

# Tolerating Memory Leaks

UT Austin Technical Report TR-07-64

December 7, 2007

Michael D. Bond    Kathryn S. McKinley

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

## Abstract

Type safety and garbage collection in managed languages eliminate memory errors such as dangling pointers, double frees, and leaks of unreachable objects. Unfortunately, programmers may still leak memory by neglecting to eliminate references to objects the program will never use again. Leaked objects decrease program locality, increase garbage collection's frequency and workload, and if they grow over time, eventually exhaust memory and crash the program.

This paper presents a novel approach called *leak tolerance* that, given enough disk space to hold leaking memory, safely eliminates performance degradations and crashes due to leaks in managed languages. Leak tolerance provides the illusion of a fixed bug, buying developers more time to actually fix it. Leak tolerance (1) identifies *stale* objects, i.e., objects not used in a while; (2) segregates in-use and stale objects; (3) activates stale objects if the program subsequently accesses them; and (4) puts stale objects on disk using OS paging or explicit I/O. Activation makes leak tolerance completely safe. We design and implement a prototype leak tolerance tool called *Melt* in a Java VM and show it adds overhead low enough for deployed use. Our results show *Melt* sometimes provides ancillary performance benefits on programs without leaks, due mainly to reduced collection times. Most importantly, we show a number of leaky programs for which existing VMs grind to a halt and then crash, whereas *Melt* keeps the programs running without degrading performance.

## 1. Introduction

Managed languages use type safety and garbage collection to improve program reliability by eliminating errors inherent in explicit memory management. For example, these features eliminate premature and repeated frees of the same object. They also improve productivity by eliminating the effort required to insert frees correctly. Unfortunately, programmers may neglect to eliminate pointers to objects the program will never use again. These objects are leaked because garbage collection cannot collect them since it uses *reachability* as an approximation of *liveness*. Leaks increase garbage collection frequency and workload, and may hurt application performance. Growing leaks slow and eventually crash the application when memory is exhausted.

Leaks are especially hard to reproduce, find, and fix since they have no immediate symptoms [19]. Leaks in managed languages are a real problem, as evidenced by the commercial and research tools that help programmers diagnose them [4, 9, 25, 28, 36, 39]. Some previous work attempts to tolerate leaks or has the potential to tolerate leaks but is either unsafe or cannot prevent slowdowns and memory exhaustion [16, 21, 30]. Goldstein et al. move large, leaking objects to disk, but their approach cannot handle small objects, and it adds space overhead proportional to leaked ob-

jects [16]. *Bookmarking collection* tolerates limited physical memory at page granularity by not collecting objects on swapped-out pages [21]. *Cyclic memory allocation* tolerates leaks but is unsafe since it overwrites potentially live data [30]. None of the prior work safely tolerates arbitrary leaks online as they occur in the wild.

This paper presents a new approach called *leak tolerance* that prevents programs with memory leaks from slowing and crashing, given enough disk space. Our approach (1) identifies *stale* objects (objects not used in a while); (2) segregates *in-use* (non-stale) and stale objects and does not access the stale space except to move objects in or out; (3) maintains safety by activating stale objects accessed by the program; and (4) uses paging or explicit I/O to transfer stale pages to disk.

We design and implement a prototype leak tolerance tool for Java called *Melt*. Experimental results show *Melt* has low enough overhead on DaCapo, SPEC JBB2000, and SPEC JVM98 to consider using it in deployed software. Identifying stale objects adds 10% overhead on average, and moving them to the stale space adds negligible overhead and often provides modest ancillary benefits since it reduces collector frequency and workload.

For two leaks in Eclipse, a leak in a Delaunay triangulation application, and two third-party microbenchmarks with leaks, we show that a VM enhanced with leak tolerance can keep leaking programs running whereas our base VM and a production VM we tested crash with out-of-memory errors. These results indicate that leak tolerance is a viable approach for safely increasing program durability with relatively low overhead.

## 2. Related Work

Although there is a lot of prior work on detecting leaks, only a few researchers have tried to tolerate leaks. Leak tolerance improves over previous approaches because it offers a comprehensive and safe solution that identifies stale objects (objects not used in a while), handles small leaking objects, and keeps time and space proportional to in-use memory.

### 2.1 Detecting Leaks

Static analysis for C and C++ detects some leaks before the program executes but can produce false positives [20]. Dynamic tools for C and C++ track allocations, heap updates, and frees to report unfreed objects [19, 26, 29, 34] or track object accesses to report stale objects [12]. Online leak detectors for managed languages identify heap growth or stale objects to find potential leaks [4, 9, 25, 28, 36, 39]. Leak tolerance also identifies stale objects but does not report diagnostic information about them, although it could be combined with a leak detection approach.

SWAT and Sleight use *read barriers* (instrumentation at every program read) to identify stale memory [9, 12]. Blackburn and Hosking and Zorn report costs of a variety of barriers [8, 46]. Read

barriers are required for incremental and concurrent collectors that move objects [3, 13, 32].

## 2.2 Language Design, Bug Tolerance, and Dynamic Memory Sizing

To help programmers avoid leaks, the Java language definition provides *weak references*; GC collects weakly-referenced objects [15]. Inserting weak references adds to the development burden and programmers may still forget to eliminate the last non-weak reference.

In the general area of bug tolerance, failure-oblivious computing [37], DieHard [5], and Rx [35] deal with memory corruption and nondeterministic errors to improve reliability. Other work adaptively triggers GC or resizes the heap, in order to improve GC performance and program locality [11, 44, 45]. These approaches do not directly address memory leaks.

## 2.3 Tolerating Leaks

Three recent publications address the problem of tolerating leaks or could potentially tolerate leaks, but compared to our approach they offer less coverage or are unsafe [16, 21, 30].

*Bookmarking collection* does not specifically target leaks but instead seeks to reduce collection overhead in page-constrained environments [21]. It *bookmarks* swapped-out pages by marking 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, while our approach uses object granularity, allowing it to isolate leaking objects on in-use pages. Bookmarking modifies the OS to trigger bookmarking of swapped-out pages, while leak tolerance can use standard OS paging or explicit I/O to write stale objects to disk.

Goldstein et al. propose moving large, leaked objects in Java to disk [16]. The approach does not handle small leaked objects, which we find are frequently leak culprits; it does not have an automatic mechanism for detecting potentially leaked objects; and it adds per-stale object overhead, which would not scale to small objects.

Cyclic memory allocation tolerates leaks in C and C++ by limiting each program allocation site to  $m$  live objects at a time [30]. It uses profiling runs to select an  $m$  for each allocation site, and it allocates into  $m$ -sized circular buffers. Cyclic memory allocation is unsafe since it may overwrite live memory, although failure-oblivious computing [37] mitigates the effects in some cases.

In summary, leak tolerance offers the first safe and comprehensive approach for tolerating leaks in managed languages.

## 3. Leak Tolerance

This section presents our leak tolerance approach, including some implementation details likely to apply to any implementation. The next section describes details specific to our implementation.

### 3.1 Objective, Goals, and Invariants

Leak tolerance'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, leak tolerance meets the following design goals:

1. To make leak tolerance scale for growing leaks, its time and space overheads must not be proportional to the leaked memory.
2. To make leak tolerance safe, it must preserve and, if needed, activate stale objects.

Furthermore, it adheres to the following invariants:

- *Stale* objects (objects not used in a while) are isolated from the in-use objects in a separate *stale space*.

- The collector never accesses objects in the stale space.
- The application never accesses objects in the stale space. Leak tolerance maintains this invariant by intercepting program attempts to access the stale space and immediately moving the object into the in-use space.

To satisfy these invariants, leak tolerance uses the following components, which we describe in the rest of this section: (1) identify stale objects (Section 3.2); (2) segregate stale objects from in-use objects and use indirection for references from stale to in-use objects (Section 3.3); (3) activate stale objects on program accesses (Section 3.4); (4) store stale objects to disk (Section 3.5).

### 3.2 Identifying Stale Objects

Leak tolerance identifies reachable but stale objects and considers them potential leaks. To track which objects are being used, leak tolerance modifies both the garbage collector and the dynamic compiler. The high-level idea is that the collector *marks* objects stale each collection, and the compiler adds instrumentation to the application to *unmark* objects at each use. At the next collection, objects *not* accessed since the last collection will be marked stale, while accessed objects will be unmarked.

For efficiency, the collector actually marks both references and objects 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 or all VMs. The collector marks *objects* stale by setting a bit in the object header.

The compiler adds instrumentation called a *conditional read barrier* [8] to every load of an object reference. The barrier checks if the reference is marked stale; if so, the barrier unmarks the reference and stores it back to memory. It also unmarks the referenced object. The following pseudocode shows the barrier:

```
b = a.f;           // Application code
if (b & 0x1) {     // Conditional read barrier
  b &= ~0x1;       // Unmark the reference
  a.f = b;         // Store unmarked reference
  b.staleHeaderBit = 0x0; // Unmark the object
}
```

This conditional barrier reduces overhead since it performs stores only the first time the application loads each reference. Checking for a marked *reference*, rather than a marked *object*, reduces overhead since it avoids an extra memory load on every read.

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 1 shows an example heap at collection time. We know objects C and D are stale (shaded gray), i.e., 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 (not stale) because the reference  $A \rightarrow B$  is in-use.

### 3.3 The Stale Space

Leak tolerance moves stale objects to the *stale space* when the garbage collector traces the heap. Figure 2 shows an abstraction of the heap after objects C and D move to the stale space.

**Stub-Scion Pairs** References from the stale space to the in-use space (e.g., references from C and D to B in Figure 2) are problematic because the collector may later move in-use objects (e.g., B). To update these references, the collector would need to update the stale space, which would violate the invariants presented earlier. Potential solutions that keep track of references from stale to in-use using in-use memory (e.g., *remembered sets* [24]) are problematic since the number of these references can be proportional to the number of stale objects.

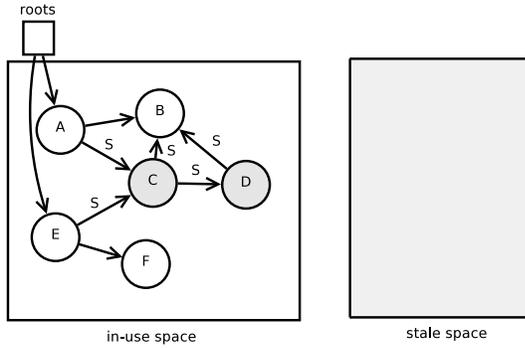


Figure 1. Stale Objects and References

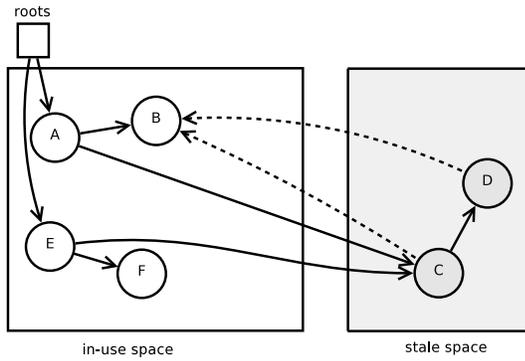


Figure 2. Segregation of In-Use and Stale Objects

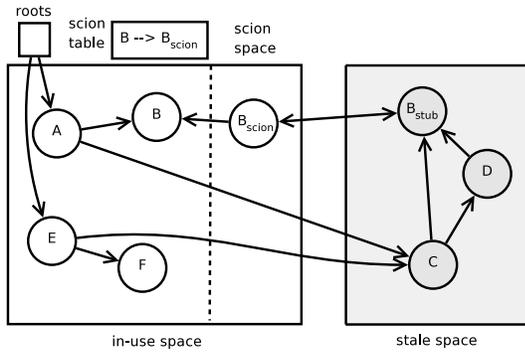


Figure 3. Stub-Scion Pairs

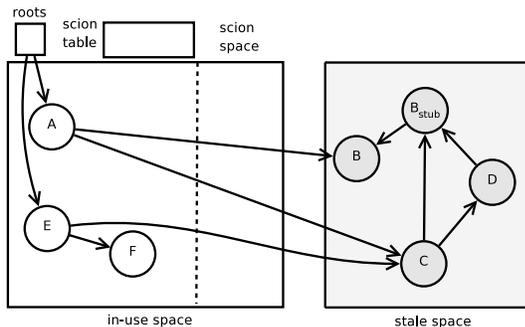


Figure 4. Scion-Referenced Object Becomes Stale

Leak tolerance solves this problem using *stub-scion pairs*, borrowed from distributed garbage collection [33]. Stub-scion pairs provide two levels of indirection. For each object in the in-use space referenced by one or more stale objects, leak tolerance creates a *stub* object in the stale space and a *scion* object in the in-use space. The collector references and modifies scions as needed to avoid touching stubs and stale objects. 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 instead refer to the stub. Figure 3 shows  $B_{\text{stub}}$  and  $B_{\text{scion}}$  providing two levels of indirection for references from C and D to B.

Scions may not move. The collector treats scions as roots in order to retain 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, leak tolerance builds a *scion lookup table* that maps from an in-use object to its scion, if any. This data structure is proportional to the number of scions, which is proportional to (but in practice much smaller than) the number of in-use objects. The collector processes the scions at the beginning of collection and rebuilds the table. Returning to Figure 2, when the collector copies C to the stale space, B initially has no entry in the scion lookup table, so it adds a mapping  $B \rightarrow B_{\text{scion}}$  to the table when it creates  $B_{\text{stub}}$  and  $B_{\text{scion}}$ . Next, when it copies D to the stale space, it finds the mapping  $B \rightarrow B_{\text{scion}}$  in the table and reuses the existing stub-scion pair. The resulting system snapshot is shown in Figure 3.

It may seem at first that leak tolerance needs 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 when B becomes stale in Figure 3. Without a stub, in order to eliminate the scion (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, leak tolerance copies B to the stale space, looks up the stub location in the scion, and points the stub to stale B. It then deletes the scion and removes the entry in the scion lookup table. Figure 4 shows the result.

This section discussed moving objects to the stale space and maintaining references to the in-use space. We defer further discussion of the stale space (e.g., transferring it to disk) until Section 3.5.

### 3.4 Activating Stale Objects

The application may occasionally attempt to access an object in the stale space. Leak tolerance prevents the application from accessing the stale space since (1) this would violate the invariant that the stale space is not part of the application’s working set, and also (2) object references in the stale space may redirect through stubs and scions and therefore the application cannot access them directly.

Leak tolerance intercepts application access to stale objects by modifying the read barrier to check for references to the stale space:

```

b = a.f;          // Application code
if (b & 0x1) { // Read barrier
    b &= ~0x1;
    if (inStaleSpace(b)) { // Check for stale space
        b = activateStaleObject(b);
    }
    a.f = b;
    b.stateHeaderBit = 0x0;
}

```

A VM method `activateStaleObject()` copies the stale object to the in-use space. Since other references may still point to the

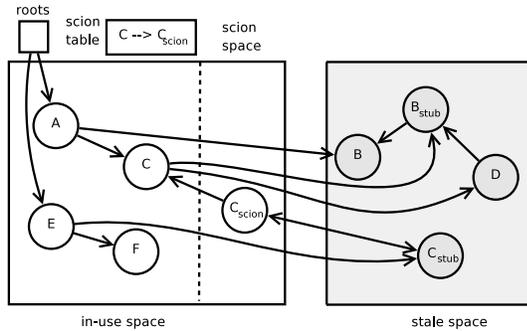


Figure 5. Stale Object Activation

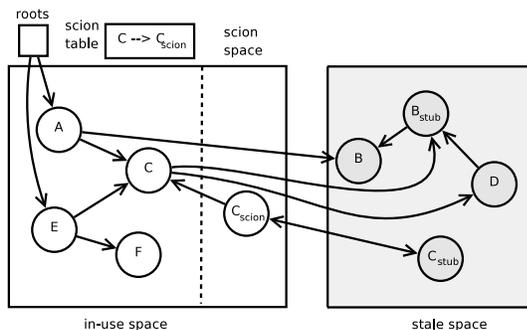


Figure 6. Reference Updates Following Activation

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.

Figure 5 shows an example. When the application tries to access `C` (which was stale in Figure 4), the read barrier activates it. 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. Note that `C` retains its references to `D` and `Bstub`, and `E` retains its reference to the old version of `C`, which is now `Cstub`.

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 (perhaps via a chain of stubs) to the scion, which points to the activated object. The first access of such a reference will update the reference to point to the activated version. Therefore, any subsequent accesses will go directly to the in-use object. For example, if the application accesses a reference from `E` to `Cstub` in Figure 5, `activateStaleObject()` follows `Cstub` to `Cscion` to `C` in the in-use space, and updates the reference, as shown in Figure 6.

### 3.5 Handling the Stale Space

Leak tolerance moves stale objects outside the working set of the application and collector. As Section 3.1 mentioned, we modify the collector so it never visits objects on stale pages. The collector does not account for these pages when computing the memory footprint of the application. This accounting gives the illusion that the memory is not in the heap.

Because leak tolerance moves stale objects to pages outside the working set of the application and collector, the operating system can page out the objects to disk. On a 64-bit platform with lots of swap space, this approach can tolerate leaks for a long time. However in some cases, it may be necessary to explicitly *swizzle* [43]

objects to disk. Swizzling supports using 64-bit addresses for stale objects on disk, even on a 32-bit platform. Stale objects referenced by in-use objects still need 32-bit addresses, but the number of these objects is proportional to in-use space.

An orthogonal and complementary approach is to reduce the size of the stale space by collecting unreachable objects. As presented, leak tolerance is *incomplete*: it does not free all unreachable objects since it does not collect the stale space. However, all unreachable, uncollectible objects will eventually move to the stale space since they are inherently stale. For example, in Figure 6, even if `C` becomes unreachable, it will still be kept alive by the scion referencing it; leak tolerance will eventually move `C` to the stale space. Using reference counting in the stale space would help collect unreachable objects, but (1) it cannot collect cycles (a fundamental limitation of reference counting), and (2) a single reference update could cause cascading frees, which could thrash badly on large leaks. Alternatively, leak tolerance could occasionally trace all memory including the stale space. This trace could have terrible performance for large leaks but might be worthwhile for a small stale space with many unreachable objects.

Another approach that could be combined with the above approaches is to compress the stale space [10]. The stale space is especially amenable to compression compared with a regular heap because the stale space is accessed infrequently.

Our prototype implementation does not compress, swizzle, or collect the stale space. It relies on the OS to page objects to disk. Since we run on a 32-bit platform, our implementation tolerates leaks until the VM runs out of virtual memory.

## 4. Implementation Details

This section presents details specific to our implementation of leak tolerance, which we call *Melt* (“MEmory Leak Tolerance”).

Leak tolerance is suitable for garbage-collected, type-safe languages using various tracing-based collectors. We implement *Melt* in Jikes RVM 2.9.2, a high-performance Java-in-Java virtual machine [1, 2, 23]. We will make *Melt* publicly available after leak tolerance is accepted for publication.

**Garbage Collection** Leak tolerance is compatible with moving and non-moving collectors that trace the heap (it does not support reference counting without significant modifications). Since leak tolerance is more beneficial and more challenging to implement for moving collectors, we implement *Melt* in a high-performance generational copying collector. This collector allocates objects into a *nursery*; when the nursery fills, the collector traces the live nursery objects and copies them into a copying *mature space*. The collector reserves half the mature space for copying. When the mature space fills, the collector performs a full-heap collection that copies all live mature objects into the copy reserve. An in-place compacting collector supports tighter heaps since it does not have a copy reserve, but Jikes RVM’s compacting collector is not robust in the latest version; we hope to run with a compacting collector for the final paper.

Jikes RVM’s memory manager, the Memory Management Toolkit (MMTk) [6], supports a variety of garbage collectors with most functionality residing in shared code. Almost all the changes for *Melt* are in this shared code. To enable *Melt* to work with a new collector, one implements only methods that specify (1) the space(s) that contain objects that can be moved to the stale space and (2) the space into which *Melt* should activate objects.

**Identifying Stale Objects** To identify stale objects, *Melt* modifies (1) the compiler to add read barriers to the application, and (2) the collector to mark heap references and objects stale (Section 3.2). Jikes RVM uses two compilers: an initial baseline compiler and an

optimizing compiler invoked for hot methods at successively higher optimization levels. We modify both compilers to add read barriers.

The collector marks references between heap objects, and it marks heap objects using an available bit in the object header. References from collection roots (registers, stacks, and statics) are considered inherently in-use and Melt *never* marks them stale. Similarly, an object is *never* marked stale if it is directly referenced by a root. Thus, the compiler inserts read barriers only for heap accesses, not root accesses, which are much more common and would make barrier overhead unacceptably high. Melt therefore never moves objects referenced directly by roots to the stale space.

Because Jikes RVM is written in Java, it adds its own objects to the heap. Although Melt could in theory detect stale objects that belong to the VM, (1) correctly inserting comprehensive read barriers in VM code would be quite challenging because the VM is not written entirely in pure Java, for example, it uses “magic” to access raw memory directly, and (2) we focus on tolerating leaks in applications rather than the VM. To differentiate VM and application objects easily, we allocate objects with VM types or that are allocated in a VM method into a special *VM object space*. Melt does not mark objects in this space or references to or from it, and it does not add read barriers to VM code or reads of objects with VM type.

**Making Objects Stale** Melt moves objects to the stale space (Section 3.3) during full-heap collections only. Since full-heap collection traces and moves every object in the mature space, Melt easily modifies this behavior to move stale objects to the stale space.

Melt supports stale *large* objects (8 KB or larger), which MMTk allocates into a special non-moving *large object space*. Since large objects are already allocated on their own pages, Melt does not move a stale large object but considers its pages part of the stale space by setting bits in a small side bit vector, where each bit represents whether a page in the large object space is stale. Since Melt considers the object stale, it does not let the collector trace the object, and it adds stub-scion pairs for references from the object. If the read barrier activates a large object, Melt activates it in-place by clearing the corresponding bits in the side bit vector.

Section 3.3 argues that Melt needs both stubs and scions for in-use objects referenced by the stale space. However, an in-use object that cannot move to the stale space needs only a scion. Thus, our implementation creates scions but not stubs for referenced objects in the VM object and large object spaces (since large objects become stale without moving).

**Staleness Threshold** Melt moves objects to the stale space and marks objects stale every  $N$  full-heap collections, where  $N$  is initially 1. However, the time between two collections may not be enough to properly identify stale objects: the application may later access many objects moved to the stale space, resulting in wasted stale space and in-use space (for scions pointing to activated objects). Melt adjusts  $N$  automatically. Currently our implementation increments  $N$  by 1 following a collection where objects are moved to the stale space, if the number of bytes activated since  $N$  collections ago is 10% or more of bytes moved to the stale space  $N$  collections ago. In addition, Melt moves objects to the stale space during any collection that MMTk identifies as an “emergency collection,” which means the VM is almost out of memory. These heuristics work fairly well for our experiments, but we leave it to future work to find a more principled approach that takes into account other factors such as allocation rate and how fast the leak is growing.

**Activating Stale Objects** Melt uses read barriers to intercept application reads to the stale space (Section 3.4). Melt immediately copies the object to the mature space and updates the reference. Since activation allocates into the in-use part of the heap, it may trigger a garbage collection. Application reads are not GC-safe

points in general, so Melt defers immediate collection by requesting an *asynchronous collection*, which causes collection to occur at the next GC-safe point.

## 5. Results

This section evaluates Melt’s performance and ability to tolerate leaks. First we present experimental methodology. We measure overhead for identifying and segregating stale objects on benchmarks without leaks, and then we evaluate how well Melt tolerates leaks in several real programs and third-party microbenchmarks.

### 5.1 Methodology

**VM Configurations** By default, Jikes RVM initially uses a baseline non-optimizing compiler to generate machine code. Over time, it dynamically identifies frequently-executed methods and recompiles them at higher optimization levels. We refer to experiments using this default execution model as using *adaptive* methodology. Because Jikes RVM uses timer-based sampling to detect hot methods, the adaptive methodology is nondeterministic. In particular, compilation allocates memory and perturbs garbage collection workload. To eliminate this source of nondeterminism, we use *replay* methodology [22, 31, 38]. Replay forces the VM 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 sampling-driven compilation.

Replay compilation uses *advice files* produced by a previous well-performing adaptive run (best of five). 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 unoptimized and optimized code. Replay eliminates run-to-run variation; we report the median of five trials to account for any external perturbation.

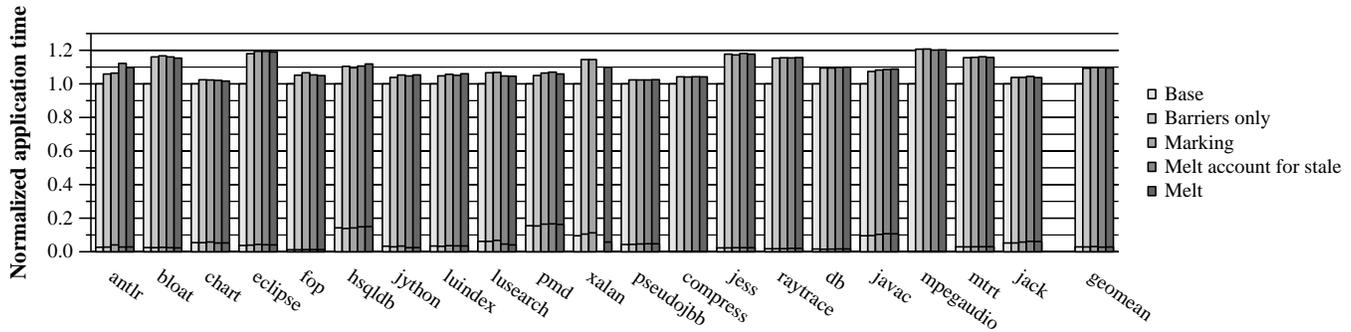
**Benchmarks** To measure Melt’s overhead, we use the DaCapo benchmarks (version 2006-10-MR1), a fixed-workload version of SPEC JBB2000 called *pseudojbb*, and SPEC JVM98 [7, 40, 41].

**Platform** All experiments 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 2.6.20.3.

### 5.2 Melt’s Overhead

**Application Overhead** Figure 7 presents the run-time overhead Melt adds to programs. We run each benchmark in a single medium heap size, two times the minimum in which it can execute, since we are primarily interested in measuring read barrier overhead. Each bar is normalized to *Base* (unmodified VM) and includes application and collection time, but not compilation time. The bottom sub-bars are the fraction of time spent in collection. *Barriers only* includes only Melt’s read barrier; the barrier’s condition is never true since the collector does not mark references stale. *Marking* adds marking of references and objects, which also exercises the read barrier’s body. *Melt account for stale* adds all the functionality of Melt (e.g., stale space, stub-scion pairs, activation), but the stale space counts as used memory. This configuration shows the benefit of reducing collection workload without reducing collection frequency. *Melt* is the default configuration, including not accounting for the stale space, that we use to tolerate leaks in the next section.

The graph shows that the read barrier alone costs 9% on average, and adding *Marking* adds an additional 1%. Dividing the heap into



**Figure 7. Application execution time overhead of read barriers, marking references and objects stale, and using the stale space.** Sub-bars are garbage collection time.

stale and in-use spaces and collecting only the in-use space (the *Melt* configurations) has a negligible effect on overall performance, although below we show its effect on collector performance is significant. The configuration *Melt account for stale* runs out of memory for *xalan* since the staleness threshold is too aggressive, and the collector moves many objects to the stale space that later become unreachable, but it cannot collect them.

We note that the performance of Jikes RVM has become increasingly competitive due to concerted efforts by developers throughout 2007. We initially implemented *Melt* in Jikes RVM 2.4.6 and measured 5% read barrier overhead. Later we ported to version 2.9.2 for robustness and ease-of-programming benefits provided by the new version, and read barrier overhead had increased to 10%! Our read barrier was the same, but program performance roughly doubled due to improvements to the VM. We have also found that Jikes RVM is comparable to Sun’s HotSpot VM on some real programs (Section 5.3).

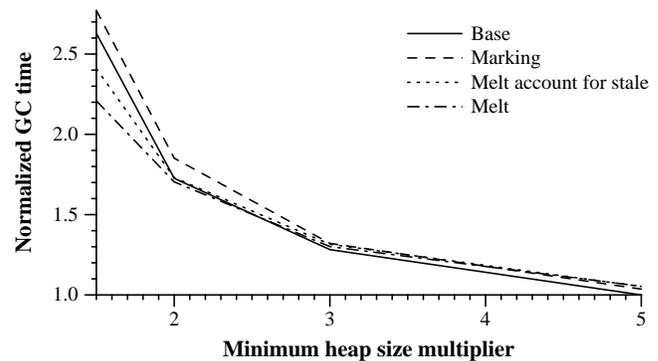
*Melt*’s 9% read barrier overhead is nontrivial but comparable to read barrier overheads used by concurrent, incremental, and real-time collectors [3, 13, 32], whose increasing prevalence may lead to general-purpose hardware support for read barriers. *Melt* achieves low overhead because the common case is just two IA32 instructions in optimized code: a register comparison and a branch. However, we find that half the barrier overhead comes from baseline-compiled code (results not shown), which by design accounts for a small fraction of execution but currently uses an expensive method call for the read barrier. For the final paper, we plan to inline the barrier’s common case into baseline-compiled code, as we currently do for optimized code.

An alternative to all-the-time read barriers is to activate *Melt* only when the program starts to run out of memory, although this would require recompiling all the code to add barriers.

Although *Melt* could potentially improve application-only (not collector) performance by compacting in-use memory, the graphs show negligible if any improvement on these non-leaking programs. We run with hardware performance counters and find that *Melt* improves over *Marking* as follows: reduces L1 misses by 4%, increases L2 misses by 3%, and reduces TLB misses by 2%. The overall increase in L2 misses is surprising and is due to a few programs experiencing huge increases (up to 80%).

**Collection Overhead** Figure 8 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.

The graph shows *Marking* slows collection by 5-7% for the smaller heap sizes. The other configurations measure both the over-



**Figure 8. Normalized GC times for *Melt* across heap sizes.**

head and benefits of using the stale space. *Melt account for stale*, which reduces the GC workload by putting objects in the stale space, recoups the overhead from marking since by avoiding tracing stale objects. *Melt*, which enjoys the benefits of reduced GC workload and frequency due to stale space discounting, speeds collection 25% over *Marking* and 19% over *Base* for the 1.5x heap.

**Compilation Overhead** Read barriers add compile-time overhead by increasing code size and slowing downstream optimizations. We find barriers increase generated code size by 4% and compilation time by 26% on average (results not shown). Because compilation accounts for just 4% on average of overall execution time, the effect of compilation on overall performance is modest.

To obtain the most realistic estimate of barrier costs, we also execute with adaptive methodology. To account for high variability, we execute 25 trials of each experiment and find that our read barrier adds 9% on average to overall run time (results not shown).

***Melt* Statistics** Table 1 presents statistics for *Melt* running the DaCapo benchmarks. We run with a small heap, 1.5 times the minimum heap size for each benchmark, in order to collect the most accurate statistics. The table presents the total number of objects moved to the stale space and activated by the program. It also shows objects in the in-use and stale spaces, pointers 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. We exclude *fop* and *hsqldb* since they execute fewer than two full-heap GCs.

The table shows that *Melt* moves 8–81 MB to the stale space, and the program activates only 0–7 MB of this memory. Some benchmarks activate a significant fraction of stale memory, for example, more than 10% for *eclipse* and *lusearch*. These high

	Total				Average per GC						
	Moved to stale		Activated		In-use		Stale		Stale→in-use	Scions	GCs
antlr	158,124	(8,118 KB)	6	(0 KB)	20,271	(4,161 KB)	158,124	(8,118 KB)	11,419	4,038	3
bloat	336,313	(15,094 KB)	50,080	(1,680 KB)	61,847	(5,340 KB)	289,781	(13,702 KB)	37,663	14,540	6
chart	193,621	(8,930 KB)	167	(7 KB)	119,421	(12,451 KB)	193,482	(8,924 KB)	29,288	5,113	6
eclipse	876,777	(41,200 KB)	188,962	(7,432 KB)	367,918	(22,166 KB)	600,704	(31,212 KB)	145,784	50,194	21
jython	205,314	(10,165 KB)	5,596	(303 KB)	65,532	(5,684 KB)	204,594	(10,135 KB)	59,873	34,083	11
luindex	158,701	(7,923 KB)	552	(13 KB)	17,632	(4,879 KB)	158,341	(7,914 KB)	28,104	1,707	4
lusearch	251,444	(14,970 KB)	30,715	(2,056 KB)	69,079	(43,261 KB)	248,695	(14,766 KB)	12,486	24,313	18
pmd	391,017	(17,594 KB)	20,280	(631 KB)	141,555	(8,668 KB)	320,121	(14,792 KB)	20,266	13,020	17
xalan	483,805	(81,136 KB)	18,005	(525 KB)	50,106	(16,189 KB)	374,124	(54,956 KB)	62,840	5,611	104

**Table 1.** Leak tolerance statistics for the DaCapo benchmarks.

activation rates are due to an initially aggressive staleness policy: Melt moves objects to the stale space if they are stale between two full-heap GCs, and it slowly increases this period if the activation rate is high (Section 4). Since our benchmarks are not long-running programs, we start with the lowest period in order to exercise Melt and measure potential improvements and degradations. Long-running programs with growing leaks would likely benefit from a less aggressive policy that starts with a longer staleness period and then adjusts it over time. The policy could also consider whether the application is about to run out of memory. Figures 7 and 8 show that the high activation rates for eclipse and lusearch do not significantly impact performance.

The next two columns of Table 1 show that a significant fraction of the heap (often more than half) is stale for a long time, which explains the reductions in collection time observed in Figure 7. Leak tolerance can improve the performance of applications that do not have leaks *per se* but have a large working set that they use only part of for significant periods of time. Leak tolerance can help programmers write applications with large working sets (e.g., larger than physical memory): the programs can keep everything in memory all the time and let leak tolerance reorganize memory, as long as the in-use working set is relatively small and does not change too frequently. Used this way, leak tolerance is analogous to a fine-grained virtual memory manager for managed languages, although we have not evaluated performance in physical memory-limited environments.

The *Stale→in-use* and *Scions* columns show the number of references from stale to in-use objects and the number of scions (i.e., the number of in-use objects referenced by stale objects), respectively. For stale-to-in-use references, we count only references out of the stale space when an object is moved there. A later activation may lead to additional references from stale to the new in-use space, as well as a scion, but we count only the scion and not the new references, which is why in some cases average scions is greater than average references from stale to in-use. The number of scions is relatively small compared to the size of the stale space, which is good since they reside in the in-use space. Each scion is two words long, and during collection each scion also uses about four words in the scion table. In the next section, we show that for growing leaks, the number of scions stays small and fairly constant, while references from stale to in-use grow with the leak, motivating leak tolerance’s use of stub-scion pairs.

### 5.3 Tolerating Leaks

This section shows that a VM enhanced with Melt tolerates several real leaks whereas without Melt, VMs crash. We present two leaks in Eclipse; a leak in Delaunay, a scientific computing application; SPECjbb2000; a leak in Mckoi, a database application; and two third-party microbenchmark leaks: SwapLeak from Sun Developer Network and ToyLeak from IBM developerWorks.

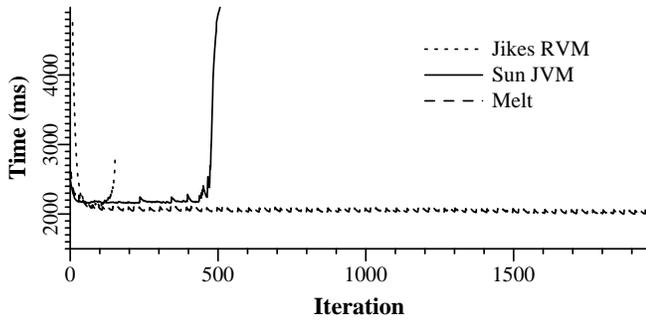
We present all the leaks we found and were able to reproduce, including two Melt does not tolerate. SPECjbb2000 periodically accesses almost all the bytes in its growing data structure, i.e., it leaks both memory and computation. The Mckoi database application leaks threads, but Melt cannot tolerate this leak because threads and their stacks are special VM objects. Relatively modest extensions to Melt would handle this case. Melt is able to tolerate leaks in Eclipse, Delaunay, and the microbenchmarks, whereas the Sun JVM 1.5.0 and Jikes RVM without Melt grind to a halt and crash.

We run with adaptive methodology using fixed heap sizes. Letting the VM grow the heap would allow the programs to run longer. In the wild, programs would outgrow physical memory and thrash, especially in a multiprocessing environment or 64-bit platform. Jikes RVM on IA32 can use at most 1.5 GB of virtual memory, and our machine has 2 GB of physical memory, so we do not evaluate running out of physical memory. The key point is that all programs have a memory ceiling, which may be heap size, physical memory, or virtual memory, and Melt helps leaking programs avoid hitting their ceiling.

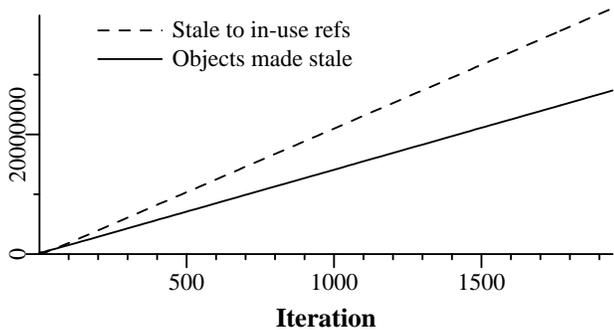
**Eclipse “Diff” Leak** Eclipse is an integrated development environment (IDE) written in Java with over 2 millions lines of source code [14]. We reproduce Eclipse bug #115789, which reports that repeatedly performing a structural (recursive) *diff*, or compare, slowly leaks memory that eventually leads to memory exhaustion. The bug, which exists in Eclipse 3.1.2 but was fixed by developers for Eclipse 3.2, occurs because a data structure for navigation history maintains references it should not.

We automate repeated structural diffs via an Eclipse plugin that reports the wall clock time for each iteration. Figure 9 shows the time each iteration takes, i.e., the effective throughput over time, for vanilla Jikes RVM 2.9.2, the Sun JVM 1.5.0, and Jikes RVM with Melt; all experiments run with a fixed heap size of 256 MB. On both vanilla Jikes and the Sun JVM, Eclipse slows as the heap fills and garbage collection becomes more frequent and expensive. Eclipse eventually crashes with an out-of-memory error on the Sun JVM and Jikes RVM. The Sun JVM runs for more iterations than Jikes because (1) it consumes less memory since Jikes uses heap memory for VM objects, and (2) in our experiments, Jikes uses copying collection, which reserves space for copying, whereas we believe the Sun JVM does not. In contrast, Melt-enabled Jikes RVM keeps running for hundreds more iterations and attains consistent throughput. The stale space eventually exhausts virtual memory, a limitation of our implementation (Section 3.5).

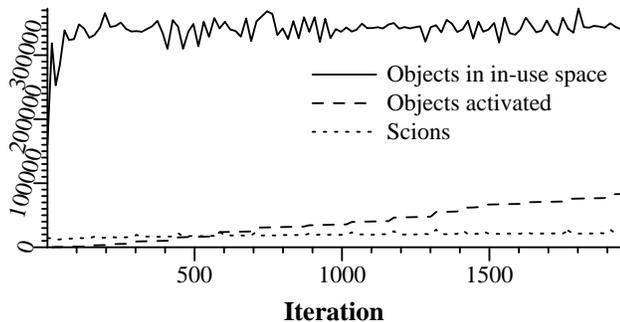
Figures 10 and 11 report numbers of objects and references at each iteration of the “Diff” leak. We divide the data between two graphs since the magnitudes vary greatly. Figure 10 shows that (cumulative) references from stale to in-use and objects made stale both grow linearly over iterations and have 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 11 shows that Melt holds in-use objects relatively



**Figure 9.** Throughput running the Eclipse “Diff” leak: execution time of each diff iteration for *Jikes RVM*, *Sun JVM*, and *Melt* in *Jikes RVM*.



**Figure 10.** Eclipse “Diff” leak with *Melt*: Stale objects and references from stale to in-use, at each iteration.

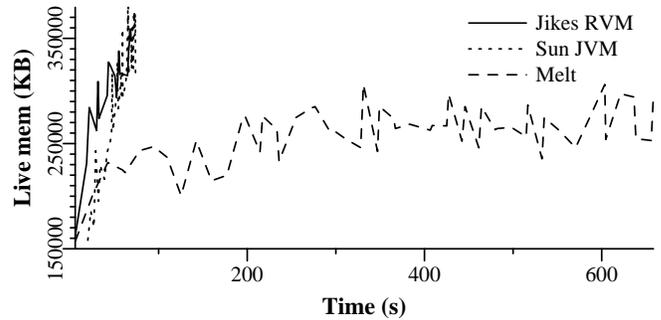


**Figure 11.** Eclipse “Diff” leak with *Melt*: In-Use objects, objects activated, and scions at each iteration.

constant over iterations. Objects activated increases linearly, but its magnitude is small compared with objects made stale. These objects are not leaks but are stale for longer than the steady-state staleness period. After increasing the staleness period early in the run, *Melt* does not increase it further since it increases the period only when the activation rate is greater than 10% (Section 4). The graph shows that scions stay relatively constant over time. This result motivates *Melt*’s use of stub-scion pairs to maintain references from stale to in-use objects.

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

**Eclipse “Copy-Paste” Leak** We also reproduce Eclipse bug #155889, which reports a growing leak when the user repeatedly deletes text, saves, pastes the same text, and saves again. We execute in a 384 MB heap and reproduce the leak manually by exer-



**Figure 12.** Eclipse “Copy-Paste” leak: live memory over time.

	Input size							
	25,000		50,000		75,000		100,000	
	Jikes	Melt	Jikes	Melt	Jikes	Melt	Jikes	Melt
128M	OOM	29	OOM	114	OOM	VSE	OOM	VSE
256M	OOM	11	OOM	75	OOM	VSE	OOM	VSE
384M	9	10	OOM	39	OOM	VSE	OOM	VSE
512M	7	9	OOM	20	OOM	48	OOM	VSE
768M	8	8	17	18	OOM	29	OOM	59
1024M	7	9	14	16	23	26	OOM	38

**Table 2.** Run time in seconds for *Delaunay* as a function of input and heap size. *OOM*: out of memory; *VSE*: virtual space exhaustion by *Melt* for the stale space.

cising the GUI by hand. Figure 12 shows live memory used over time, as reported at the end of each full-heap GC, for the same VM configurations as Figure 9. We show heap growth with respect to time since correlating iterations would be difficult for a manually reproduced leak. The memory footprints of *Jikes RVM* and the *Sun VM* grow roughly linearly over time, and then they crash. *Melt*’s memory footprint grows initially as Eclipse starts and the first copy-paste is performed. Then *Melt* holds in-use memory fairly constant until the stale space exhausts virtual memory.

**Delaunay Mesh Leak** Next we present a leak in *Delaunay*, an application that performs a *Delaunay* triangulation, which generates a triangle mesh for a set of points [17]. 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.

Unlike the Eclipse leaks, this leak is not a growing leak in a long-running program. Rather, this leak degrades program performance and prevents the program from running under input sizes and heap sizes that would work without the leak. To highlight this problem, we execute the program with a variety of input and heap sizes and show results in Table 2: run time, *OOM* (out of memory crash), or *VSE* (virtual space exhaustion crash). Even on the smallest input size, if the heap is small, the program crashes without *Melt*. With *Melt*, some input sizes and heap sizes exhaust *Jikes RVM*’s virtual memory. Even with *Melt*, *Delaunay* slows down because the program activates stale memory too often. However, given sufficient heap space, *Melt* uniformly runs larger input sizes whereas the program would otherwise crash. Once the leak has been fixed, we plan to compare the fixed version’s performance with *Melt* running the leaky version.

**ToyLeak from IBM developerWorks** We also obtained a leak from an IBM developerWorks column [18]. We call it *ToyLeak*. It is relatively simple—just 55 lines—but includes two different leaks. One is a slow-growing leak due to an off-by-one error that leads to an object not being removed from a *Vector*. The other leak

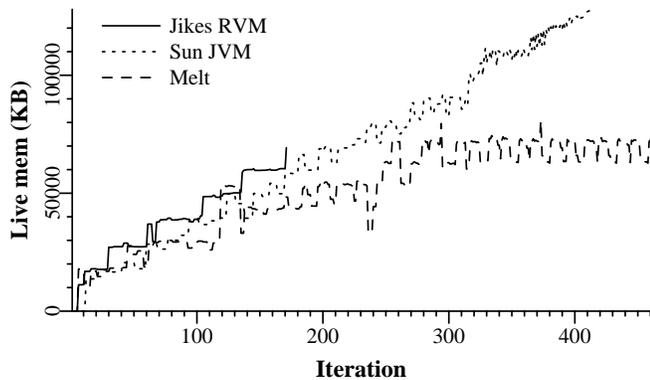


Figure 13. ToyLeak: Live memory at each iteration.

	48M	64M	80M	96M	112M	128M	144M
Jikes	Early	Early	Leak	Leak	Leak	1.3	1.5
Melt	5.2	4.9	4.4	3.7	3.0	1.5	1.6

Table 3. Run times in seconds for SwapLeak, without and with Melt, for various heap sizes. *Early*: out of memory before leaky code executed; *Leak*: out of memory while executing leaky code.

repeatedly adds objects to a set, does not remove them, and it grows somewhat more quickly. The program executes in iterations and exercises both leaks during each iteration.

The time for each iteration is not representative of the leak’s progress since the example calls `sleep` and uses `Vector.removeElementAt()`, which takes time proportional to the vector’s size, so we report live memory instead. Figure 13 shows the amount of (in-use) memory in the heap per iteration for Jikes RVM, Sun JVM, and Melt, running in a 128 MB heap. The graph shows that Melt tolerates this leak and maintains the live memory footprint.

**SwapLeak from Sun Developer Network** The following leak comes from a message posted on the Sun Developer Network [42]. The message asks for help understanding why the author’s program runs out of memory. The program first initializes an array of `SObject`s, which each contain an inner class `Rep`. Second, the program 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 memory. However, a response to the message explains that VMs keep 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, but Melt provides the illusion of a fix without needing to understand or apply the fix.

Table 3 shows run times (seconds) for unmodified Jikes RVM and Melt running the leak with various heap sizes. *Early* means the program runs out of memory during the program’s first step, so 48-64 MB is not enough to run the program, with or without the leak. Although this is not the intended use of Melt, it runs the program at these sizes because it moves non-leaking data to the stale space, which is later activated. *Leak* means the program runs out of memory during its second, leaking step. Melt tolerates these cases by moving the leaked objects to the stale space. For larger heap sizes, both VMs execute the program, and Melt performs 7-15% slower than unmodified Jikes RVM.

**SPECjbb2000 Leak** `SPECjbb2000` simulates an order processing system and is intended for evaluating server-side Java performance [41]. It contains a known, growing memory leak that manifests when it runs for a long time without changing ware-

houses. It leaks because it adds but does not correctly remove some orders from an order list that should have no net growth.

Although `SPECjbb2000` experiences unbounded heap growth over time, it is not strictly a memory leak since the vast majority of bytes leaked are in-use: the program periodically accesses all orders in the order list. Thus the leak is also a *computation leak*. It is difficult to imagine a solution to this kind of problem, since the system has no way to differentiate memory used in a useful way and used in a useless way.

We execute on Melt and find it allows `SPECjbb2000` to run somewhat longer than with unmodified Jikes RVM because about 20% of memory is stale throughout the run. Melt keeps this memory in the stale space, but growing in-use memory eventually causes an out-of-memory error, albeit somewhat later than without Melt.

**Mckoi SQL Database Leak** We reproduced a memory leak reported on a message board for Mckoi SQL Database, a database management system written in Java [27]. 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 thread associated with the connection, 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 the stack 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.

## 6. Conclusion

Garbage collection and type safety save programmers from many memory bugs, but dead, reachable objects hurt performance and crash programs. Given enough disk space, our leak tolerance approach keeps programs from slowing down and running out of memory. Its key properties are keeping time and space proportional to in-use memory, rather than leaked memory, and preserving safety by allowing the program to activate an object. Our Melt implementation adds low enough overhead for deployed use, and it provides modest locality and GC benefits for ordinary programs. For growing leaks in real programs, Melt avoids running out of memory, although our prototype implementation fails when the stale space exhausts virtual memory.

With plenty of disk space, leak tolerance frees developers and users from worrying too much about memory leaks in managed languages. It buys developers more time to fix leaks, and it keeps users happy by providing the illusion there is no leak.

## References

- [1] 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, S. F. 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.
- [2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.
- [3] 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.
- [4] BEA. JRockit Mission Control. <http://dev2dev.bea.com/jrockit/-tools.html>.

- [5] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM Conference on Programming Language Design and Implementation*, pages 158–168, 2006.
- [6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ACM International Conference on Software Engineering*, pages 137–146, 2004.
- [7] 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. Wiederemann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.
- [8] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *ACM International Symposium on Memory Management*, pages 143–151, 2004.
- [9] M. D. Bond and K. 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.
- [10] 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.
- [11] W. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-guided Proactive Garbage Collection for Locality Optimization. In *ACM Conference on Programming Language Design and Implementation*, pages 332–340, 2006.
- [12] T. M. Chilimbi and M. 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.
- [13] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):965–975, Nov. 1978.
- [14] Eclipse.org Home. <http://www.eclipse.org/>.
- [15] B. Goetz. Plugging memory leaks with weak references, 2005. <http://www-128.ibm.com/developerworks/java/library/j-jtp11225/>.
- [16] M. Goldstein, O. Sheury, and Y. Weinsberg. Can Self-Healing Software Cope With Loitering? In *International Workshop on Software Quality Assurance*, pages 1–8, 2007.
- [17] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized Incremental Construction of Delaunay and Voronoi Diagrams. In *Colloquium on Automata, Languages and Programming*, pages 414–431, 1990.
- [18] S. C. Gupta and R. Palanki. Java memory leaks – Catch me if you can, 2005. [http://www.ibm.com/developerworks/rational/library/05/0816\\_GuptaPalanki/index.html](http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/index.html).
- [19] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Winter USENIX Conference*, pages 125–136, 1992.
- [20] D. L. Heine and M. 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.
- [21] M. Hertz, Y. Feng, and E. D. Berger. Garbage Collection without Paging. In *ACM Conference on Programming Language Design and Implementation*, pages 143–153, 2005.
- [22] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80, 2004.
- [23] Jikes RVM. <http://www.jikesrvm.org>.
- [24] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [25] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Java. In *ACM Symposium on Principles of Programming Languages*, pages 31–38, 2007.
- [26] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise Detection of Memory Leaks. In *International Workshop on Dynamic Analysis*, pages 25–31, 2004.
- [27] 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>.
- [28] 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.
- [29] N. Nethercote and J. Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, 2007.
- [30] H. H. Nguyen and M. Rinard. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ACM International Symposium on Memory Management*, pages 15–29, 2007.
- [31] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. 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.
- [32] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: A Real-Time Garbage Collector for Multiprocessors. In *ACM International Symposium on Memory Management*, pages 159–172, 2007.
- [33] D. Plainfossé. *Distributed Garbage Collection and Reference Management in the Soul Object Support System*. PhD thesis, Université Paris-6, Pierre-et-Marie-Curie, 1994.
- [34] 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.
- [35] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In *ACM Symposium on Operating System Principles*, pages 235–248, 2005.
- [36] Quest. JProbe Memory Debugger. <http://www.quest.com/jprobe/debugger.asp>.
- [37] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. Beebe. Enhancing Server Availability and Security through Failure-Oblivious Computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 303–316, 2004.
- [38] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC<sup>2</sup>: High-Performance Garbage Collection for Memory-Constrained Environments. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 81–98, 2004.
- [39] SciTech Software. .NET Memory Profiler. <http://www.scitech.se/memprofiler/>.
- [40] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.
- [41] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.
- [42] Sun Developer Network Forum. Java Programming [Archive] - garbage collection dilemma (sic), 2003. <http://forum.java.sun.com/thread.jspa?threadID=446934&forumID=31>.
- [43] P. 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.
- [44] F. Xian, W. Srisa-an, and H. 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.
- [45] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic Heap Sizing: Taking Real Memory into Account. In *ACM International Symposium on Memory Management*, pages 61–72, 2004.
- [46] B. Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, University of Colorado at Boulder, 1990.