Bell: Bit-Encoding Online Memory Leak Detection [*]

Michael D. Bond Kathryn S. McKinley

Dept. of Computer Sciences
University of Texas at Austin
{mikebond,mckinley}@cs.utexas.edu

Abstract

Memory leaks compromise availability and security by crippling performance and crashing programs. Leaks are difficult to diagnose because they have no immediate symptoms. Online leak detection tools benefit from storing and reporting per-object *sites* (e.g., allocation sites) for potentially leaking objects. In programs with many small objects, per-object sites add high space overhead, limiting their use in production environments.

This paper introduces *Bit-Encoding Leak Location* (Bell), a statistical approach that *encodes* per-object sites to 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, Bell uses brute-force *decoding* to recover the site with high accuracy.

We use this approach to encode object allocation and last-use sites in *Sleigh*, a new leak detection tool. Sleigh detects *stale* objects (objects unused for a long time) and uses Bell decoding to report their allocation and last-use sites. Our implementation steals four unused bits in the object header and thus incurs no per-object space overhead. Sleigh's instrumentation adds 29% execution time overhead, which adaptive profiling reduces to 11%. Sleigh's output is directly useful for finding and fixing leaks in SPEC JBB2000 and Eclipse, although sufficiently many objects must leak before Bell decoding can report sites with confidence. Bell is suitable for other leak detection approaches that store per-object sites, and for other problems amenable to statistical per-object metadata.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability, Statistical Methods

General Terms Reliability, Performance, Experimentation

Keywords Memory Leaks, Low-Overhead Monitoring, Probabilistic Approaches, Managed Languages

[*] This work is supported by NSF CCR-0311829, NSF ITR CCR-0085792, NSF CCF-0429859, NSF CISE infrastructure grant EIA-0303609, DARPA F33615-03-C-4106, DARPA NBCH30390004, Intel, and IBM. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

1. Introduction

Memory bugs are a notorious source of errors that compromise the availability and security of mission-critical systems. Memory bugs dominate US-CERT and CERT/CC vulnerability reports [9, 32], and the business cost of downtime due to software crashes is substantial [29]. Memory-related bugs include dangling pointers, double frees, buffer overflows, and leaks. *Memory leaks* occur because of

1. *Lost objects:* a program neglects to free a heap-allocated object that subsequently becomes unreachable, and
2. *Useless objects:* a program keeps a reference to an object but never uses the object again.

Leaks degrade performance, and growing leaks crash programs. Leaks may occur only in production environments and take hours, days, or weeks to manifest. Malicious users can exploit memory leaks to launch denial-of-service attacks. Memory leaks are harder to detect than other memory errors because they have no immediate symptoms [18].

Managed languages such as Java and C# are increasingly popular [16] in part because garbage collection and type safety solve many memory errors including lost objects, but they do not solve leaks due to useless objects. Leaks occur in practice in Java and C#, and many tools exist for detecting leaks in these languages [3, 22, 24, 27, 28].

Existing approaches to finding leaks in managed and unmanaged programs have serious limitations that include high space and time overhead, limiting their usefulness in production environments, or they trade accuracy and utility for lower overhead [3, 10, 18, 22, 24, 25, 26, 27, 28]. Many leak detection approaches track per-object source information such as allocation site [3, 10, 18, 25, 28]. These approaches impose space overhead of as much as 75% [10], which is undesirable when the end goal is to conserve memory.

In this paper, we introduce *Bit-Encoding Leak Location* (Bell), a novel approach for correlating object instances and *sites* (source locations such as allocation sites) with extremely low space overhead. Bell encodes the site for an object in a single bit using an *encoding* function $f(site, object)$ 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 require per-object information, such as leak detection, both in managed and unmanaged languages.

We use Bell to implement a new leak detector for Java called *Sleigh*. Sleigh, like SWAT from previous work [10], adds instrumentation at allocations and reads to identify *stale* memory (memory the program has not used in a while), and reports the allocation

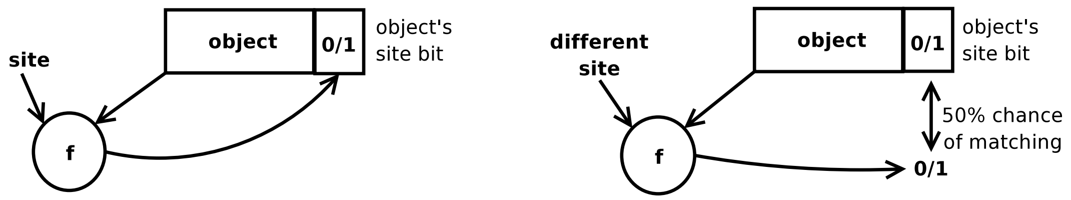


Figure 1. (a) An object's encoded site is stored in its site bit. (b) A different site matches the object with $\frac{1}{2}$ probability.

and last-use site(s) of stale objects. Sleigh (1) inserts instrumentation at each allocation and use site that performs Bell encoding; (2) clocks object staleness using a two-bit saturating *logarithmic counter* that it zeroes at use sites and increments from k to $k+1$ every b^k garbage collections for a user-defined base b; and (3) periodically decodes stale objects' sites. Sleigh uses four bits per object: one for allocation site, one for last-use site, and two for staleness. Our implementation steals unused bits in the object header and thus adds no per-object space overhead. Sleigh's instrumentation increases execution time by 29% on average, which adaptive profiling [10] reduces to 11%. Sleigh uses a mark-sweep garbage collector because Bell does not support moving objects, although we describe how to implement Sleigh with a generational mark-sweep collector.

Sleigh finds and helps fix memory leaks in SPEC JBB2000 and Eclipse [15, 31], which have known memory leaks. The fix for SPEC JBB2000 was previously known while the Eclipse leak was unfixed. Sleigh outputs the allocation and last-use sites responsible for stale objects, and for the subset of objects on the boundary between in-use and stale objects. This information is directly useful for fixing the leaks, although the programs need to run long enough to leak enough objects to be reported by Bell decoding.

The primary contribution of this paper is the novel Bell mechanism that efficiently encodes per-object information into a single bit, and decodes it with high confidence. The secondary contribution is Sleigh, a new memory leak detector that uses Bell to encode sites and a logarithmic counter to represent staleness, reducing space overhead to just four bits per object and incurring no per-object space overhead and average time overhead of 11% (29% without adaptive profiling).

2. Bit-Encoding Leak Location

This section presents Bit-Encoding Leak Location (Bell), a novel approach for encoding per-object information into a single bit.

2.1 Encoding

Bell *encodes* per-object information from a known, finite set in a single bit. In this paper, 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* takes two parameters, the site and object address, and returns zero or one:

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

Bell computes $f(\textit{site}, \textit{object})$ 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 $f(\textit{site}, \textit{object})$ 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 1 shows an example of the first property of an unbiased function. Section 2.3 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.

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

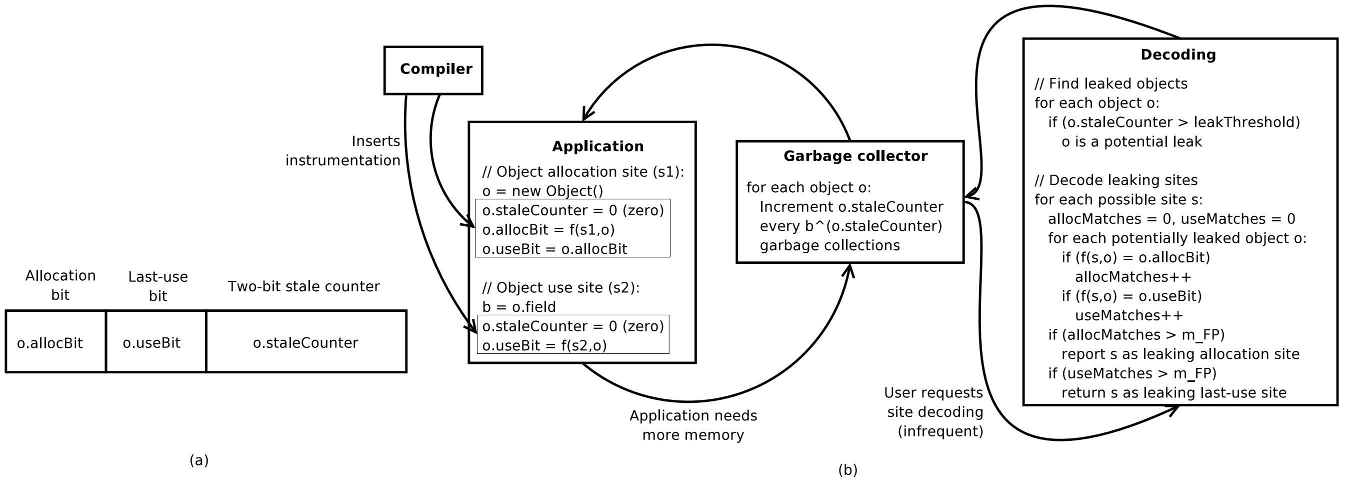
Bell decodes per-object sites using a brute-force approach that evaluates f for every object and every site:

```
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
```

Because of statistical variability, $2m_{\textit{site}} - n$ 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
```

The appendix describes how we compute m_{FP} so that decoding avoids false positives with high probability (99%). By weeding out



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

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%). The appendix describes how we compute n_{min}. The following table reports n_{min} for various numbers of sites and 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

The table 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.

2.3 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:

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

$f_{singleMult}$ 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 Sleigh implementation uses a segregated free list allocator (Section 3.6), 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:

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

We also experimented with

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

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.

3. Sleigh

This section describes *Sleigh*, a new memory leak detector that tracks staleness (time since last use) to find leaks, and uses Bell to identify sites associated with stale objects. We implement Sleigh on top of Jikes RVM 2.4.2, a high-performance Java-in-Java virtual machine. We have made Sleigh publicly available on the Jikes RVM Research Archive [20].

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 [17] 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.

Figure 2(a) shows the four bits that Sleigh uses in each object's header. Figure 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.

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 $f(site, object)$ 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 $f(site, object)$ and stores the result in the last-use bit. Figure 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.

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 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.

3.4 Decoding

Sleigh occasionally performs Bell decoding to identify the site(s) that allocated and last used (highly) stale objects. The user can configure Sleigh to trigger decoding periodically (e.g., every hour or every thousand GCs), or the user could trigger it on demand via a remote signal (not currently implemented). Decoding occurs during the next GC after being triggered. Figure 2(b) shows how GC occasionally invokes decoding, and it shows pseudocode for decoding based on the decoding algorithm from Section 2.2. 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 2.2), 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 separately from the VM executing the application, on a different CPU or machine (currently unimplemented). The VM would need to send the highly stale object addresses and the possible sites (or a delta since the last decoding), and the separate execution context 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 $f(site, object)$ (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.

3.5 Decreasing Instrumentation Costs

The instrumentation Sleigh adds at object uses (field and array element reads) can be costly because it executes frequently. Sleigh removes redundant instrumentation and uses adaptive profiling [10] 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 [8], 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 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 only happen to an object pointed at by a local (stack) variable continuously from 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 5.4). This time overhead is low compared to other memory leak detection tools (Section 6), but may be too expensive for online production use. To reduce this overhead, we borrow *adaptive profiling* from Chilimbi and Hauswirth [10], 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 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%.

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 [5], which allocates using a segregated free list and does not move heap objects.

Mark-sweep is not considered to be 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 [2]) 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 works with C and C++ memory managers, since they do not move objects.

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. Finding and Fixing Leaks

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

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 4.2 and 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 5). In Section 4.4, Sleigh samples object uses using adaptive profiling (*Sleigh AP* in Section 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.

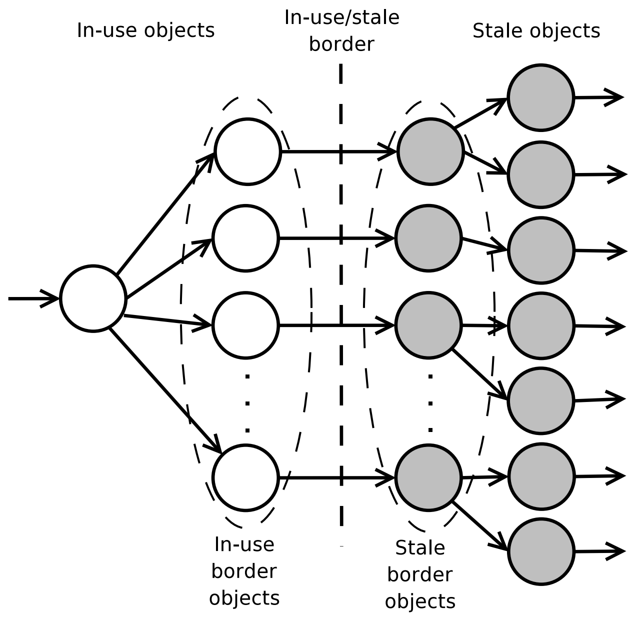


Figure 3. Sleigh implicitly divides the heap into in-use and stale objects.

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 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. 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 [10].

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.

Platform We perform our experiments 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,

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

Table 1. Decoding statistics for Sleigh running SPEC JBB2000. *Decoding processes at most 100,000 objects.

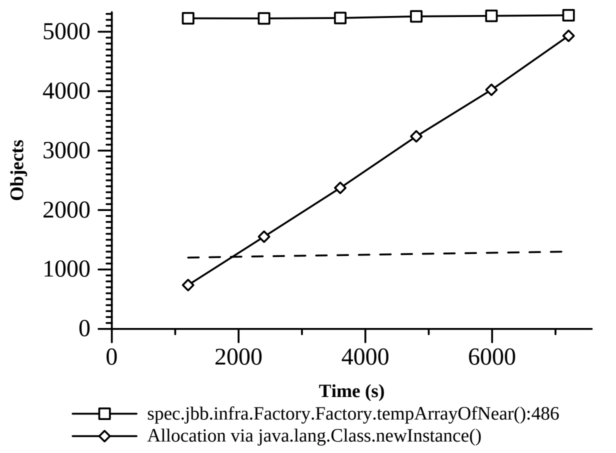


Figure 4. Reported *allocation sites* for SPEC JBB2000 when decoding processes *stale border objects* only.

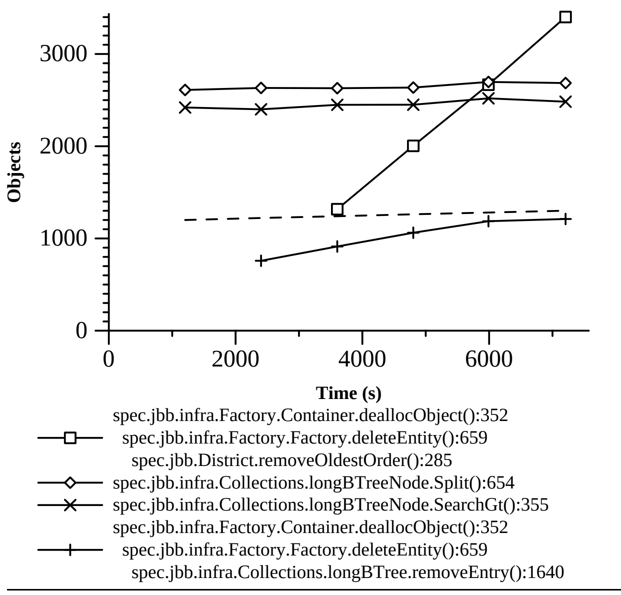


Figure 5. Reported *last-use sites* for SPEC JBB2000 when decoding processes *stale border objects* only.

a 2MB unified 8-way set associative L2 on-chip cache, and 2GB main memory, running Linux 2.6.12.

Benchmarks We evaluate Sleigh on two leaks in SPEC JBB2000 and Eclipse 3.1.2 [15, 31].

4.2 SPEC JBB2000

SPEC JBB2000 simulates an order processing system and is intended for evaluating server-side Java performance [31]. SPEC JBB2000 contains a known, growing memory leak that manifests when it runs for a long time without changing warehouses. The leak occurs because SPEC JBB2000 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 1 presents statistics from running Sleigh on SPEC JBB2000 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 SPEC JBB2000'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 and 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 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 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 SPEC JBB2000's leak: the fix replaces SPEC JBB2000's only call to `removeOldestOrder()` with two different lines that properly remove orders from SPEC JBB2000'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 SPEC JBB2000 developer could quickly fix the leak based on Figure 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 2.2). 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 SPEC

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

Table 2. Decoding statistics for Sleigh running Eclipse. *Decoding processes at most 100,000 objects.

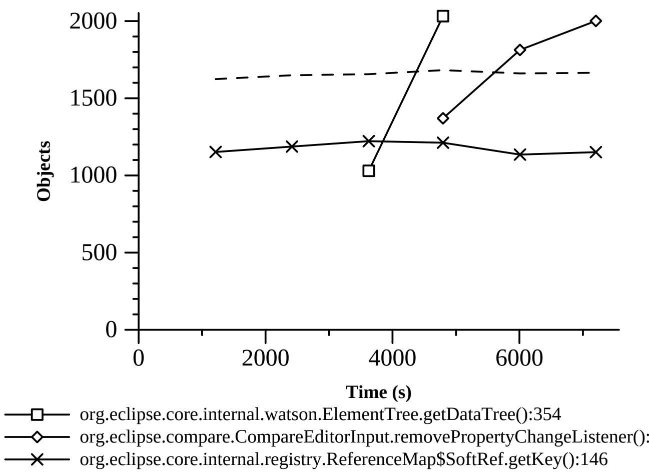


Figure 6. Reported *last-use sites* for Eclipse when decoding processes *stale border objects* only.

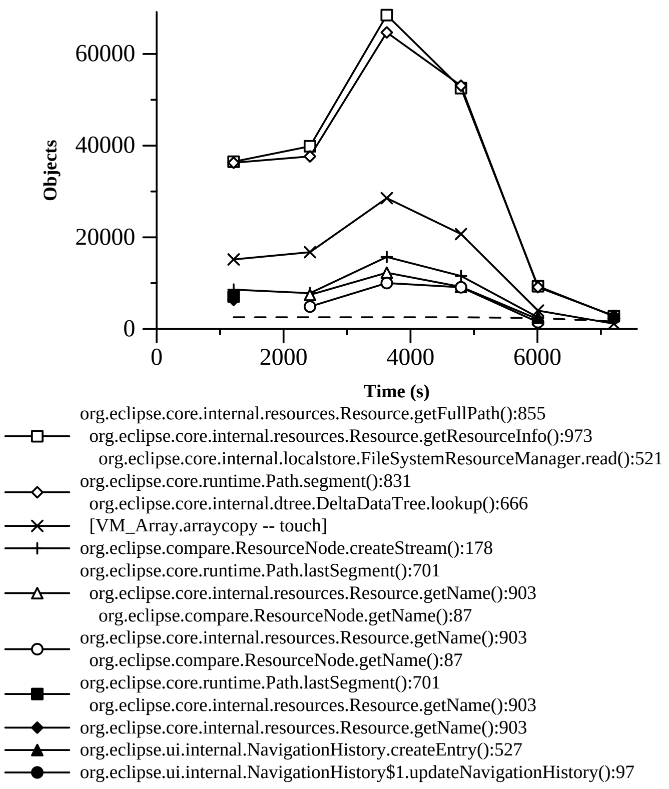


Figure 7. Reported *last-use sites* for Eclipse when decoding processes *in-use border objects* only.

JBB2000 well enough to know if the plot for in-use objects is useful for fixing the leak.

SPEC JBB2000's heap growth is due to both stale and in-use objects: `Orders` grow in number but are used, whereas `Containers` 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 SPEC JBB2000 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.

4.3 Eclipse

Eclipse 3.1.2 is a popular integrated development environment (IDE) written in Java [15]. 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 [5] 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 2 shows information about running Eclipse using Sleigh, in the same format as Table 1. 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 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 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 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 (Section 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 1 and 2 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 8 shows last-use sites for stale border objects from SPEC JBB2000. This plot is noisier than Figure 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 SPEC JBB2000 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.

4.5 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 [10]. 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 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 2.2). 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 (SPEC JBB2000 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 2.2) 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.

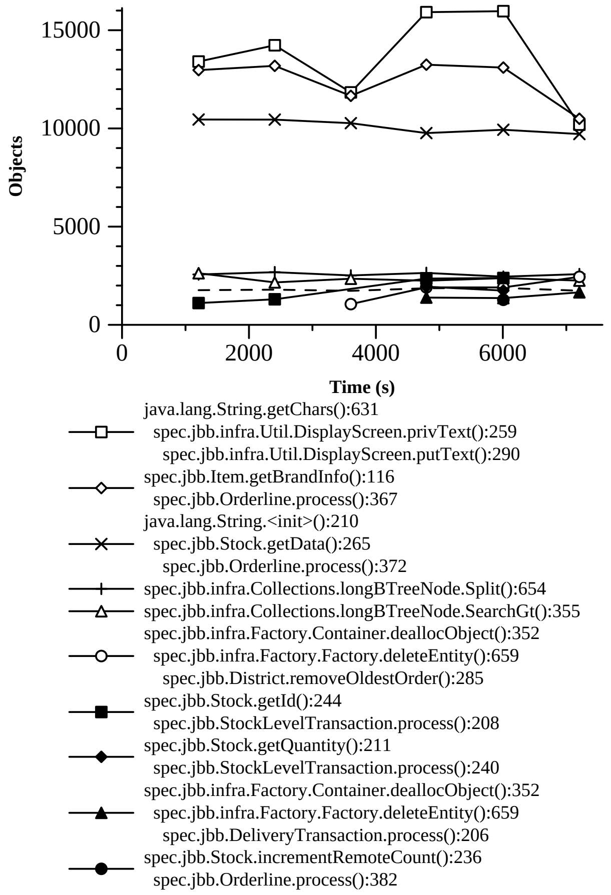


Figure 8. Reported *last-use sites* for SPEC JBB2000 when decoding processes *stale border objects* only, using *adaptive profiling*.

5. Sleigh's Runtime Performance

This section evaluates Sleigh's space and time overheads.

5.1 Methodology

Execution Jikes RVM runs by default using *adaptive* methodology, which dynamically identifies frequently executed methods and recompiles them at higher optimization levels. Because it uses timer-based sampling to detect hot methods, the adaptive compiler is non-deterministic. To measure performance, we use *replay compilation* methodology, which is deterministic. Replay compilation

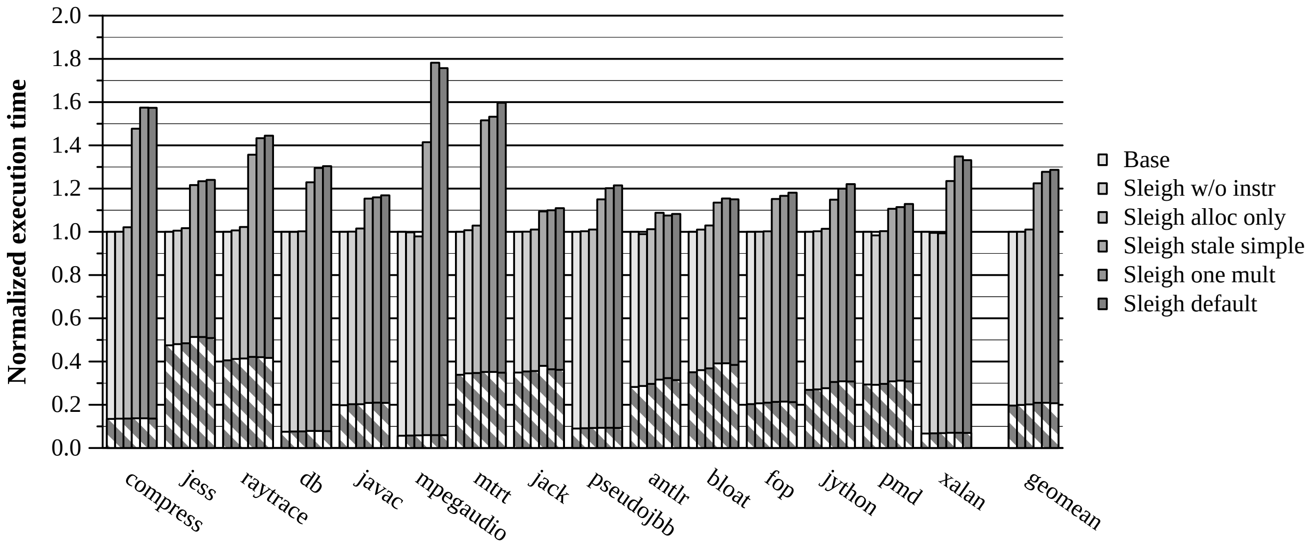


Figure 9. Components of Sleigh runtime overhead.

forces Jikes RVM to compile the same methods in the same order at the same point in execution on different executions and thus avoids high variability due to the compiler.

Replay compilation uses *advice files* produced by a previous well-performing adaptive run (best of 10). The advice files specify (1) the optimization level for compiling each method, (2) the dynamic call graph profile, and (3) the edge profile. Fixing these inputs, we execute two consecutive iterations of the application. During the first iteration, Jikes RVM optimizes code using the advice files. The second iteration executes only the application with a realistic mix of optimized code.

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 evaluates decoding's performance).

Platform We use the platform described in Section 4.1.

Benchmarks We evaluate Sleigh's performance using the SPEC JVM98 benchmarks, the DaCapo benchmarks (beta050224) that execute on Jikes RVM, and a fixed-workload version of SPEC JBB2000 called `pseudojbb` [6, 30, 31]. We omit the DaCapo benchmarks `hsqldb` and `ps` because we could not get them to run correctly with Jikes RVM, with or without Sleigh; both have known issues addressed in version 1.0 of the DaCapo benchmarks [6].

5.2 Space Overhead

Sleigh uses four bits per object to maintain staleness and encode allocation and last-use sites (Section 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).

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.

5.4 Time Overhead

Sleigh adds time overhead to maintain objects' stale counters and to encode objects' allocation and last-use site bits. Figure 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 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 2.3) 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%.

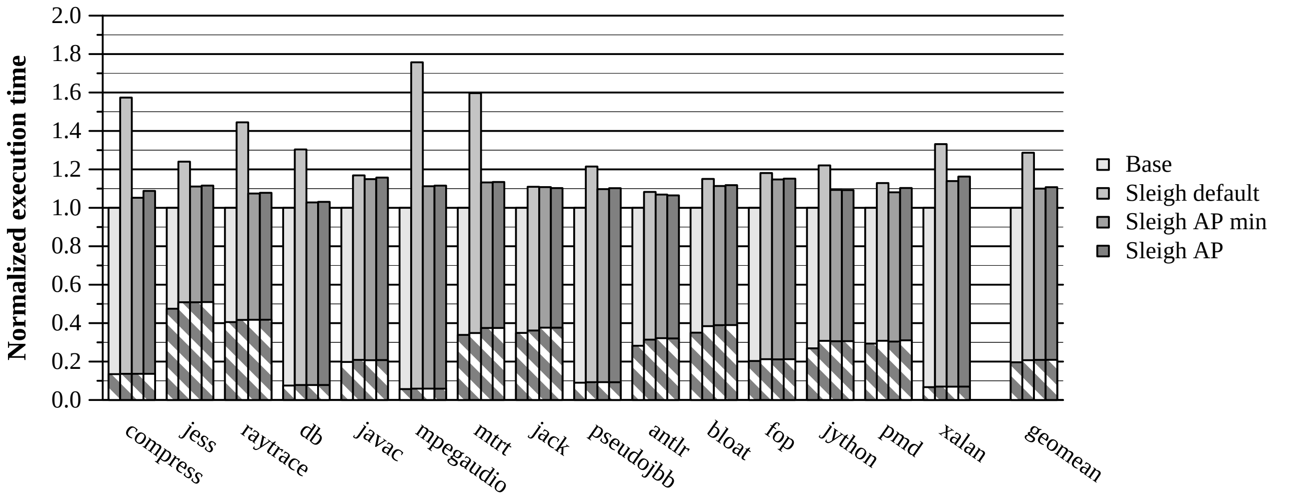


Figure 10. Sleigh runtime overhead with adaptive profiling.

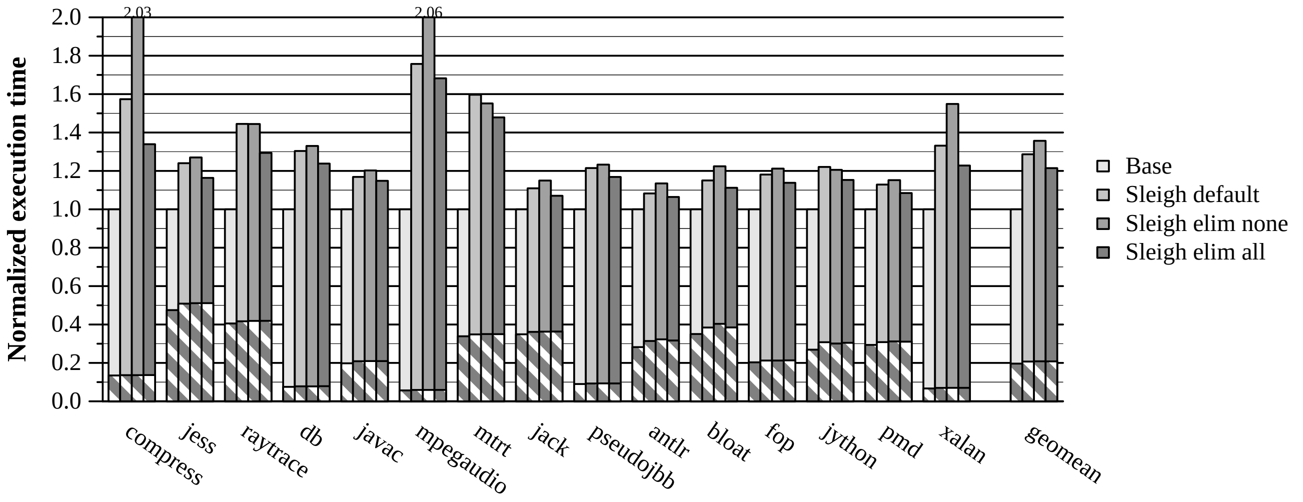


Figure 11. Sleigh runtime overhead with and without redundant instrumentation optimizations.

Adaptive Profiling Sleigh uses adaptive profiling to lower its instrumentation overhead at object uses (Section 3.5). Figure 10 shows the overhead of Sleigh with adaptive profiling. *Base* and *Sleigh default* are the same as in Figure 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 [10], 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.

Redundant Instrumentation All Sleigh configurations presented so far remove fully redundant but not partially redundant instrumentation (Section 3.5). Figure 11 shows the overhead of Sleigh with various redundant instrumentation optimizations. *Base* and *Sleigh default* are the same as in Figure 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.

6. Related Work

This section compares Bell and Sleigh to previous work in memory leak detection.

Static Analysis Static analysis finds memory leaks in programs without runtime overhead (e.g., [19]) but reports false positives since it must make conservative assumptions about control flow. Dynamic class loading in Java complicates static analysis since some classes may not be available at testing time. Current static analysis tools find lost objects but not useless objects. Finding useless objects statically seems inherently very challenging.

Dynamic Monitoring and Per-Object Information *Dynamic monitoring* tools find leaks at runtime, and many maintain and report per-object source information such as allocation site [3, 10, 18, 25, 28]. This information helps fix leaks but adds significant per-object overhead. These tools could benefit from Bell encoding, as long as sufficiently many objects leak. If just a few objects leak, Bell cannot decode per-object source information accurately, but the most problematic leaks are usually large and/or growing.

An alternative to Bell's statistical approach is to store *unencoded* per-object information for a sample of objects (e.g., dynamic object sampling [21]). Sampling avoids Bell encoding and decoding but still adds some space overhead and requires instrumentation that checks whether an object is in the sampled set.

Pre-Release Testing Tools Valgrind [25] and Purify [18] find memory leaks, as well as many other memory errors. 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. Sleigh finds only leaks while these tools find many memory errors, but Sleigh has low enough space and time overhead to consider using in production runs.

SWAT SWAT finds leaks in C and C++ programs by guessing that stale objects are leaks [10]. Sleigh borrows SWAT's staleness approach to find leaks. SWAT and Sleigh may report false positives (stale memory that will be used eventually), although these reports probably indicate poor memory usage. Both tools track per-object staleness and maintain per-object allocation and last-use sites, but SWAT adds several words of space overhead per object, while Sleigh saves space but cannot report sites that do not leak many objects because of its statistical nature. For C programs that allocate and custom-manage large chunks of memory [4], SWAT has low space overhead. On the C benchmark `twolf`, which allocates many small objects, SWAT adds 75% space overhead. Many programs heap-allocate many small objects (24-32 bytes per object on average) [14], where Bell's space-efficient mechanism offers substantial space advantages.

Leak Detection for Managed Languages JRockit [3], .NET Memory Profiler [28], JProbe [27], LeakBot [24], and Cork [22] 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 runtime.

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). In contrast, Sleigh uses staleness (time since last use) to find memory leaks and thus finds all memory the application is not using. Sleigh may report false positives if non-leaking memory is not used for a while, although these reports probably indicates poor memory usage.

SafeMem 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 [26]. 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. SafeMem requires some hardware and operating system support, whereas Sleigh's software approach offers comparable overheads and is implemented in the compiler and virtual machine.

Instrumentation Optimization Sleigh uses data-flow analysis to find partially and fully redundant instrumentation at object uses, and it removes fully redundant instrumentation (Section 3.5). The instrumentation at object uses (reads) is called a *read barrier* [7]. 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) [1, 7, 33]. Bacon et al. use common subexpression elimination to remove fully redundant read barriers, which reduces average overhead from 6 to 4% on the PowerPC [1]. Since our barrier 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 [13, 23]. For example, a well-known idea in communication complexity is that two bitstrings 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 [23]. Extracting random bits from two weakly random input sources (Bell's encoding function) is a well-studied area in communication complexity [11]. We are not aware of any work that probabilistically encodes and decodes program behavior as Bell does.

7. Conclusions

Bit-Encoding Leak Location (Bell) is a novel approach for encoding per-object information from a known, finite set in a single bit and decoding the information accurately given enough objects. We use Bell in Sleigh to find the program sites that allocated and last used leaked memory. We show Sleigh's output is directly useful for fixing a leak in SPEC JBB2000 and a previously unfixed leak in Eclipse, although enough objects must leak before Sleigh reports key sites. Sleigh incurs no per-object space overhead in our implementation and has low time overhead, making it suitable for production runs.

Bell solves a general problem and can be applied to other applications amenable to statistical per-object information. Bell could encode per-object allocation sites in a growth-based leak detector for just 1% overhead (Figure 9). It could be applied to other forms of profiling that use per-object information, such as profiling lifetimes of allocation sites for pretenuring [21]. While Bell needs many object instances to identify a site accurately, it can determine that a single object has *not* been encoded together with a particular site: an object and site that do not match were definitely not encoded together, while a match is inconclusive. Bell offers a compromise between accuracy and overhead that may be appealing for some applications.

A. Avoiding False Positives and Negatives

Section 2.2 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):

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

Before decoding, Sleigh solves for m_{FP} and m_{min} using the *Commons-Math* library [12].

Acknowledgments

We thank Maria Jump, Xianglong Huang, Steve Blackburn, Robin Garner, Alan Adamson, Elena Ilyina, and Ricardo Morin for help with Jikes RVM and benchmarks. We thank Xianglong Huang, Nicholas Nethercote, Daniel Jimẽnez, Samuel Guyer, and Maksim Orlovich for helpful discussions. We thank Andrew Mills and Jesse Kamp for help with related work in information theory and communication complexity. We thank Emery Berger, Katherine Coons, Chen Ding, Boris Grot, Jungwoo Ha, Byeongcheol Lee, Naveen Neelakantam, Nicholas Nethercote, Ben Wiedermann, and the anonymous reviewers for their helpful comments about the paper.

References

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

[2] BEA. JRockit. http://dev2dev.bea.com/jrockit/.

[3] BEA. JRockit Mission Control. http://dev2dev.bea.com/jrockit/-tools.html.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering Custom Memory Allocation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, 2002.

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

[6] 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 *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[7] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *International Symposium on Memory Management*, pages 143–151, 2004.

[8] P. Briggs and K. D. Cooper. Effective Partial Redundancy Elimination. In *Conference on Programming Language Design and Implementation*, pages 159–170, 1994.

[9] CERT/CC. CERT/CC Advisories. http://www.cert.org/advisories/.

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

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

[12] Commons-Math: The Jakarta Mathematics Library. http://jakarta.-apache.org/commons/math/.

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

[14] 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.

[15] Eclipse.org Home. http://www.eclipse.org/.

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

[17] N. Grcevski, A. Kielstra, K. Stoodley, M. G. Stoodley, and V. 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.

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

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

[20] Jikes RVM Research Archive. http://jikesrvm.sourceforge.net/info/-research-archive.shtml.

[21] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic Object Sampling for Pretenuring. In *International Symposium on Memory Management*, pages 152–162, 2004.

[22] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Java. Technical Report TR-06-07, The University of Texas at Austin, 2006. Under submission.

[23] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1996.

[24] N. Mitchell and G. 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.

[25] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[26] F. Qin, S. Lu, and Y. 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.

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

[28] SciTech Software. .NET Memory Profiler. http://www.scitech.se/-memprofiler/.

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

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

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

[32] US-CERT. US-CERT Vulnerability Notes Database. http://www.kb.-cert.org/vuls/.

[33] B. Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, University of Colorado at Boulder, 1990.