory by default. In SELECT, leak pruning chooses references to prune, based on information collected during the OBSERVE state (described in detail in Section 7.2).

In principle, we would like to move to the PRUNE state only when the program has completely exhausted memory. However, executing until reachable objects fill available memory can be expensive. Because reachable memory usually grows more slowly than the allocation rate, allocations trigger more and more collections as memory fills the heap. Thus, we support two options: (1) leak pruning moves to PRUNE when the heap is 100% full after collection, i.e., the VM is about to throw an out-of-memory error,[1] or (2) it moves to the PRUNE state immediately after finishing a collection in the SELECT state. The VM has flexibility in how it reports memory usage, since details such as object header sizes are not visible to the application, so the second option is not necessarily a violation of program semantics. We believe the second option is more appealing since it avoids the application grinding to a halt before pruning can commence. Users should consider the "nearly full" threshold to be the maximum heap size and "full" to be extra headroom to perform GCs efficiently. We use option (2) by default but also evaluate (1). Regardless, after entering PRUNE once, leak pruning always enters PRUNE immediately following SELECT since it has already exhausted memory once.

The PRUNE state *poisons* selected references by invalidating them and not traversing the objects they reference. The collector then automatically reclaims objects that were reachable only from the pruned references. If the collector reclaims enough memory so that the heap is no longer nearly full, leak pruning returns to the OBSERVE state. Otherwise, it returns to SELECT and identifies more references to prune.

[1]In this case leak pruning remains in SELECT until heap exhaustion but does not repeat the selection process.

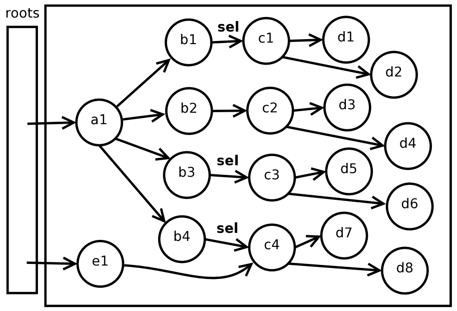


Figure 7.3: **Example heap after the SELECT state.** References selected for pruning are marked with **sel**.

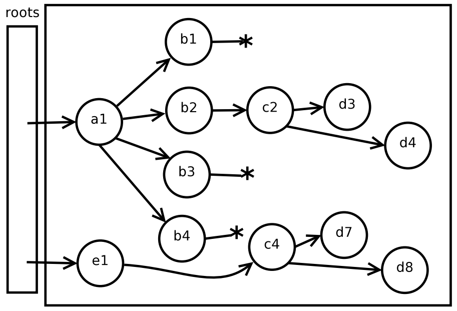


Figure 7.4: **Example heap at the end of GC in PRUNE state.** Poisoned references end in an asterisk (*).

Example. Figure 7.3 shows an example heap when leak pruning enters the PRUNE state. Each circle is a heap object. Each object instance has a name based on its *class*, e.g., b1, b2, b3, and b4 are instances of class B. The selection algorithm uses class to select references to prune (Section 7.2). The figure shows that objects a1 and e1 are directly reachable from the program roots (registers, stacks, and statics), and other objects are transitively reachable. Suppose leak pruning selects three references to prune, labeled **sel** in the figure: b1 → c1, b3 → c3, and b4 → c4.

7.1.3 Reclaiming Reachable Memory

During a full-heap collection in the PRUNE state, the collector repeats its analysis, but this time poisons selected references and reclaims all objects reachable only from these references as shown in Figure 7.4. The collector reclaims objects reachable only from pruned references since it does not trace pruned references. For example, the subtree rooted at c4 is not reclaimed because it is transitively reachable from the roots via object e1, whereas b3 has the only reference to its subtree.

Leak pruning poisons a reference by setting its second-lowest-order bit (Section 7.2.4). Setting the reference to null is insufficient since that could change program semantics. If the program accesses a poisoned reference, the VM intercepts the access and throws an internal error with an attached out-of-memory error. This behavior preserves semantics since the program previously ran out of memory when it entered the PRUNE state for the first time.

7.1.4 Exception and Collection Semantics

The Java VM specification says the VM may throw OutOfMemoryError only at program points responsible for allocating resources, e.g., new expressions or expressions that may trigger class initialization [LY99a]. Program accesses to pruned memory are at reference loads, which are not memory-allocating expressions. The Java specification however permits InternalError to be thrown asynchronously at any program point. Our implementation thus throws an InternalError if the program accesses a pruned reference.

When the VM runs out of memory, leak pruning records and defers the error. However, if the application can catch and handle the out-of-memory error, then deferring the error violates semantics. Catching out-of-memory errors is uncommon since these errors are not easy to remedy. In Java, a regular try { ... } catch (Exception ex) { ... } will not catch an OutOfMemoryError since it is on a different branch of the Throwable class hierarchy. Some applications, such as Eclipse, catch all errors in an outer loop and allow other components to proceed, but the Eclipse leaks we evaluate cannot do useful work after they catch out-of-memory errors. Deciding whether to reclaim memory or throw an out-of-memory error when there is a corresponding catch block, can be an option set by users or developers.

Leak pruning may affect object finalizers, which are custom methods that help clean up non-memory resources when an object is collected, e.g., to close a file associated with an object. Pruning causes objects to be collected earlier than without pruning, so calling finalizers could change program behavior. A strict leak pruning implementation would disable finalizers for the rest of the program after it started pruning, which does not technically violate the Java specification since there is no timeliness guarantee for finalizers. Our implementation currently continues to call finalizers after pruning starts, which would likely be the option selected by users and developers in order to avoid running out of non-memory resources while tolerating memory leaks.

Leak pruning provides information for leak diagnostics. When the VM first runs out of memory, leak pruning optionally reports an out-of-memory "warning." If the program later accesses a pruned reference, the VM throws an InternalError whose getCause() method returns the original OutOfMemoryError. In verbose mode, leak pruning provides information about the data structures chosen for pruning by the SELECT state, as well as the reasons they were selected.

7.2 Algorithm and Implementation

This section describes our algorithm and implementation for predicting which reachable objects are leaked, selecting which references to prune, poisoning them, and detecting attempted accesses to poisoned references. Leak pruning first identifies references to data structures that are highly *stale*. It prunes stale data structures based on the following criteria: (1) no instance of the data structure was stale for a while and then used again, and (2) the data structures contain many bytes.

7.2.1 Predicting Dead Objects

Our prediction algorithm has the following key objectives: (1) perfect accuracy, (2) high coverage, and (3) low overhead. If the prediction algorithm is not perfect, the program will access a pruned object and will terminate. However, if the prediction algorithm is not aggressive enough, it will not prune all the leaking objects. Of course, predicting liveness perfectly in all cases is beyond reach, but we have developed an algorithm with high coverage and accuracy that works well in many cases. Any prediction algorithm preserves correctness since leak pruning ensures accesses to reclaimed memory are intercepted.

Since leaks add space and time overhead, the prediction algorithm should not make matters worse. In particular, we should not add space proportional to the objects in the heap. Our algorithm steals three available bits in object headers and two unused lowest-order bits in object-to-object references. When the program starts to run out of memory, it stores concise summaries of which reference *types* are highly stale.

7.2.2 The OBSERVE State

In the OBSERVE state, leak pruning starts keeping track of each object's staleness, and it also summarizes accesses to highly stale references.

Tracking staleness. In the OBSERVE state, leak pruning tracks each object's *staleness*, i.e., how long since the program last used it. Our implementation maintains staleness using a three-bit *logarithmic stale counter* in each object's header, which we first used in our leak detector using Bell. A value k in an object's stale counter means the program last used the object approximately 2^k collections ago. We maintain each stale counter's value by (1) incrementing object counters in each collection and (2) inserting instrumentation to clear an object's counter when it is used.

The collector keeps an exact count of the number of full-heap garbage collections. Every full-heap collection i increments an object's stale counter if and only if i evenly divides 2^k, where k is the current value of the counter. As for Melt, the collector sets the lowest bit of every object-to-object *reference* to allow for a quick read barrier test. The following pseudocode shows the read barrier:

```
b = a.f;                   // Application code
if (b & 0x1) {             // Read barrier
  // out-of-line cold path
  t = b;                   // Save ref
  b &= ~0x1;               // Clear lowest bit
  a.f = b; [iff a.f == t]  // Atomic w.r.t. read
  b.staleCounter = 0x0;    // Atomic w.r.t. read
}
```

Similar to the Melt read barrier, the cold path is out-of-line in a separate

method, and the barrier update is atomic with respect to the read, to avoid overwriting another thread's write.

Edge table. Starting in the OBSERVE state, leak pruning maintains an *edge table* to track the staleness of heap references based on type. For a stale edge in the heap, $src \rightarrow tgt$, the table records the Java class of the source and target objects: $src_{class} \rightarrow tgt_{class}$. Each entry summarizes an equivalence relationship over object-to-object references: two references are equivalent if their source and target objects each have the same class. Each edge entry $src_{class} \rightarrow tgt_{class}$ records: bytesUsed, which is used later in the SELECT state, and maxStaleUse, which identifies edge types that are stale for a long time, but not dead. Leak pruning only prunes objects that are more stale than their entry's maxStaleUse. We record in maxStaleUse the all-time maximum value of tgt's stale counter when a barrier accesses a reference $src_{class} \rightarrow tgt_{class}$. The read barrier executes the following code as part of its out-of-line cold path:

```
if (b.staleCounter > 1)
  edgeTable[a.class->b.class].maxStaleUse =
   max(edgeTable[a.class->b.class].maxStaleUse, b.staleCounter);
```

The update occurs only if the object's stale counter is at least 2, since a value of 1 is not very stale (stale only since the last full-heap collection). We find stale objects are used infrequently, and the edge table update occurs infrequently.

7.2.3 The SELECT State

A full-heap collection in SELECT chooses *one* edge type for pruning. It divides the regular transitive closure, which marks live all reachable objects, into two phases:

1. The *in-use transitive closure* starts with the roots (registers, stacks, statics) and marks live objects, except for when it encounters a stale reference whose target object has a stale counter at least two greater than its edge table entry's maxStaleUse value. We conservatively use two greater, instead of one, since the stale counters only approximate the logarithm of staleness. These references are *candidates* for pruning, and we put them on a *candidate queue*.

2. The *stale transitive closure* starts with references in the candidate queue; these references' target objects are the roots of stale data structures. While it marks them live, it computes the bytes reachable from each root, i.e., the size of the stale data structure. The stale closure adds this value to bytesUsed for the edge entry for the candidate reference. The value computed depends on processing order since objects can be reachable from multiple stale roots. Fortunately the decision affects pruning accuracy but not correctness, and it is not a problem in practice.

At the end of this process, leak pruning iterates over each entry in the edge table, finding the entry with the greatest bytesUsed value, and it resets all bytesUsed values. The PRUNE state poisons stale instances of this edge type.

Example. Figure 7.5 shows the heap and an edge table for Figures 7.3 and 7.4 during SELECT. Each object is annotated with the value of its stale counter. The in-use closure adds the references marked **cand** to the candidate queue, but it does not add b2 → c2 since c2's stale counter is less than 2. It also does not add e1 → c4. Its stale counter must be at least 4, i.e., 2 more than maxStaleUse, which is 2 for E → C in this example. The stale closure processes the objects

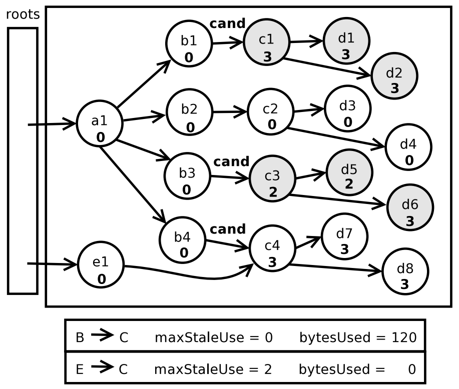


Figure 7.5: **Example heap during the SELECT state.**

reachable only from candidate references, which are shaded gray. Objects c4, d7, and d8 are processed by the *in-use closure* since they are reachable from non-candidate reference e1 → c4. If we suppose each object is 20 bytes, then bytesUsed for B → C is 120 bytes. We select this edge entry for pruning since it has the greatest value of bytesUsed.

7.2.4 The PRUNE State

During collection in PRUNE, the collector again divides the transitive closure into in-use and stale closures, but during the in-use closure it prunes all references that correspond to the selected edge type and whose target objects have staleness values that are at least two more than the entry's maxStaleUse. The collector poisons each reference in the candidate set by setting its *second-lowest* bit. The collector does not trace the reference's target. Future collections see the reference is poisoned and do not dereference it. The following pseudocode shows how the collector poisons references during the in-use closure:

```
while (!inUseQueue.isEmpty()) {
  ObjectReference object = inUseQueue.pop();
  for (slot : object's fields or array elements) {
    ObjectReference target = *slot; // dereference slot
    EdgeType et = getEdgeType(slot);
    if (target.staleCounter >= edgeTable[et].maxStaleUse + 2) {
      if (et == selectedEdgeType) {
        slot |= 0x2; // set second-lowest bit of slot
      } else {
        candidateQueue.push(slot); // defer to stale closure
      }
    } else () {
      slot.store(traceObject(target));
      // Note: traceObject(target) calls
      // inUseQueue.push(target)
      // if it is the first to trace target.
    }
  }
}
```

7.2.5 Intercepting Accesses to Pruned References

In order to intercept program accesses to pruned references, we overload the read barrier's conditional to check the two lowest bits:

```
b = a.f;          // Application code
if (b & 0x3) {    // Check two lowest bits
  // out-of-line cold path
  if (b & 0x2) {  // Check if pruned
    InternalError err = new InternalError();
    err.initCause(avertedOOME);
    throw err;
  }
  /* rest of read barrier */
}
```

The read barrier body checks for a poisoned reference by examining the second-lowest bit. If the bit is set, the barrier throws an InternalError. To help users

and developers, it attaches the original OutOfMemoryError that *would* have been thrown earlier.

7.2.6 Concurrency and Thread Safety

Our implementation supports multi-threaded programs executing on multiple processors. Previously we discussed how atomic updates in the read barrier preserve thread safety. The edge table is a global structure that can be updated by multiple threads in the read barrier or during collection. We need global synchronization only on the edge table when adding a new edge type, which is rare. We never delete an edge table entry. When updating an entry's data, we could use fine-grained synchronization to protect each entry. Since we expect conflicts to be rare, and bytesUsed and maxStaleUse are parameters to an algorithm whose result does not affect program correctness, we do not synchronize their updates.

By default, the garbage collector is parallel [BCM04]. It uses multiple collector threads to traverse all reachable objects. The implementation uses a shared pool from which threads obtain local work queues to minimize synchronization and load-balance. Because many objects have multiple references to them, the collector prevents more than one thread from processing an object with fine-grained synchronization on the object. We piggyback on these mechanisms to implement the in-use and stale transitive closures. In the stale closure, a single thread processes all objects reachable from a candidate edge. The stale closure is parallel since multiple collector threads can process the closures of distinct candidates simultaneously.

7.3 Performance of Leak Pruning

This section measures the overhead leak pruning adds to observe and select references for pruning. We show leak pruning adds little or no overhead if the program is not leaking. When tolerating leaks, it adds modest overhead to select and prune references.

7.3.1 Application Overhead

Leak pruning adds overhead because it inserts read barriers into application code and tracks staleness and selects references to prune during garbage collection. Figure 7.6 shows application and collection times without compilation. Execution times are normalized to *Base*. Each bar is the mean of five trials; the error bars show the range across trials. Replay compilation keeps variability low in all cases except for bloat. The sub-bars at the bottom are the fraction of time spent in collection.

The *Barriers* configuration includes only leak pruning's read barriers. Similar to Melt, our implementation uses all-the-time barriers, which add 6% on average, but a future implementation could wait to add barriers until the OBSERVE state. Since the inlined parts of their read barriers are virtually the same, leak pruning adds the same compilation and code size overheads as Melt.

Observe shows the overhead of the OBSERVE state. For these experiments, we force the OBSERVE state all the time. This configuration adds updating of each object's staleness header bits during collection and updating of maxStaleUse for edge types that become stale but the program later uses. Similarly, the *Select* configuration represents the overhead of always being in the SELECT state: performing the stale trace, updates to bytesUsed in the edge

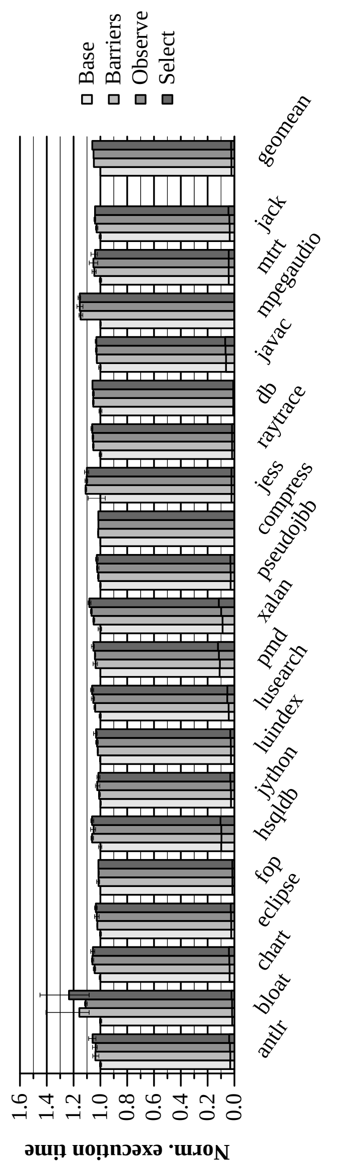


Figure 7.6: **Application execution time overhead of leak pruning.** Sub-bars are GC time.

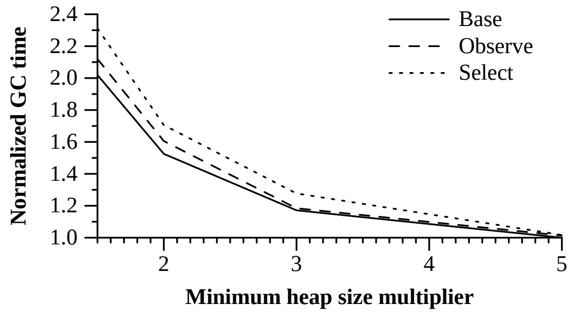


Figure 7.7: **Normalized collection time for leak pruning across heap sizes.** Y-axis starts at 1.0.

table, and selection of an edge type to prune. These configurations add no noticeable overhead since they mainly add collection overhead, which accounts for 2% of overall execution time on average.

7.3.2 Garbage Collection Overhead

Figure 7.7 plots the geometric mean of normalized GC time over all the benchmarks as a function of heap sizes ranging from 1.5 to 5 times the minimum heap size in which each benchmark executes. The smaller the heap size, the more often the program exhausts memory and invokes the collector. Observe adds up to 5% overhead to mark the lowest bit of references and update staleness. Selecting references to prune every collections adds 9% more, for a total of 14%. Since collection adds only modestly to total time, this overhead does not add appreciably to overall application time. The read barriers incur the majority of overall program overhead.

Leak	**Leak pruning's effect**	**Reason**
EclipseCP	Runs >100X longer	All reclaimed?
EclipseDiff	Runs >200X longer	Almost all reclaimed
ListLeak	Runs indefinitely	All reclaimed
SwapLeak	Runs indefinitely	All reclaimed
MySQL	Runs 35X longer	Most reclaimed
JbbMod	Runs 21X longer	Most reclaimed
SPECjbb2000	Runs 4.7X longer	Some reclaimed
Mckoi	Runs 1.6X longer	Some reclaimed
DualLeak	No help	None reclaimed
Delaunay	No help	Short-running

Table 7.1: **Ten leaks and leak pruning's effect on them.**

7.4 Effectiveness of Leak Pruning

Table 7.1 summarizes leak pruning's effectiveness of the same 10 leaks evaluated in the previous chapter. Leak pruning performs similarly to Melt in many cases: both approaches execute the top four leaks over 24 hours and possibly indefinitely (Melt's lifetime is limited by available disk space).

Leak pruning extends MySQL's lifetime significantly since it correctly prunes dead objects reachable from Statement objects without pruning Statement objects, which are periodically accessed during hash table re-hashing. Melt also extends MySQL's lifetime, but its less accurate prediction algorithm leads to overhead because it moves Statement objects to disk and then later activates them.

Melt and leak pruning both fail to provide much help SPECjbb2000 and DualLeak since much of their heap growth is live, and Mckoi because of implementation limitations. Melt provides limited help to Delaunay but incurs high activation overhead, whereas leak pruning does not help at all since Delaunay does not run long enough for leak pruning to make good decisions

Finally, Melt seems to run JbbMod until disk exhaustion, whereas leak pruning runs it 21 times longer. Leak pruning fails sooner because either (1) it fails to identify and prune some small fraction of the leak, or (2) it incorrectly prunes a reference that is later used.

A best-of-both-worlds approach could store leaks to disk and prune leaks to avoid running out of disk space. Our implementation handles the most challenging case: identifying and pruning leaks without falling back on the disk to increase the time until heap exhaustion.

We execute each program in a heap chosen to be about twice the size needed to run the program if it did not leak. We have evaluated leak pruning with four heap sizes for each leak (data omitted for brevity) [BM09] and found leak pruning's effectiveness is generally not sensitive to maximum heap size, except it sometimes fails to identify and prune the correct references in tight heaps, since it has little time to do so.

7.4.1 EclipseDiff

We omit detailed results for each leaky program since leak pruning's effect on program reliability and performance is similar to Melt's. Here we describe the results for EclipseDiff, which leaks large subtrees of difference results whose roots are ResourceCompareInput objects. These root objects are in use, but the result of the subtree is dead. Leak pruning selects and prunes several edge types with source type ResourceCompareInput. With leak pruning, EclipseDiff should eventually exhaust memory since some heap growth is live, but the subtree rooted at each ResourceCompareInput is comparatively much larger, so leak pruning turns a fast-growing leak into a very slow-growing leak. We run

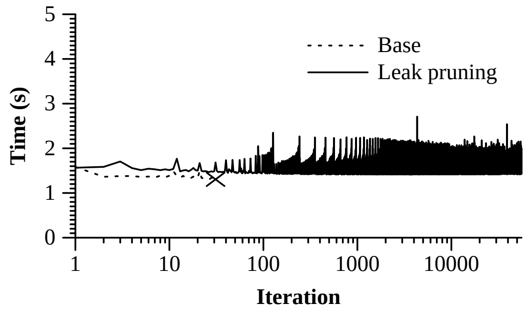


Figure 7.8: **Time per iteration for EclipseDiff.** X-axis is logarithmic.

EclipseDiff with leak pruning for 24 hours, and it does not run out of memory.

Figure 7.1 (page 139) shows reachable memory in the heap with and without leak pruning for its first 2,000 iterations. Figure 7.8 plots time for each iteration for all 55,780 iterations, using a logarithmic x-axis. Selection and pruning during GC extend some iterations up to two times because unmodified GC adds significant overhead to iterations interrupted by GC, and selection and pruning add additional overhead. However, long-term throughput stays consistent.

7.5 Accuracy and Sensitivity

This section examines implementation considerations. It shows that (1) the prediction mechanism in leak pruning is more accurate than just staleness or ignoring data structure sizes, (2) its space overhead is small, and (3) completely exhausting memory before pruning can initially degrade performance significantly.

7.5.1 Alternative Prediction Algorithms

This section evaluates whether our prediction algorithm's complexity is merited, by comparing it to two simpler alternatives:

Most stale. In the SELECT state, this algorithm identifies the highest staleness level of any object. In the DELETE state, it prunes all references to every object with this staleness level. This algorithm is the same as Melt's algorithm, which simply predicts that highly stale objects are leaks. Other approaches that move objects to disk use the same predictor [BGH+07, GSW07, TGQ08]. LeakSurvivor also avoids moving previously activated objects back to disk.

Individual references. This algorithm is similar to our default, except that it elides the stale transitive closure. In SELECT, it updates the edge table for *every* reference whose target object's stale counter is at least 2 greater than maxStaleUse. Each update simply adds the size of the target object to the edge type's bytesUsed value. Thus, this algorithm identifies individual leaked references, not leaked data structures.

The middle columns of Table 7.2 show the effectiveness of these prediction algorithms measured in iterations. For example, EclipseCP with *Indiv refs* terminates after 41 iterations because the algorithm selects and prunes highly stale String → char[] references. The program later tries to use one of these references. In contrast, our default algorithm prunes reference types org.eclipse.jface.text.-DefaultUndoManager$TextCommand → String and org.eclipse.jface.text.Document-Event → String, automatically reclaiming the growing, leaked Strings without accidentally deleting other live Strings. In general, our algorithm matches or

	Alternative selection algorithms				Edge
Leak	Base	Most stale	Indiv refs	Default	types
EclipseCP	11	10	41	≥1,115	2,203
EclipseDiff	259	228	3,380	≥55,780	1,817
ListLeak	110	108	Same →	≥2,788,755	56
SwapLeak	5	5	11	≥11,368	83
MySQL	18	35	114	634	230
JbbMod	204	41	911	4,267	209
SPECjbb2000	135	97	625	632	197
Mckoi	44	47	71	72	308
DualLeak	145	149	144	143	69

Table 7.2: **Effectiveness of several selection algorithms.** The final column is space overhead from the edge table.

outperforms the others since (1) it considers references' types (unlike *Most stale*) and (2) it considers data structures (unlike *Individual references*, which tries to identify each leaked reference type).

7.5.2 Space Overhead

Our implementation adds space overhead to store information about edge types in the edge table. For simplicity, it uses a fixed-size table with 16K slots using closed hashing [CLRS01]. Each slot has four words—source class, target class, maxStaleUse, and bytesUsed—for a total of 256K. A production implementation could size the table dynamically according to the number of edge types. The last column of Table 7.2 shows the number of edge types used by our implementation when it runs each leak. We simply measure the number of edge types in the edge table at the end of the run, since we never remove an edge type from the table. Eclipse is complex and uses a few thousand edge types; the database and JBB leaks are real programs but less complex and store hundreds of types; and the microbenchmark leaks store fewer than 100

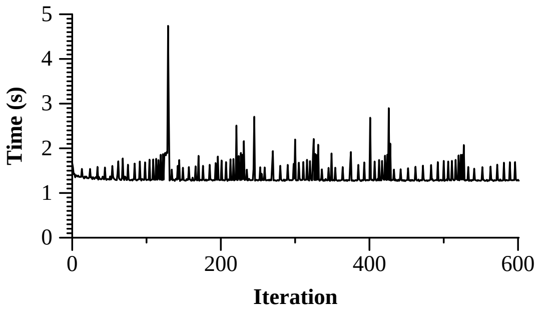


Figure 7.9: **Time per iteration for EclipseDiff when it must exhaust memory prior to pruning.**

edge types.

7.5.3 Full Heap Threshold

By default, our implementation starts pruning references when the heap is 90% full (Section 7.1.2). However, the user can specify that leak pruning wait to prune references until the heap is 100% full, i.e., when it is just about to throw an out-of-memory error. Figure 7.9 shows the throughput of EclipseDiff for its first 600 iterations using a 100% heap fullness threshold. The first spike, at about 125 iterations, occurs because Eclipse slows significantly as GCs become very frequent: each GC reclaims only a small fraction of memory, so the next GC occurs soon after. Later spikes are smaller because leak pruning prunes references when the heap is only 90% full (since the program has already exhausted memory once); some of the overhead is due to the overhead of selecting and pruning references. The spike is about 2.5X greater than the other spikes, which may be a reasonable trade-off in order to run the program as long as possible before commencing pruning.

7.6 Conclusion and Interpretation

Leak pruning closes the gap between reachability and liveness by predicting which objects constitute this gap and reclaiming them without changing semantics. It can tolerate unbounded memory with bounded resources in many cases, which is particularly compelling for embedded systems without disks, but can add value to any system. Our prediction algorithm performs almost as well as any algorithm could do: except in one case, leak pruning fails only due to live memory growth, which other semantics-preserving leak tolerance approaches cannot handle.

Melt and leak pruning are complementary approaches. Melt makes full use of system resources to provide strong availability guarantees, but it is limited by resource finiteness. Leak pruning conserves resources and has weaker availability guarantees. A combination of the two approaches would work better than either by using available disk space until full and then reclaiming objects.

Chapter 8

Related Work

Bug diagnosis falls into two general categories: static and dynamic analysis. Static analysis suffers from scalability issues and false positives, and it does not easily handle modern language features. Dynamic analysis finds bugs in real executions, but most approaches add high time and space overhead, so they are used in testing, where they miss behavior occurring only in deployed runs. Liblit presents a framework for bug detection in deployment [Lib04]. His statistical, invariant-based approach finds different bugs than our flow tracking-based approaches, and it requires multiple buggy instances whereas our approaches need only one. Most prior work on tolerating bugs focuses on nondeterministic errors. Prior work on tolerating leaks does not preserve semantics, offers less coverage, or does not guarantee time and memory usage remain proportional to live memory.

8.1 Static Analysis

By finding bugs in all possible executions, static analysis finds bugs without having to execute the buggy code. Unfortunately, static analysis tools often produce many false positives because they rely on coarse approximations of dynamic program behavior, since path- and context-sensitive analysis is often too expensive for large programs. Perhaps most problematic is pointer analysis, which even at its most powerful makes significant approximations for

heap objects [WL04]. Dillig et al. present a fully path- and context-sensitive data-flow analysis [DDA08], but it still produces false positives because its pointer analysis makes significant approximations such as conflating all heap objects into a single node. Shape analysis models the structure of the heap but has not been successfully applied to real, large programs.

Previous bug detection work includes a number of practical static analysis tools for detecting bugs. Pattern-based systems such as PMD are effective at identifying potential null dereferences but lack the dataflow analysis often needed to identify the reason for the bug [PMD]. FindBugs uses data-flow analysis to identify null dereferences and includes a notion of confidence to reduce the false positive rate, although this also results in false negatives [HP04, HSP05]. ESC/Java employs a theorem prover to check that pointer dereferences are non-null [FLL$^+$02]. Both FindBugs and ESC/Java are primarily intraprocedural and rely on user annotations to eliminate false positives due to method parameters and return values. JLint and Metal include an interprocedural component to track the states of input parameters [HCXE02, Jli]. Several of these detectors are made intentionally *unsound* to reduce the false positive rate. This choice allows code with bugs to pass silently through the system and fail at run time.

Prior work uses static analysis to find memory leaks in C and C++ programs but reports false positives since it makes conservative assumptions about control flow [CPR07, HL03]. Prior work identifies lost (unreachable and unfreed) objects, whereas we focus on leaks due to useless (reachable but dead) objects. Detecting useless objects statically seems inherently very challenging.

Our dynamic approaches complement static analyses. Because they make approximations with respect to context- and path-sensitivity and the heap, static analyses have trouble finding bugs that involve long sequences of control or data flow. Dynamic approaches can diagnose these more complex bugs that static tools miss. For example, *thin slicing* provides programmers with statements that affect value of interest, but within a limited scope in order to avoid reporting too many false positives [SFB07]. Thin slicing helps programmers find some bugs, but origin tracking can find the causes of the remaining bugs that involve arbitrarily long data and control flow. On balance, origin tracking's bug focus is narrower, and it identifies the origin only and not intervening flow.

Modern languages include features such as dynamic dispatch, reflection, and dynamic class loading that complicate static analysis since control flow is unclear before run time. In the case of dynamic class loading, the code may not even be available at analysis time.

8.2 Dynamic Analysis

Dynamic analysis, the dual of static analysis, finds bugs in real program executions. Testing runs do not exercise all program behavior that occurs in deployment, but most existing dynamic analyses are too expensive for deployed use.

8.2.1 Pre-Release Testing Tools

Valgrind [NS07] and Purify [HJ92] find errors such as memory corruption, memory leaks, and uses of undefined values in C and C++ programs [HJ92, NS07]. They add heavyweight instrumentation at every memory access, allocation, and free, and use conservative garbage collection to find lost objects. These tools have overheads from 2x to 20x, coupled with high per-

object space overhead. They are too expensive for production runs; they target testing runs and provide high accuracy and versatility.

8.2.2 Invariant-Based Bug Detection

Invariant-based bug detection (also called anomaly-based bug detection and statistical bug isolation) uses multiple program runs to identify program behavior features correlated with errors [EPG$^+$07, HL02, Lib04, LTQZ06, ZLF$^+$04]. Our work is complementary to these approaches since they have different usage models and can detect different types of bugs. Whereas invariant-based bug detection requires multiple executions with multiple inputs (AVIO can use the same inputs [LTQZ06]), our approaches require just one buggy execution. Our approach is thus applicable to software that has few running instances that fail infrequently (e.g., fighter jet software).

PCC could add low-overhead dynamic context sensitivity to existing invariant-based bug detection systems, which generally have not used context sensitivity because of its previous cost. Invariant-based bug detection already adds overhead too high for deployed use [EPG$^+$07, HL02], requires many users experiencing a bug to identify its cause [LNZ$^+$05], or relies on hardware to achieve low overhead [LTQZ06, ZLF$^+$04]. *Artemis* lowers the overhead of these tools by limiting profiling to infrequently-executed "contexts," which include input variables' values but not calling context [FM06]. PCC could efficiently add calling context to Artemis's context computation.

8.2.3 Tracking Control and Data Flow

Our bug diagnosis approaches track control and data flow and the relationship between them, which are fundamental to program understanding.

Dynamic program slicing tracks the statements that affect a value via control, data dependencies, or omitted statements [AH90, NM03, ZGG06, ZTGG07]. It adds high overheads (e.g., 10-100x slowdowns). Origin tracking and Bell limit which data they track but add overhead low enough for deployed systems.

TraceBack records a fixed amount of recent control flow (similar to dynamic slicing but without data flow) during execution and reports it in the event of a crash [ASM$^+$05]. In contrast, origin tracking provides the *data-flow* origin of the faulting value; and PCC summarizes new interprocedural control-flow throughout execution rather than just before the fault.

TaintCheck is a security tool that tracks which values are tainted (i.e., from untrusted sources) and detects if they are used in dangerous ways [NS05]. TaintCheck shadows each byte, recording for each tainted value: the system call from which it originated, a stack trace, and the original tainted value. Thus TaintCheck uses a form of explicit origin tracking that requires both extra space and time (extra operations are required to propagate the taint values). Value piggybacking would not be appropriate for TaintCheck because tainted values cannot have other values piggybacked onto them as they do not in general have spare bits.

8.2.4 Diagnosing Memory Leaks

Leak detectors for native languages find leaks by tracking heap reachability or application accesses to the heap. Prior leak detectors for managed languages infer leaks from dynamic heap growth. Our leak diagnosis and tolerance approaches predict leaks based on application accesses to the heap.

Leak Detection for Native Languages

Dynamic tools for C and C++ track allocations, heap updates, and frees to report unfreed objects [HJ92, MRB04, NS07] or track object accesses to report stale objects [CH04, QLZ05].

SWAT finds leaks in C and C++ programs by guessing that *stale* objects (objects not used in a while by the program) are leaks [CH04]. Our leak detection and tolerance approaches borrow SWAT's staleness approach to find leaks. SWAT adds several words of space overhead per object, while our leak detector uses Bell to save space but cannot report sites that do not leak many objects due to its statistical nature. For C programs that allocate and custom-manage large chunks of memory [BZM02], SWAT has low space overhead. On the C benchmark twolf, which allocates many small objects, SWAT adds 75% space overhead. Many native programs and most managed programs heap-allocate many small objects (24-32 bytes per object on average [DH99]), where Bell's space-efficient mechanism offers substantial space advantages.

SafeMem employs a novel use of error-correcting code (ECC) memory to monitor memory accesses in C programs, in order to find leaks and catch some types of memory corruption [QLZ05]. For efficiency, ECC memory monitors only a subset of objects, which SafeMem finds by grouping objects into types and using heuristics that identify potentially leaking types.

Leak Detection for Managed Languages

JRockit [Orab],JProbe [Que], LeakBot [MS03], Cork [JM07], and .NET Memory Profiler [Sci] are among the many tools that find memory leaks in Java and C# programs. These tools use heap growth and heap differencing to find objects that cause the heap to grow. JRockit provides low-overhead trend analysis, which reports growing types to the user. At the cost of more overhead, JRockit can track and report the instances and types that are pointing to growing types, as well as object allocation sites. LeakBot takes heap snapshots and uses an offline phase to compare the snapshots. It uses heuristics based on common leak paradigms to insert instrumentation at run time.

These tools use growth as a heuristic to find leaks, which may result in false positives (growing data structures or types that are not leaks) and false negatives (leaks that are not growing). Our approaches use staleness (time since last use) to find all memory the application is not using, although they cannot find live heap growth that the program is using. Our leak detector may report false positives if non-leaking memory is not used for a while, although these reports probably indicate poor memory usage.

8.2.5 Work Related to Bell

Per-object information. An alternative to Bell's statistical approach is to store *un-encoded* per-object information for a sample of objects (e.g., dynamic object sampling [JBM04]). Sampling avoids Bell encoding and decoding but still adds some space overhead and requires conditional instrumentation that checks whether an object is in the sampled set.

Instrumentation Optimization. Our leak detector uses data-flow analysis to find partially and fully redundant instrumentation at object uses, and it removes fully redundant instrumentation. The instrumentation at object uses (reads) is called a *read barrier* [BH04]. Prior work studies the overheads of a variety of read barriers and finds lightweight barriers can be cheap (5 to 8%

overhead on average), but more complex barriers are expensive (15 to 20% on average) [BCR03, BH04, Zor90]. Bacon et al. use common subexpression elimination to remove fully redundant read barriers, which reduces average overhead from 6 to 4% on PowerPC [BCR03]. Since our leak detector includes a load, store, and two multiplies, redundancy elimination still does not reduce its overhead to the levels in previous work.

Information theory and communication complexity. Bell encoding and decoding are related to concepts in information theory and communication complexity [CT91, KN96]. For example, a well-known idea in communication complexity is that two bit strings can share just one bit with each other to determine if they are the same string: they both hash against the same public key, and a non-match indicates they are different, while a match is inconclusive [KN96]. Extracting random bits from two weakly random input sources (Bell's encoding function) is a well-studied area in communication complexity [CG88]. We are not aware of any work that probabilistically encodes and decodes program behavior as Bell does.

8.2.6 Work Related to Probabilistic Calling Context

This section discusses prior work in calling context profiling. It first considers stack-walking, then heavyweight approaches that construct a calling context tree (CCT), and finally sampling-based approaches. We also consider related forms of profiling.

Walking the stack. One approach for identifying the current calling context is to walk the program stack, then look up the corresponding calling context identifier in a calling context tree (CCT) [NS07, SN05]. Unfortunately, walking the stack more than very infrequently is too expensive for production environments, as shown in Section 3.4.4.

Calling context tree. An alternative approach to walking the stack is to build a dynamic calling context tree (CCT) where each node in the CCT is a context, and maintain the current position in the CCT during execution [ABL97, Spi04]. This instrumentation slows C programs down by a factor of 2 to 4. The larger number of contexts in Java programs and the compile-time uncertainty of virtual dispatch further increase CCT time and space overheads. CCT nodes are 100 to 500 bytes in previous work, whereas PCC values are very compact in comparison, since each one only needs 32 or 64 bits, and storing them in a half-full hash table achieves good run-time performance, as shown in Section 3.4.4.

Sampling-based approaches. Sampling-based and truncation approaches keep overhead low by identifying the calling context infrequently [BM06a, FMCF05, HG03, Wha00, ZSCC06]. Clients use hot context information for optimizations such as context-sensitive inlining [HG03] and context-sensitive allocation sites for better object lifetime prediction and region-based allocation [ISF06, SZ98]. Hazelwood and Grove sample the stack periodically to collect contexts to drive context-sensitive inlining [HG03]. Zhuang et al. improve on sampling-based stack-walking by performing *bursty* profiling after walking the stack, since it is relatively cheap to update the current position in the CCT on each call and return for a short time [ZSCC06]. Bernat and Miller limit profiling to a subset of methods [BM06a]. Froyd et al. use unmodified binaries and achieve extremely low overhead through stack sampling [FMCF05].

Sampling is useful for identifying hot calling contexts, but it is not suitable for clients such as testing, security, and debugging because sampling sacrifices coverage, which is key for these clients.

Although PCC primarily targets clients requiring high coverage, it could potentially improve the accuracy-overhead trade-off of sampling-based approaches. Zhuang et al.'s calling context profiling approach avoids performing bursty sampling at already-sampled calling contexts [ZSCC06]. Currently they walk the stack to determine if the current context has been sampled before, but instead they could use PCC to quickly determine, with high probability, if they have already sampled a calling context.

Dynamic call graph profiling. Dynamic optimizers often profile call edges to construct a dynamic call graph (DCG) [AHR01, LRBM07, QH04, SYK$^+$01], which informs optimizations such as inlining. DCGs lack context sensitivity and thus provide less information than calling context profiles.

Path profiling. Ball-Larus path profiling computes a unique number on each possible path in the control flow graph [BL96]. An intriguing idea is applying path profiling instrumentation to the dynamic call graph and computing a unique number for each possible context. This approach is problematic because call graphs, which have thousands of nodes for our benchmarks, are typically much larger than control flow graphs (CFGs). The number of possible paths both through CFGs and call graphs is exponential in the size of the graph in practice, so the statically possible contexts cannot be assigned unique 32- or even 64-bit values. Other challenges include: (1) recursion, which leads to cyclic graphs; (2) dynamic class loading, which modifies the graph at run time; and (3) virtual dispatch, which obscures call targets and complicates call edge instrumentation. Wiedermann computes a unique number per context at run time by applying Ball-Larus path numbering to the call graph, but does not evaluate whether large programs can be numbered uniquely [Wie07]. His approach uses C programs, avoiding the challenges of dynamic class loading and virtual dispatch, and handles recursion by collapsing strongly-connected components in the call graph. Melski and Reps present *interprocedural path profiling* that captures both inter- and intraprocedural control flow, but their approach does not scale because it adds complex call edge instrumentation, and there are too many statically possible paths for nontrivial programs [MR99].

As Section 3.1 points out, much prior work uses path profiling to understand dynamic behavior in testing, debugging, and security, but dynamic object-oriented languages need calling context, too, since it captures important behavior. Paths and calling contexts are complementary since paths capture *intraprocedural* control flow while calling context provides *interprocedural* control flow. One could imagine combining PCC and path profiling for best-of-both-worlds approaches in residual testing, invariant-based bug detection, and anomaly-based intrusion detection.

8.3 Tolerating Bugs

Most prior work on tolerating bugs focuses on nondeterministic errors, including memory corruption errors sensitive to layout. A common response to failure is to restart the application, but this approach does not work in general, for example, for mission-critical or autonomous systems. Prior work tolerates memory leaks but does not preserve semantics or does not scale.

8.3.1 Avoiding Bugs

An attractive alternative to tolerating bugs is to provide language support to help programmers avoid code with bugs in the first place. Unfortunately most approaches require additional programmer effort and are not fool-proof.

Chalin and James propose extending Java with “never null” pointer types, which are the default, and requiring “possibly null” pointers to be annotated specially [CJ06]. This feature makes it harder to forget to initialize pointers, but as long as null pointers are possible, the problem can still occur.

HeapSafe helps C programmers by ensuring correct manual memory management with dynamic reference counting of heap objects [GEB07]. It also adds a language feature called *delayed free scopes*, during which dangling references can exist. The approach adds run-time overhead and helps manual memory management bugs only.

To help programmers avoid leaks and manage large heaps, the Java language definition provides *weak* and *soft* references. The collector always reclaims weakly-referenced objects, and it reclaims softly-referenced objects if the application experiences memory pressure [Goe05, Goe06]. Inserting soft and weak references adds to the development burden, and programmers may still cause a leak by forgetting to eliminate the last strong (not weak or soft) reference.

8.3.2 Tolerating General Bugs

DieHard, *Rx*, *Grace*, and *Atom-Aid* tolerate nondeterministic errors due to memory corruption and concurrency bugs [BYL$^+$08, BZ06, LDSC08, QTSZ05]. Rx rolls back to a previous state and tries to re-execute in a different environment. DieHard uses random memory allocation, padding, and redundancy to probabilistically decrease the likelihood of errors [BZ06, QTSZ05]. Grace and *Atom-Aid* tolerate some concurrency bugs by using block-atomic execution to reduce or eliminate the occurrence of buggy interleavings [BYL$^+$08, LDSC08].

Failure-oblivious computing tolerates general failure bugs. It ignores out-of-bound writes and manufactures values for out-of-bounds reads. Its approach could also tolerate failure bugs in type-safe languages, such as null pointer and array out-of-bounds errors, by simply ignoring them [RCD$^+$04].

8.3.3 Tolerating Memory Leaks

Prior work for tolerating memory leaks either does not preserve semantics, does not offer full coverage, or does not scale.

Tolerating Memory Pressure

Many VMs dynamically size the heap based on application behavior. For example, some approaches adaptively trigger GC or resize the heap in order to improve GC performance and program locality [CBC$^+$06, XSaJ07, YBKM06, YHB$^+$04].

When the application's heap size exceeds its working set size, *bookmarking collection* reduces collection overhead [HFB05]. It cooperates with the operating system to *bookmark* swapped-out pages by marking in-memory objects they reference as live. The garbage collector then never visits bookmarked pages. Bookmarking can compact the heap but cannot move objects referenced by bookmarked pages. It tracks staleness on page granularity. Melt

instead uses object granularity, grouping and isolating leaking objects.

Static liveness detection of GC roots can reduce the *drag* between when objects die and when they are collected [HDH02] but cannot deal with objects that are dead but reachable.

Leaks in Native Languages

Cyclic memory allocation tolerates leaks in C and C++ by limiting allocation sites to m live objects at a time [NR07]. Profiling runs determine m for each allocation site, and subsequent executions allocate into m-sized circular buffers. Cyclic memory allocation does not preserve semantics since the program is silently corrupted if it uses more than m objects, although failure-oblivious computing [RCD$^+$04] mitigates the effects in some cases.

Plug tolerates leaks in C and C++ with an allocator that segregates objects by age and allocation site, increasing the likelihood that leaked and in-use objects will reside on distinct pages [NBZ08]. Plug deals with later fragmentation via *virtual compaction*, which maps two or more virtual pages to the same physical page if the allocated slots on the pages do not overlap. Plug's approach helps native languages since objects cannot move, but collectors in managed languages can reorganize objects. In addition, segregating leaked and in-use objects is insufficient for managed languages since tracing collectors by default access the whole heap.

Leaks in Managed Languages

Like Melt, *LeakSurvivor* and *Panacea* tolerate leaks by transferring potentially leaked objects to disk [BGH$^+$07, GSW07, TGQ08]. They reclaim virtual and physical memory and modify the collector to avoid accessing objects moved to disk.

Panacea supports moving stale objects to disk [BGH$^+$07, GSW07]. The approach requires annotations for objects that can be moved to disk, and these objects must be serializable to get put on disk. Panacea does not scale for small, stale objects—which we find are frequent leak culprits—because it uses proxy objects for swapped-out objects. An advantage of Panacea is that it is implemented at the library level and needs no VM modifications.

LeakSurvivor [TGQ08] is the closest related work to Melt and is concurrent work with Melt. Both approaches free up virtual and physical memory by transferring highly stale objects to disk, and both preserve semantics by returning accessed disk objects to memory. Unlike Melt, LeakSurvivor cannot guarantee space and time proportional to in-use memory because references from stale to in-use objects continue to use space even if the in-use objects become stale. In particular, entries in LeakSurvivor's *Swap-Out Table* (SOT) (similar to Melt's scion table) cannot be eliminated if the target object moves to disk, since incoming pointers from disk are unknown. In contrast, Melt uses two levels of indirection, stub-scion pairs, to eliminate scions referencing objects later moved to the stale space (Section 6.1.2). For the three leaks evaluated in LeakSurvivor, the SOT grows only slightly, but it is unclear if they grow proportionally to the leak since the experiments are terminated after two hours, before the leaks would have filled the disk. Melt adds less overhead than LeakSurvivor to identify stale objects (6% vs. 21%) since LeakSurvivor accesses an object's header on each read, while Melt uses referenced-based conditional read barriers to avoid accessing object headers in the common case.

Melt, LeakSurvivor, and Panacea preserve semantics since they retrieve

objects from disk if the program accesses them. Since they retrieve objects from disk, the prediction mechanisms do not have to be perfect, just usually right. If the systems are too inaccurate, performance will suffer. All will eventually exhaust disk space and crash. Leak pruning, on the other hand, requires perfect prediction to defer a leak successfully. Consequently, it uses a more sophisticated algorithm for predicting dead objects (Section 7.5.1 compares leak pruning's prediction algorithm to the algorithm used in prior work that uses the disk). Leak pruning is potentially less tolerant of errors because it must throw an error if it makes a mistake, but we find this happens for only one leak. In its favor, leak pruning uses bounded resources, making it suitable when disk space runs out or no disk is available (e.g., embedded systems).

8.3.4 Related Work for Melt: Orthogonal Persistence and Distributed GC

Melt uses mechanisms that have been used in orthogonal persistence, distributed garbage collection, and other areas. Orthogonal persistence uses object faulting, pointer swizzling, and read barriers to support transparent storage of objects on disk [ADJ$^+$96, HC99, HM93, MBMZ00, ZBM01]. Pointer swizzling can also be used to support huge address spaces [Mos92, Wil91]. Our implementation uses swizzling to support a 64-bit disk space on a 32-bit platform. Read barriers are widely used in concurrent garbage collectors [BCR03, BH04, DLM$^+$78, HNCB99, PFPS07, Zor90]. Distributed collectors use stub-scion pairs for references between machines [Pla94]. We use stub-scion pairs to support references from stale to in-use objects. Although Melt borrows existing mechanisms, previous work does not combine these mechanisms in the same way as Melt, i.e., to identify, isolate, and activate stale memory.

In complex programs, dynamic analysis finds bugs that static analysis misses since it approximates at data and control flow merge points. Most dynamic analysis approaches are only suitable for testing time since they add high overhead, whereas ours have low enough overhead for deployed use. Most prior work on tolerating bugs targets failures due to nondeterministic bugs and memory bugs. Unlike previous approaches, our leak tolerance approaches preserve semantics and guarantee time and memory usage proportional to in-use memory.

Chapter 9

Conclusion

This dissertation concludes with a summary of the work presented and a discussion of future work.

9.1 Summary

Deployed software contains bugs because complexity makes it hard to write correct code and to fully validate or exhaustively test software prior to deployment. These bugs cause software to fail, wasting billions of dollars and sometimes threatening human life. The uncertainty of software is impeding a future where society can reap benefits from increased reliance on software systems.

We argue that deployment is an ideal and necessary environment for both diagnosing and tolerating bugs. We show that unlike most prior approaches, ours add modest overhead, and they help developers find and fix bugs and help users deal with buggy software. Our diagnosis approaches track control and data flow throughout the program, in order to report culprit code and data if the program fails. This information helps programmers understand the program operations directly responsible for the failure. These approaches rely on novel insights into how to track control and data flow efficiently and usefully. Our leak tolerance approaches help several programs run indefinitely or significantly longer by removing likely leaked objects from the working set of the application and garbage collector. They help close the gap between liveness and reachability by predicting dead objects while preserving semantics.

Several of these approaches are ready for immediate adoption in deployed systems, where they would help developers and users with existing bugs. More broadly, the efficiency and effectiveness of our approaches demonstrate the viability of diagnosing and tolerating bugs at deployment time. They point to a future where developers routinely use powerful approaches to find and fix elusive bugs occurring only at deployment time, and users eagerly use automatic tolerance approaches to help them get the most out of buggy software.

9.2 Future Work

We have demonstrated the potential of using PCC to add dynamic context sensitivity to analyses that detect new or anomalous behavior, such as residual testing, invariant-based bug detection, and anomaly-based intrusion detection (Section 3.1). Future work could demonstrate the viability of using PCC for these clients and quantify the usefulness of context sensitivity for improving these applications. We are currently working toward using PCC to help detect intrusions that involve anomalous contexts. We are also developing an approach for reconstructing calling contexts from PCC values (PCC values as presented here are not reversible). This new approach would add context sensitivity to any dynamic analysis, including Bell and origin tracking.

Section 4.4.5 discusses several opportunities for future work on Bell. They include lower-overhead read barriers modeled on our leak tolerance barriers; considering type compatibility between sites and objects during decoding

to improve accuracy and decrease decoding time; and demonstrating Bell in a generational mark-sweep collector.

Melt and leak pruning are complementary approaches that should be combined. Melt provides availability guarantees, but growing leaks eventually fill the disk. When this happens, leak pruning could reclaim some on-disk objects. With more time to observe program behavior, leak pruning is likely to make better decisions. Melt simply predicts that highly stale objects are leaks, which incurs high activation overhead if many highly stale objects are live. Melt could benefit from leak pruning's more precise prediction algorithm to identify highly stale objects that are likely to be used again.

Our leak tolerance approaches handle leaks due to dead, not live, objects. If the program continues to access leaked objects, moving them to disk or collecting them is not possible. In several cases, programs keep leaked objects alive inadvertently when the underlying data structures access the objects. For example, two programs from Chapters 6 and 7 use a hash table containing leaked objects that the programs do not access again. However, when the hash table grows, it re-hashes and thus accesses all its objects. We propose the design of *leak-tolerant data structures* that avoid accessing objects that the program is not using. A leak-tolerant hash table could store stale entries separately from other entries (e.g., in a linked list). To preserve semantics, `get()` and `put()` operations need to check for equality with other key objects in the hash table. However, the VM could avoid equality checks in certain cases. If the `equals()` method is the default `Object.equals()`, then the VM knows based on the object's address whether it is on disk or has been collected automatically. The VM could handle other cases by summarizing the stale objects and symbolically executing `equals()` on an over-approximated, abstract representation of all stale objects in the hash table.

Bibliography

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, 1997.

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[AFG+00] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[AH90] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *ACM Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[AH02] Taweesup Apiwattanapong and Mary Jean Harrold. Selective Path Profiling. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, 2002.

[AHR01] Matthew Arnold, Michael Hind, and Barbara G. Ryder. An Empirical Study of Selective Optimization. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 49–67, London, UK, 2001. Springer-Verlag.

[All01] Eric Allen. Diagnosing Java Code: The Dangling Composite bug pattern. `http://www-128.ibm.com/developerworks/java/library/j-diag2/`, 2001.

[Apa] Apache Software Foundation. Commons-Math: The Apache Commons Mathematics Library. `http://commons.apache.org/math/`.

[ASM+05] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow. In *ACM Conference on Programming Language Design and Implementation*, pages 201–212, 2005.

[Bal01] T. Ball. The SLAM Toolkit: Debugging System Software via Static Analysis, 2001.

[BC94] Preston Briggs and Keith D. Cooper. Effective Partial Redundancy Elimination. In *ACM Conference on Programming Language Design and Implementation*, pages 159–170, 1994.

[BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ACM International Conference on Software Engineering*, pages 137–146, 2004.

[BCR03] D. Bacon, P. Cheng, and V. Rajan. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *ACM Symposium on Principles of Programming Languages*, pages 285–298, 2003.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[BGH+07] David Breitgand, Maayan Goldstein, Ealan Henis, Onn Shehory, and Yaron Weinsberg. PANACEA–Towards a Self-Healing Development Framework. In *Integrated Network Management*, pages 169–178, 2007.

[BH04] Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or Foe? In *ACM International Symposium on Memory Management*, pages 143–151, 2004.

[Bin97] D. Binkley. Semantics Guided Regression Test Cost Reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[BL96] Thomas Ball and James R. Larus. Efficient Path Profiling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, 1996.

[BM06a] Andrew R. Bernat and Barton P. Miller. Incremental Call-Path Profiling. *Concurrency and Computation: Practice and Experience*, 2006.

[BM06b] Michael D. Bond and Kathryn S. McKinley. Bell: Bit-Encoding Online Memory Leak Detection. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, 2006.

[BM07] Michael D. Bond and Kathryn S. McKinley. Probabilistic Calling Context. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–112, 2007.

[BM08] Michael D. Bond and Kathryn S. McKinley. Tolerating Memory Leaks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 109–126, 2008.

[BM09] Michael D. Bond and Kathryn S. McKinley. Leak Pruning. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009. To appear.

[BNK$^+$07] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 405–422, 2007.

[BYL$^+$08] Emery D. Berger, Ting Yang, Tongping Liu, Divya Krishnan, and Gene Novark. Grace: Safe and Efficient Concurrent Programming. Technical Report UM-CS-2008-017, University of Massachusetts, 2008.

[BZ06] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[BZM02] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering Custom Memory Allocation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, 2002.

[CBC$^+$06] W. Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided Proactive Garbage Collection for Locality Optimization. In *ACM Conference on Programming Language Design and Implementation*, pages 332–340, 2006.

[CG88] Benny Chor and Oded Goldreich. Unbiased Bits from Sources of Weak Randomness and Probabilistic Communication Complexity. *SIAM J. Comput.*, 17(2):230–261, 1988.

[CG06a] A. Chakrabarti and P. Godefroid. Software Partitioning for Effective Automated Unit Testing. In *ACM & IEEE International Conference on Embedded Software*, pages 262–271, 2006.

[CG06b] Trishul M. Chilimbi and Vinod Ganapathy. HeapMD: Identifying Heap-based Bugs using Anomaly Detection. In *ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 219–228, 2006.

[CH04] Trishul M. Chilimbi and Matthias Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[CJ06] Patrice Chalin and Perry James. Non-null references by default in java: Alleviating the nullity annotation burden. Technical Report 2006-003, Concordia University, 2006.

[CKV$^+$03] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap Compression for Memory-Constrained Java Environments. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 282–301, 2003.

[Cli08] Cliff Click. Blog entry, January 2008. `http://blogs.azulsystems.com/cliff/2008/01/adding-transact.html`.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[CPR07] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical Memory Leak Detection using Guarded Value-Flow Analysis. In *ACM Conference on Programming Language Design and Implementation*, pages 480–491, 2007.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley & Sons, 1991.

[CTW05] Cliff Click, Gil Tene, and Michael Wolf. The Pauseless GC Algorithm. In *ACM/USENIX International Conference on Virtual Execution Environments*, pages 46–56, 2005.

[DaC] DaCapo Benchmark Regression Tests. `http://jikesrvm.anu.edu.au/~dacapo/`.

[DDA08] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, Complete and Scalable Path-Sensitive Analysis. In *ACM Conference on Programming Language Design and Implementation*, pages 270–280, 2008.

[DH99] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *European Conference on Object-Oriented Programming*, pages 92–115, 1999.

[DLM$^+$78] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[Ecla] Eclipse Bug System Home Page. `http://www.eclipse.org/bugs/`.

[Eclb] Eclipse.org Home. `http://www.eclipse.org/`.

[EPG$^+$07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao.

The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[FKF$^+$03] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *IEEE Symposium on Security and Privacy*, page 62. IEEE Computer Society, 2003.

[FL05] J. Fenn and A. Linden. *Hype Cycle Special Report for 2005.* Gartner Group, 2005.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[FM06] Long Fei and Samuel P. Midkiff. Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies. In *ACM Conference on Programming Language Design and Implementation*, pages 84–95, 2006.

[FMCF05] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In *ACM International Conference on Supercomputing*, pages 81–90, 2005.

[GEB07] David Gay, Rob Ennals, and Eric Brewer. Safe Manual Memory Management. In *ACM International Symposium on Memory Management*, pages 2–14, 2007.

[GKS90] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized Incremental Construction of Delaunay and Voronoi Diagrams. In *Colloquium on Automata, Languages and Programming*, pages 414–431, 1990.

[GKS$^+$04] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark G. Stoodley, and Vijay Sundaresan. Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Virtual Machine Research and Technology Symposium*, pages 151–162, 2004.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[Goe05] Brian Goetz. Plugging memory leaks with weak references, 2005. `http://www-128.ibm.com/developerworks/java/library/j-jtp11225/`.

[Goe06] Brian Goetz. Plugging memory leaks with soft references, 2006. `http://www-128.ibm.com/developerworks/java/library/j-jtp01246.html`.

[GP05] Satish Chandra Gupta and Rajeev Palanki. Java memory leaks – Catch me if you can, 2005. `http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/index.html`.

[Gro00] William Gropp. Runtime Checking of Datatype Signatures in MPI. In *European PVM/MPI Users' Group Meeting on Recent*

Advances in Parallel Virtual Machine and Message Passing Interface, pages 160–167, London, UK, 2000. Springer-Verlag.

[GSW07] Maayan Goldstein, Onn Shehory, and Yaron Weinsberg. Can Self-Healing Software Cope With Loitering? In *International Workshop on Software Quality Assurance*, pages 1–8, 2007.

[HBM$^+$04] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The Garbage Collection Advantage: Improving Program Locality. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80, 2004.

[HC99] Antony L. Hosking and Jiawan Chen. PM3: An Orthogonal Persistent Systems Programming Language – Design, Implementation, Performance. In *International Conference on Very Large Data Bases*, pages 587–598, 1999.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-Specific, Static Analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[HDH02] 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*, 24(6):593–624, 2002.

[Her08] Matthew Hertz, 2008. Personal communication.

[HFB05] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage Collection without Paging. In *ACM Conference on Programming Language Design and Implementation*, pages 143–153, 2005.

[HG03] Kim Hazelwood and David Grove. Adaptive Online Context-Sensitive Inlining. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 253–264, 2003.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Winter USENIX Conference*, pages 125–136, 1992.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ACM International Conference on Software Engineering*, pages 291–301, 2002.

[HL03] David L. Heine and Monica S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *ACM Conference on Programming Language Design and Implementation*, pages 168–181, 2003.

[HM93] Antony L. Hosking and J. Eliot B. Moss. Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 288–303, 1993.

[HNCB99] Antony L. Hosking, Nathaniel Nystrom, Quintin I. Cutts, and Kumar Brahnmath. Optimizing the Read and Write Barriers for

Orthogonal Persistence. In *International Workshop on Persistent Object Systems*, pages 149–159, 1999.

[HP04] David Hovemeyer and William Pugh. Finding Bugs is Easy. In *Companion to ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–136, 2004.

[HRD$^+$07] Jungwoo Ha, Christopher J. Rossbach, Jason V. Davis, Indrajit Roy, Hany E. Ramadan, Donald E. Porter, David L. Chen, and Emmett Witchel. Improved Error Reporting for Software that Uses Black Box Components. In *ACM Conference on Programming Language Design and Implementation*, pages 101–111, 2007.

[HRS$^+$00] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults. *Software Testing, Verification & Reliability*, 10(3):171–194, 2000.

[HSP05] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 13–19, 2005.

[IF02] Hajime Inoue and Stephanie Forrest. Anomaly Intrusion Detection in Dynamic Execution Environments. In *Workshop on New Security Paradigms*, pages 52–60, 2002.

[Ino05] Hajime Inoue. *Anomaly Detection in Dynamic Execution Environments*. PhD thesis, University of New Mexico, 2005.

[ISF06] H. Inoue, D. Stefanović, and S. Forrest. On the Prediction of Java Object Liftimes. *ACM Transactions on Computer Systems*, 55(7):880–892, 2006.

[JBM04] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic Object Sampling for Pretenuring. In *ACM International Symposium on Memory Management*, pages 152–162, 2004.

[Jika] Jikes RVM. `http://www.jikesrvm.org`.

[Jikb] Jikes RVM Research Archive. `http://www.jikesrvm.org/Research+Archive`.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[Jli] Jlint. `http://jlint.sourceforge.net`.

[JM07] Maria Jump and Kathryn S. McKinley. Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages. In *ACM Symposium on Principles of Programming Languages*, pages 31–38, 2007.

[JR08] Richard E. Jones and Chris Ryder. A Study of Java Object Demographics. In *ACM International Symposium on Memory Management*, pages 121–130, 2008.

[JTM07] Daniel Jackson, Martyn Thomas, and Lynette I. Millett, editors. *Software for Dependable Systems: Sufficient Evidence?* National Research Council, The National Academies Press, 2007.

[KB08] Stephen W. Kent and Michael D. Bond. Bad Apples Suite, 2008. `http://www.cs.utexas.edu/~mikebond/papers.html#bad-apples-suite`.

[KLP88] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: An Interpreter-Based Programming Environment for the C Language. In *Summer USENIX Conference*, pages 161–71, 1988.

[KN96] Eyal Kushilevitz and Noam Nisan. *Communication Complexity.* Cambridge University Press, 1996.

[LBFD05] Julien Langou, George Bosilca, Graham Fagg, and Jack Dongarra. Hash Functions for Datatype Signatures in MPI. In *European Parallel Virtual Machine and Message Passing Interface Conference*, pages 76–83, 2005.

[LDSC08] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ACM/IEEE International Symposium on Computer Architecture*, pages 277–288, 2008.

[Lib04] Benjamin Robert Liblit. *Cooperative Bug Isolation.* PhD thesis, University of California at Berkeley, 2004.

[LNZ+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *ACM Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[LRBM07] Byeongcheol Lee, Kevin Resnick, Michael D. Bond, and Kathryn S. McKinley. Correcting the Dynamic Call Graph Using

Control Flow Constraints. In *International Conference on Compiler Construction*, pages 80–95, 2007.

[LTQZ06] Shan Lu, Joe Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[LY99a] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Prentice Hall PTR, 2nd edition, 1999.

[LY99b] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Edition).* Prentice Hall PTR, 1999.

[LYY+05] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining Behavior Graphs for Backtrace of Noncrashing Bugs. In *SIAM International Converence on Data Mining*, pages 286–297, 2005.

[MBMZ00] Alonso Marquez, Stephen M Blackburn, Gavin Mercer, and John Zigman. Implementing Orthogonally Persistent Java. In *International Workshop on Persistent Object Systems*, pages 247–261, 2000.

[Mck] Mckoi SQL Database. `http://www.mckoi.com/database/`.

[Mck02] Mckoi SQL Database message board: memory/thread leak with Mckoi 0.93 in embedded mode, 2002. `http://www.mckoi.com/database/mail/subject.jsp?id=2172`.

[Mos92] J. Eliot B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Computers*, 18(8):657–673, 1992.

[MR99] David Melski and Thomas Reps. Interprocedural Path Profiling. In *International Conference on Compiler Construction*, pages 47–62, 1999.

[MRB04] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. Precise Detection of Memory Leaks. In *International Workshop on Dynamic Analysis*, pages 25–31, 2004.

[MS03] Nick Mitchell and Gary Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming*, pages 351–377, 2003.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[Mye79] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[Nat02] National Institute of Standards and Technology. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3, 2002.

[NBZ08] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Plug: Automatically Tolerating Memory Leaks in C and C++ Applications. Technical Report UM-CS-2008-009, University of Massachusetts, 2008.

[NM03] Nicholas Nethercote and Alan Mycroft. Redux: A Dynamic Dataflow Tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[NR07] Huu Hai Nguyen and Martin Rinard. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ACM International Symposium on Memory Management*, pages 15–29, 2007.

[NS05] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium*, 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[OOK$^+$06] Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. Replay Compilation: Improving Debuggability of a Just-in-Time Compiler. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 241–252, 2006.

[Oraa] Oracle. JRockit. `http://www.oracle.com/technology/products/jrockit/`.

[Orab] Oracle. JRockit Mission Control. `http://www.oracle.com/technology/products/jrockit/missioncontrol/`.

[PFPS07] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A Real-Time Garbage Collector for Multiprocessors. In *ACM International Symposium on Memory Management*, pages 159–172, 2007.

[Pla94] David Plainfossé. *Distributed Garbage Collection and Reference Management in the Soul Object Support System.* PhD thesis, Université Paris-6, Pierre-et-Marie-Curie, 1994.

[PMD] PMD. `http://pmd.sourceforge.net`.

[PY99] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *ACM International Conference on Software Engineering*, pages 277–284, 1999.

[QH04] Feng Qian and Laurie Hendren. Towards Dynamic Interprocedural Analysis in JVMs. In *USENIX Symposium on Virtual Machine Research and Technology*, pages 139–150, 2004.

[QLZ05] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *International Symposium on High-Performance Computer Architecture*, pages 291–302, 2005.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In *ACM Symposium on Operating Systems Principles*, pages 235–248, 2005.

[Que] Quest. JProbe Memory Debugger. `http://www.quest.com/jprobe/debugger.asp`.

[RCD$^+$04] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee. Enhancing Server Availability and Security through Failure-Oblivious Computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 303–316, 2004.

[RKS05] A. Rountev, S. Kagan, and J. Sawin. Coverage Criteria for Testing of Object Interactions in Sequence Diagrams. In *Fundamental Approaches to Software Engineering*, LNCS 3442, pages 282–297, 2005.

[Sci] SciTech Software. .NET Memory Profiler. `http://www.scitech.se/memprofiler/`.

[Sco98] D. Scott. *Assessing the Costs of Application Downtime.* Gartner Group, 1998.

[SFB07] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin Slicing. In *ACM Conference on Programming Language Design and Implementation*, pages 112–122, 2007.

[SK06] Sunil Soman and Chandra Krintz. Efficient and General On-Stack Replacement for Aggressive Program Specialization. In *International Conference on Programming Languages and Compilers*, 2006.

[SMB04] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC^2: High-Performance Garbage Collection for Memory-Constrained Environments. In *ACM Conference on*

Object-Oriented Programming, Systems, Languages, and Applications, pages 81–98, 2004.

[SN05] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference*, pages 17–30, 2005.

[Sou] SourceForge.net. `http://www.sourceforge.net/`.

[Spi04] J. M. Spivey. Fast, Accurate Call Graph Profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[Sta99] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.

[Sta01] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[Sun03a] Sun Developer Network Forum. Java Programming [Archive] - garbage collection dilema (sic), 2003. `http://forum.java.sun.com/thread.jspa?threadID=446934`.

[Sun03b] Sun Developer Network Forum. Reflections & Reference Objects - Java memory leak example, 2003. `http://forum.java.sun.com/thread.jspa?threadID=456545`.

[SYK$^+$01] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 180–195, 2001.

[SZ98] Matthew L. Seidl and Benjamin G. Zorn. Segregating Heap Objects by Reference Behavior and Lifetime. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1998.

[TGQ08] Yan Tang, Qi Gao, and Feng Qin. LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages. In *USENIX Annual Technical Conference*, pages 307–320, 2008.

[TIO07] TIOBE Software. TIOBE programming community index, 2007. `http://tiobe.com.tpci.html`.

[VNC07] Kapil Vaswani, Aditya V. Nori, and Trishul M. Chilimbi. Preferential Path Profiling: Compactly Numbering Interesting Paths. In *ACM Symposium on Principles of Programming Languages*, pages 351–362, 2007.

[Wha00] John Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *ACM Conference on Java Grande*, pages 78–87. ACM Press, 2000.

[Wie07] Ben Wiedermann. Know your Place: Selectively Executing Statements Based on Context. Technical Report TR-07-38, University of Texas at Austin, 2007.

[Wil91] Paul R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *ACM SIGARCH Comput. Archit. News*, 19(4):6–13, 1991.

[WL04] John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision

Diagrams. In *ACM Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[WM07] Joan D. Winston and Lynette I. Millett, editors. *Summary of a Workshop on Software-Intensive Systems and Uncertainty at Scale*. National Research Council, The National Academies Press, 2007.

[WS02] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *ACM Conference on Computer and Communications Security*, pages 255–264. ACM Press, 2002.

[XSaJ07] Feng Xian, Witawas Srisa-an, and Hong Jiang. MicroPhase: An Approach to Proactively Invoking Garbage Collection for Improved Performance. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 77–96, 2007.

[YBKM06] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual Memory Support for Garbage-Collected Applications. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 103–116, 2006.

[YHB$^+$04] Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Automatic Heap Sizing: Taking Real Memory into Account. In *ACM International Symposium on Memory Management*, pages 61–72, 2004.

[ZBM01] John N. Zigman, Stephen Blackburn, and J. Eliot B. Moss. TMOS: A Transactional Garbage Collector. In *International Workshop on Persistent Object Systems*, pages 138–156, 2001.

[ZGG06] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning Dynamic Slices with Confidence. In *ACM Conference on Programming Language Design and Implementation*, pages 169–180, 2006.

[ZLF$^+$04] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants. In *IEEE/ACM International Symposium on Microarchitecture*, pages 269–280, 2004.

[Zor90] Benjamin Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, University of Colorado at Boulder, 1990.

[ZSCC06] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, Efficient, and Adaptive Calling Context Profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 263–271, 2006.

[ZTGG07] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards Locating Execution Omission Errors. In *ACM Conference on Programming Language Design and Implementation*, pages 415–424, 2007.

[ZZPL05] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Anomalous Path Detection with Hardware Support. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 43–54, 2005.

Vita

Michael David Bond graduated from Thomas Jefferson High School for Science and Technology in Alexandria, Virginia, in 1999. He received the degrees of Bachelor of Science and Master of Computer Science from the University of Illinois at Urbana-Champaign in 2002 and 2003. He entered the Ph.D. program at the University of Texas at Austin in 2003.

Permanent address: 804A E. 32nd 1/2 St.
Austin, Texas 78705

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.