Tolerating Memory Leaks *

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, a program still leaks memory if it maintains references to objects it will never use again. Leaked objects decrease program locality and increase garbage collection frequency and workload. A growing leak will eventually exhaust memory and crash the program.

This paper introduces a *leak tolerance* approach called *Melt* that safely eliminates performance degradations and crashes due to leaks of dead but reachable objects in managed languages, given sufficient disk space to hold leaking objects. Melt (1) identifies *stale* objects that the program is not accessing; (2) segregates in-use and stale objects by storing stale objects to disk; and (3) preserves safety by activating stale objects if the program subsequently accesses them. We design and build a prototype implementation of Melt in a Java VM and show it adds overhead low enough for production systems. Whereas existing VMs grind to a halt and then crash on programs with leaks, Melt keeps many of these programs running much longer without significantly degrading performance. Melt provides users the illusion of a fixed leak and gives developers more time to fix leaky programs.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Error handling and recovery

General Terms Reliability, Performance, Experimentation

* This work was supported by an Intel PhD Fellowship, NSF CNS-0719966, NSF CCF-0429859, NSF EIA-0303609, DARPA F33615-03-C-4106, Intel, CISCO, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

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 double (repeat) frees and premature frees that leave dangling pointers. They also eliminate the effort required to insert frees correctly, improving programmer productivity. Unfortunately, programmers may neglect to eliminate pointers to objects the program will never use again. These objects are *leaked* because garbage collection uses *reachability* as an over-approximation of *liveness*. Leaks increase garbage collection frequency and workload and may hurt application performance by bloating the working set size. 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 [24]. Leaks in managed languages are a problem in practice and a number of recent tools help developers diagnose and fix leaks [44, 9, 34, 38, 49, 52]. Unfortunately, leaks occur in deployed software because, even with these tools, leaks sometimes escape detection by developers. The goal of *leak tolerance* is to *provide users the illusion there is no leak*—the program does not slow to a halt or crash. Leak tolerance is not a replacement for fixing leaks; it gives developers more time and information to fix leaks while improving the user experience.

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

Melt keeps programs performing well by guaranteeing time and space remain proportional to *in-use* (non-leaked) memory. It restricts the application and garbage collector to accessing in-use objects in memory and prohibits accesses to stale objects on disk, and it keeps metadata proportional to in-use memory. Other leak tolerance approaches for garbage-collected languages do not provide this guarantee [11, 21, 57]. *Bookmarking collection* is similar to Melt in that it restricts collection to in-use pages, but it operates at page granularity [26], whereas Melt seeks to tolerate leaks with fine-grain object tracking and reorganization.

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

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

2. Leak Tolerance with Melt

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

1. Time and space overheads are proportional to the in-use memory, not leaked memory.
2. Melt provides safety by preserving and, if needed, activating stale objects.

Furthermore, Melt adheres to the following *invariants*:

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

We satisfy these invariants as follows: (1) Melt identifies stale objects (Section 2.1); (2) it segregates stale objects from in-use objects by moving stale objects to disk, and it uses double indirection for references from stale to in-use objects (Section 2.2); and (3) it intercepts program attempts to access objects in the stale space and immediately moves the object into the in-use space. (Section 2.3). Section 2.4 presents how Melt decides when and which stale objects to move to disk, based on how close the program is to running out of memory.

2.1 Identifying Stale Objects

We classify reachable objects that the program has not referenced in a while as *stale*. If the program never accesses them again, they are true leaks. As we show later, some leaks manifest as in-use (live) objects. For example, the program forgets to delete objects from a hash table, keeps adding objects, and then rehashes all elements every time it grows beyond the current limit. Staleness thus under-approximates leaks of in-use objects.

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

The compiler adds instrumentation called a *conditional read barrier* [8] to every load of an object reference. The barrier checks whether the reference is stale. If it is stale, the barrier unmarks the referenced object and the reference. The following pseudocode shows the barrier:

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

This conditional barrier reduces overhead since it performs stores only the first time the application loads each reference in each mutator epoch (the *mutator* is the application alone, not the collector). Checking for a marked *reference*, rather than a marked *object*, reduces overhead since it avoids an extra memory load.

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

At the next collection, each object will be marked stale if and only if the application did not load a reference to it since the previous collection. Figure 1 shows an example heap with stale (shaded gray) objects C and D. They have not been accessed since the last collection, because all their incoming references are stale, marked with S. Although B has incoming stale references, B is in-use because the reference A $\rightarrow$ B is in-use.

2.2 The Stale Space

When the garbage collection traces the heap, it now also moves stale objects to the *stale space*, which resides on disk. For example, the collector moves stale objects C and D from Figure 1 to the stale space, as illustrated by Figure 2.

Stub-scion pairs. References from stale objects to in-use objects are problematic because moving collectors such as copying and compacting collectors move in-use objects. For example, consider moving B, which has references from C and D. If B moves, we do not want to touch stale objects to update their outgoing references, which would violate the invariants. We solve this problem by using *stub-scion pairs*, borrowed from distributed garbage collection [46]. Stub-scion pairs provide two levels of indirection. Melt creates a *stub* object in the stale space and a *scion* object in the in-use space for each in-use object that is referenced by one or more stale object(s). The collector avoids touching stubs and stale objects by referencing and updating the scion. The stub has a single field that points to the scion.

The scion has two fields: one points to the in-use object and the other points back to its stub. We modify references in the stale space that refer to an in-use object to refer instead to the stub. Figure 3 shows B_{stub} and B_{scion} providing two levels of indirection for references from C and D to B. Scions may not move. The collector treats scions as roots, retaining in-use objects referenced by stale objects. If the collector moves an object referenced by a scion, it updates the scion to point to the moved object.

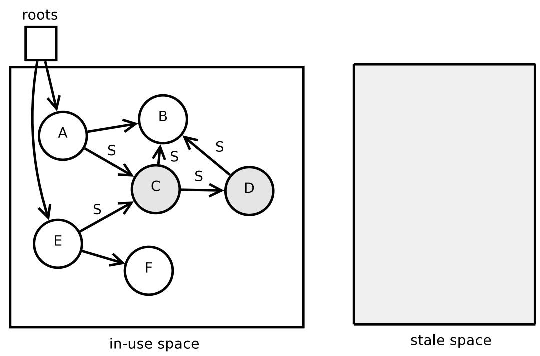

Figure 1. Stale Objects and References

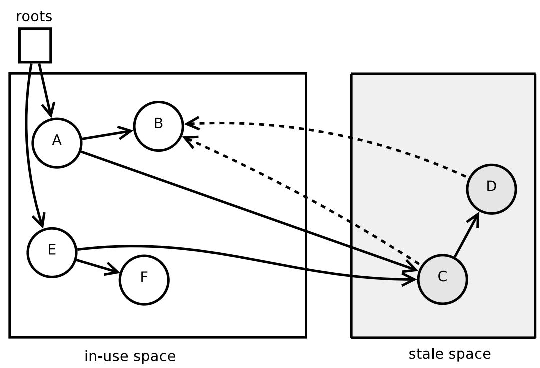

Figure 2. Segregation of In-Use and Stale Objects

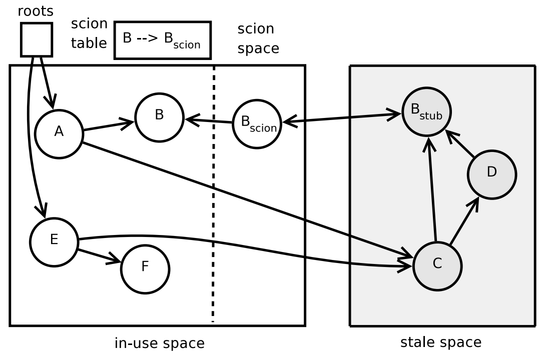

Figure 3. Stub-Scion Pairs

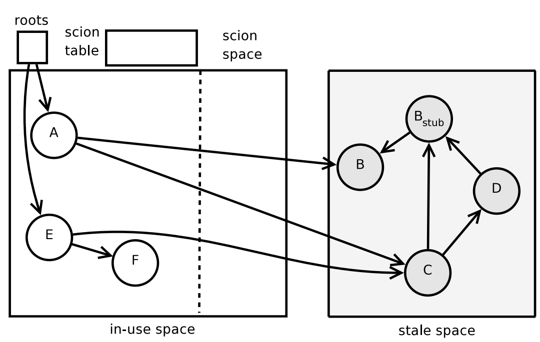

Figure 4. Scion-Referenced Object Becomes Stale

To ensure each in-use object has only one stub-scion pair, we use a *scion lookup table* that maps from an in-use object to its scion, if it has one. This data structure is proportional to the number of scions, which is proportional to the number of in-use objects in the worst case, but is usually much smaller in practice. The collector processes the scions at the beginning of collection. Returning to Figure 2, when the collector copies C to the stale space, B initially has no entry in the scion lookup table, so Melt adds a mapping B $\rightarrow B_{scion}$ to the table when it creates B_{stub} and B_{scion}. Next, when it copies D to the stale space, it finds the mapping B $\rightarrow B_{scion}$ in the table and re-uses the existing stub-scion pair. The resulting system snapshot is shown in Figure 3.

It may seem at first that we need scions but not necessarily stubs, i.e., stale objects could point directly to the scion. However, we need both because an in-use object referenced by a scion may become stale later. For example, consider the case when B becomes stale in Figure 3. In order to eliminate the scion without a stub (to avoid using in-use memory for stale-to-stale references), we would need to find all the stale pointers to the scion, which violates the stale space invariant to never visit stale objects after instantiation. Instead, Melt copies B to the stale space, looks up the stub location in the scion, and points the stub to stale B. Note that Melt accesses the disk both to modify the stub and to move the new stale object. This accesses do not violate invariants since are part of moving an object to the stale space. Melt then deletes the scion and removes the entry in the scion lookup table. Figure 4 shows the result.

2.3 Activating Stale Objects

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

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

The VM method `activateStaleObject()` copies the stale object to the in-use space. Since other references may still point to the stale version, `activateStaleObject()` creates a stub-scion pair for the activated object as follows: (1) it converts the stale space object version into a stub, and

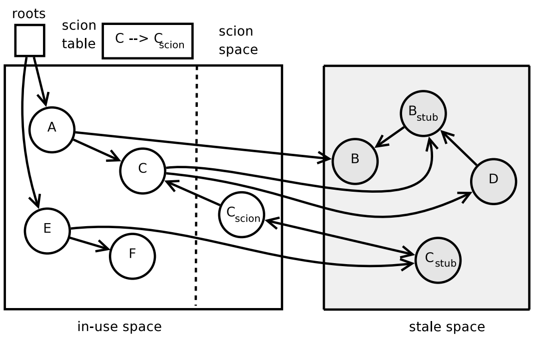

Figure 5. Stale Object Activation

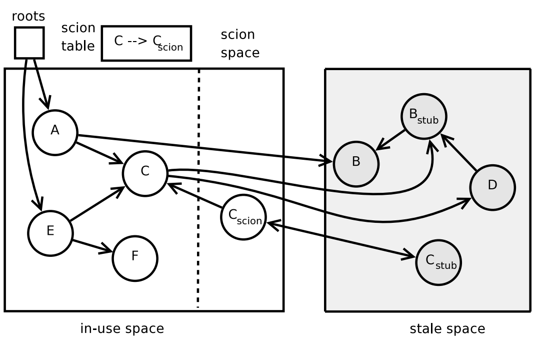

Figure 6. Reference Updates Following Activation

(2) it creates a scion and points the stub at the scion. The scion points to the activated object. The store to `a.f` must be atomic with respect to the original value of `b`, i.e., `[iff a.f == t]`.

Consider activating C from Figure 4. First, `activateStaleObject()` copies C to the in-use space. Then it replaces stale C with a stub, allocates a scion, and links them all together, as shown in Figure 5. Note that C retains its references to D and B_{stub}, and E retains its reference to the old version of C, which is now C_{stub}.

If the application later follows a different reference to the previously stale object in the stale space, `activateStaleObject()` finds the stub in the object's place, which it follows to the scion, which in turn points to the activated object. The first access of such a reference will update the reference to point to the activated version and any subsequent accesses will go directly to the in-use object. For example, if the application accesses a reference from E to C_{stub} in Figure 5, `activateStaleObject()` follows C_{stub} to C_{scion} to C in the in-use space and updates the reference, as shown in Figure 6.

2.4 When to Move Objects to the Stale Space

Melt can mark objects as stale and/or move objects to the stale space on any full-heap garbage collection. However, it does not make sense to incur this overhead if the application is not leaking memory. Furthermore, Melt could potentially fill the disk for a non-leaking application, producing an error

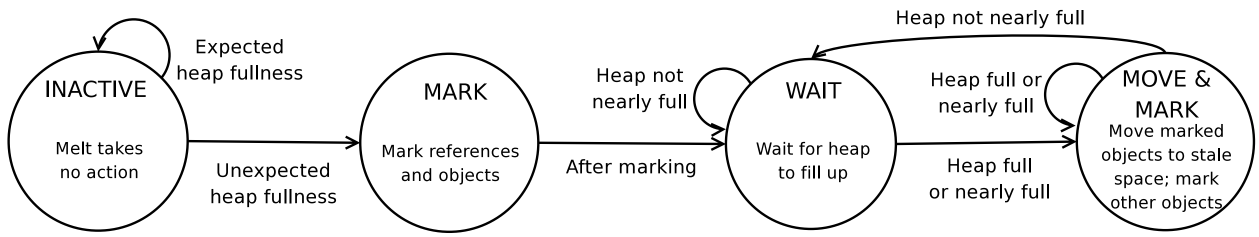

Figure 7. State diagram for when Melt marks objects as stale and moves objects to the stale space.

where none would have existed. Thus, Melt decides whether to mark and move based on *how full the heap is* as shown in Figure 7.

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

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

3. Implementation Details

This section presents details specific to our implementation. Our approach is suitable for garbage-collected, type-safe languages using tracing garbage collectors. We call our implementation Melt for simplicity. We implement Melt in Jikes RVM 2.9.2, a high-performance Java-in-Java virtual machine [1, 2, 32]. The DaCapo benchmark regression tests page shows that Jikes RVM performs the same as Sun Hotspot 1.5 and 15–20% worse than Sun 1.6, JRockit, and J9 1.9, all configured for high performance [16]. Our performance measurements are therefore relative to an excellent baseline.

We have made Melt publicly available on the Jikes RVM Research Archive [33].

3.1 VM Issues

Garbage collection. Melt's design is compatible with moving and non-moving tracing collectors, such as copying, compaction, and mark-sweep, all of which are in use by modern high-performance VMs. To demonstrate the flexibility and generality of Melt, we use a high-performance generational copying collector. Since Melt correctly handles moving in-use objects, the most challenging case, it can easily handle non-moving collectors as well. The generational 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 mature copy reserve.

Jikes RVM's memory manager, the Memory Management Toolkit (MMTk) [6], supports a variety of garbage collectors with most functionality residing in shared code. Melt resides mostly in this shared code. To support another collector, one must implement a method that specifies (1) the space(s) that contain potentially stale objects and (2) the space into which to 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. Jikes RVM uses two compilers: an initial baseline compiler and an optimizing compiler invoked for hot methods at successively higher optimization levels. Both add Melt read barriers. For efficiency and simplicity, we exclude VM objects and objects directly pointed to by roots (registers, stacks, and statics) as candidates for the stale space.

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

Activating stale objects. Melt uses read barriers to intercept application reads to the stale space (Section 2.3). Melt immediately copies the object to the mature space (or a large object space if the object is large) and updates the reference. Since activation allocates into the in-use part of the heap, it may trigger a garbage collection (GC). Application reads are not necessarily *GC-safe points*. GC-safe points require the VM to be able to enumerate all the pointers into the heap, i.e., to produce a stack map of the local, global, and temporary variables in registers. In Jikes RVM (as in many other VMs), allocations, method entries, method exits, and loop back edges are GC-safe points. If an activation triggers a GC, Melt defers collection by requesting an *asynchronous collection*, which causes collection to occur at the next GC-safe point.

3.2 Stale Space on Disk

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

Figures 8 and 9 show the 64-bit on-disk stale space representation for Figures 5 and 6. The main difference is the mapping stub space, which provides indirection for references from the in-use space to the stale space. Three types of references are 64 bits: mapping stubs, references in stale objects, and pointers from scions to their stubs. If a stale object references an in-memory object, e.g., $C_{stub} \rightarrow C_{scion}$ in Figure 9, the reference uses only the lower 32 bits.

We use *swizzling* [39, 58] to convert references between 32-bit in-memory and 64-bit on-disk addresses. When the collector moves an object to the stale space, it *unswizzles* outgoing reference slots. If a slot references a mapping stub, the collector stores the target of the mapping stub in the slot, in order to avoid using in-use memory (the mapping stub) for an intra-disk reference. When a read barrier activates an object in the stale space, it *swizzles* outgoing references by creating the mapping stub for each slot that references a 64-bit object. When the application activates C in Figure 8, Melt swizzles its references to B_{stub} and D by creating mapping stubs BS_{ms} (mapping stub of B_{stub}) and D_{ms}, and redirecting the references through them, as shown in Figure 9.

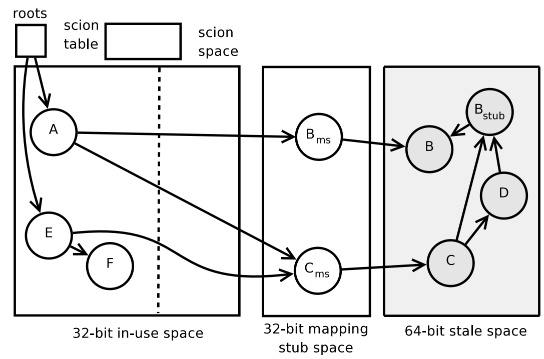

Figure 8. Figure 4 with On-Disk Stale Space

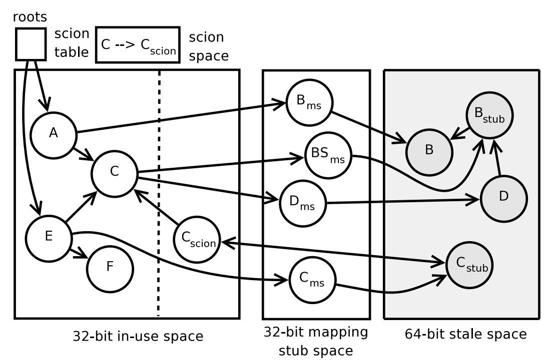

Figure 9. Figure 5 with On-Disk Stale Space

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

3.3 Multithreading

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

Stale space accesses occur when the collector moves an object to the stale space or the application activates a stale object. Melt increases parallelism by using one file per collector thread (there is one collector thread per processor) and by using thread-local buffers for stale objects before flushing them to disk. Each thread allocates stale objects to a different part of the 64-bit stale address range: the high 8 bits of the address specify the thread ID.

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

3.4 Saving Stale Space

This section discusses approaches for reducing the size of the stale space. We have not implemented these approaches. With Melt as described, garbage collection is *incomplete* because it does not collect the stale space. Stale objects may become unreachable after they are moved to the stale space and furthermore, they may refer to in-use objects. These uncollectible in-use objects will eventually move to the stale space since they are inherently stale. For example, even if C in Figure 6 becomes unreachable, the scion will keep it alive and it will eventually move to the stale space. One solution would be to reference-count the stale space, but reference counting cannot collect cycles. Alternatively, Melt could occasionally trace all memory including the stale space. An orthogonal approach would be to compress the stale space [12]. The stale space is especially suitable for compression compared with a regular heap because the stale space is accessed infrequently.

4. Results

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

4.1 Performance 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. For example, compilation allocates memory and perturbs garbage collection workload. To eliminate this source of nondeterminism, we use *replay* methodology [31, 43, 51]. Replay uses advice files to force the VM to compile the same methods at the same level and point in execution and with the same profile information executions and thus avoids high variability due to sampling-driven compilation.

Benchmarks. To measure Melt's overhead, we use the DaCapo benchmarks version 2006-10-MR1, a fixed-workload version of SPECjbb2000 called pseudojbb, and SPECjvm98 [7, 53, 54].

Platform. Performance experiments execute on a dual-core 3.2 GHz Pentium 4 system with 2 GB of main memory running Linux 2.6.20.3. Each core has a 64-byte L1 and L2 cache line size, a 16-KB 8-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, and a 1-MB unified 8-way set associative L2 on-chip cache. The top four leaks in Table 2 execute on a Core 2 Quad 2.4 GHz system with 2 GB of main memory running Linux 2.6.20.3, with 126 GB of free disk space. Each core has a 64-byte L1 and L2 cache line size, an 8-way 32-KB L1 data/instruction cache, and each pair of cores shares a 4-MB 16-way L2 on-chip cache.

4.2 Melt's Overhead

Application overhead. Figure 10 presents the run-time overhead of Melt. We run each benchmark in a single medium heap size, two times the minimum in which it can execute. Each bar is normalized to *Base* (an unmodified VM) and includes application and collection time, but not compilation time. Each bar is the median of five trials; the thin error bars show the range of the five trials. For all experiments, except for some bloat experiments, run-to-run variation is quite low since replay methodology eliminates almost all nondeterminism. The variation in bloat is high in general and not related to these configurations. The bottom sub-bars are the fraction of time spent in garbage collection.

Barriers includes only Melt's read barrier; the barrier's condition is never true since the collector does not mark references stale. *Marking* performs marking of references and objects *on every full-heap GC*, i.e., Melt is always in the MARK state (Section 2.4). *Melt memory* performs marking and moving to the stale space on every full-heap GC (i.e., Melt is always in the MOVE & MARK state), but the stale space is in memory rather than on disk. This configuration is analogous to adding a third generation based on object usage in a generational collector. Finally, *Melt* marks objects and moves objects to the on-disk stale space on every full-heap GC.

The graph shows that the read barrier alone costs 6% on average, and adding *Marking* adds no noticeable overhead. The *Melt memory* configuration, which divides the heap into in-use and in-memory stale spaces, has a negligible effect on overall performance. In fact, it sometimes improves collector performance (see below). Storing stale objects on disk (*Melt*) adds 1% to average execution time because of the extra costs of swizzling between 32- and 64-bit references and transferring objects to and from disk. Melt improves the performance of a few programs relative to barrier overhead. This improvement comes from better program locality (jython and lusearch) and lower GC overhead (xalan).

Melt's 6% read barrier overhead is comparable to read barrier overheads for concurrent, incremental, and real-time collectors [4, 17, 45], 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

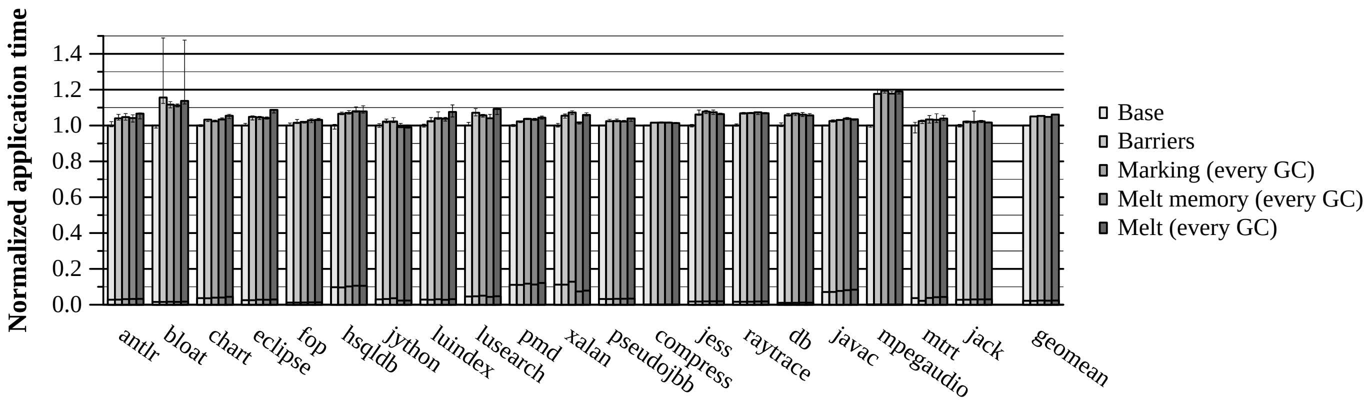

Figure 10. Application execution time overhead of Melt configurations. Sub-bars are GC time.

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

Table 1. Statistics for the DaCapo benchmarks running *Melt (every GC)* with 1.5 times the minimum heap size.

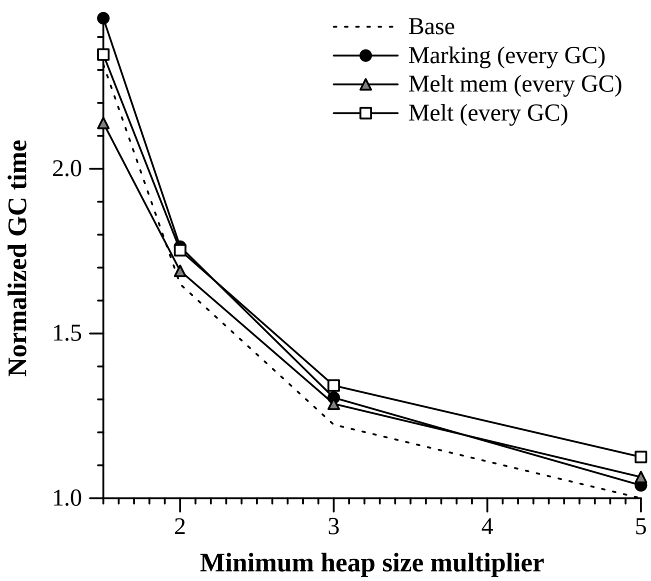

Figure 11. Normalized GC times for Melt configurations across heap sizes.

and a branch. An alternative to all-the-time read barriers would be to start without read barriers and recompile all methods only when the program entered the MARK state.

Collection overhead. Figure 11 shows the geometric mean of the time spent in garbage collection as a function of heap size for all our benchmarks using Melt. We measure GC times at 1.5x, 2x, 3x, and 5x the minimum heap size for each benchmark. Times are normalized to *Base* with 5x min heap. Note that the y-axis starts at 1 and not 0.

The graph shows *Marking* slows collection by up to 7% for the smaller heap sizes. The other configurations measure both the overhead and benefits of using the stale space. *Melt memory*, which enjoys the benefits of reduced GC workload and frequency due to stale space discounting, speeds collection 15% over *Marking* and 8% over *Base* for the 1.5x heap. *Melt* adds up to 10% GC overhead over an in-memory stale space due to pointer swizzling and transferring objects to and from disk. This configuration adds up to 10% over the baseline in large heaps, but benefits and costs are roughly equal at the smallest heap size, where Melt nets just 1% over the baseline.

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

Melt statistics. Table 1 presents stale, in-use, and other statistics for Melt running the DaCapo benchmarks and marking and moving objects every full-heap GC (i.e., the

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

Table 2. Ten leaks and Melt's ability to tolerate them.

Melt configuration used in Figures 10 and 11). We run with a small heap, 1.5 times the minimum heap size for each benchmark, in order to trigger frequent collections and thus exercise Melt more heavily. The table presents the total number of objects moved to the stale space and activated by the program. It also shows objects in the in-use and stale spaces, pointers from in-use to stale and from stale to in-use, and scions, averaged over each full-heap GC except the first, which we exclude since it does not move any objects to the stale space. The final column is the number of full-heap GCs. We exclude fop and hsqldb since they execute fewer than two full-heap GCs.

The table shows that Melt moves 9–151 MB to the stale space, and the program activates 0–17 MB of this memory. Some benchmarks activate a significant fraction of stale memory, for example, more than 10% for bloat, eclipse, and lusearch due to this experiment's aggressive policy of moving objects to the stale space on every GC. The next two columns of Table 1 show that often more than half of the heap is stale for a long time, which explains the reductions in collection time observed in Figure 10. Leak tolerance can improve the performance of applications that do not have leaks *per se* but only use a small portion of a larger working set for significant periods of time. Used this way, leak tolerance is analogous to a fine-grained virtual memory manager for managed languages.

The *In*→*St* column shows the average number of references from in-use to stale objects. These references require a mapping stub to redirect from 32-bit memory to 64-bit disk, but there are usually signficantly fewer mapping stubs than in-use objects. The *St*→*In* and *Scions* columns show the number of references from stale to in-use objects and the number of scions, respectively. The next section shows that for growing leaks, the number of scions stays small and proportional to in-use memory, while references from stale to in-use grow with the leak, motivating leak tolerance's use of stub-scion pairs.

4.3 Tolerating Leaks

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

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

We run the following experiments in maximum heap sizes chosen to be about twice what each program would need if it were not leaking. All the programs except Delaunay have growing leaks, so their behavior with and without Melt is not very sensitive to maximum heap size. All programs have a memory ceiling, which may be heap size, physical memory, or virtual memory, although physical memory is always a ceiling since it causes GC to thrash [26, 60]. Melt extends a progam's memory ceiling to include all available disk space, substantially postponing a crash. We run Jikes

RVM in uniprocessor mode because in multiprocessor mode, the VM often crashes before completing runs lasting many hours, apparently due to bugs in Melt or Jikes RVM.

EclipseDiff. Eclipse is an integrated development environment (IDE) written in Java with over 2 millions lines of source code [18]. We reproduce Eclipse bug #115789, which reports that repeatedly performing a structural (recursive) *diff*, or compare, slowly leaks memory that eventually leads to memory exhaustion. The leak occurs because a data structure for navigation history maintains references it should not. It exists in Eclipse 3.1.2 but was fixed by developers for Eclipse 3.2 after we reported a fix in previous leak detection work [9].

We automate repeated structural differences via an Eclipse plugin that reports the wall clock time for each iteration of the difference. Figure 12 shows the time each iteration takes for vanilla Jikes RVM 2.9.2, the Sun JVM 1.5.0, and Jikes RVM with Melt. We use iterations as the x-axis. This figure shows the first 300 iterations in order to compare the three VMs, and Figure 13 shows the performance of just Melt for its entire run (terminated by us after 24 hours). Unmodified Jikes RVM slows and crashes after about 50 iterations when its heap fills. Sun JVM, which uses a more space-efficient collector than the generational copying collector used by Jikes in our experiments, runs almost 200 iterations before grinding to a halt and crashing.

Melt's performance stays steady in the long term with variations in the short term. All VMs' performance varies per iteration because iterations interrupted by a full-heap GC take longer. Melt's performance varies more because full-heap GCs that move objects to the stale space take longer: Melt moves objects to the stale space, unswizzles their references, and creates stub-scion pairs and mapping stubs. Melt buffers new stale objects in memory during these GCs, and it flushes these buffers to disk gradually during application execution. Without this gradual flushing, performance varies more. When we terminate Melt at 24 hours, it has written over 80 GB to the on-disk stale space.

Figures 14 and 15 show reachable memory, as reported at the end of the last full-heap GC, for the same VMs at each iteration. Unmodified Jikes RVM and Sun JVM fill the heap as the leak grows, while Melt starts moving stale objects to the disk when the heap reaches 80% full, and it keeps memory usage fairly constant in the long term. The figures show that memory usage oscillates gradually between about 100 and 130 MB: (1) Melt moves objects to *buffers* for the stale space when usage reaches 130 MB; (2) it then slowly flushes these buffers to disk over time; and (3) in the meantime, the leak continues to increase heap size until it reaches 130 MB again and triggers Melt to repeat the cycle.

Figures 16 and 17 report numbers of objects and references at each iteration of the EclipseDiff. We divide the data between two graphs since the magnitudes vary greatly. Figure 16 shows that references from stale to in-use and objects

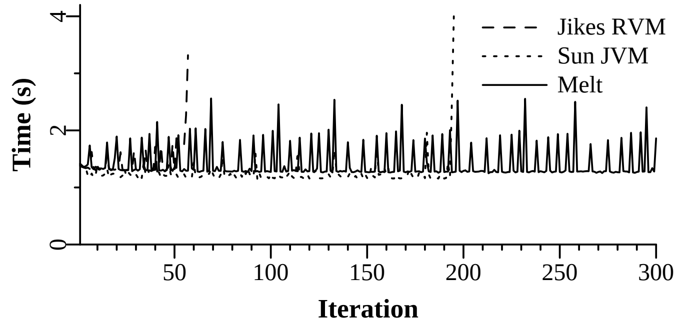

Figure 12. Performance comparison of *Jikes RVM*, *Sun JVM*, and *Melt* for the first 300 iterations of EclipseDiff.

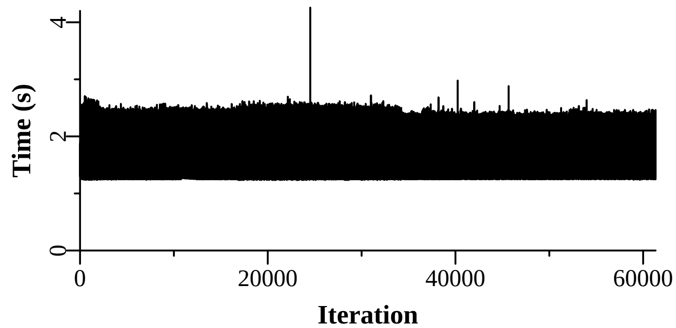

Figure 13. Performance of *Melt* running EclipseDiff leak for 24 hours.

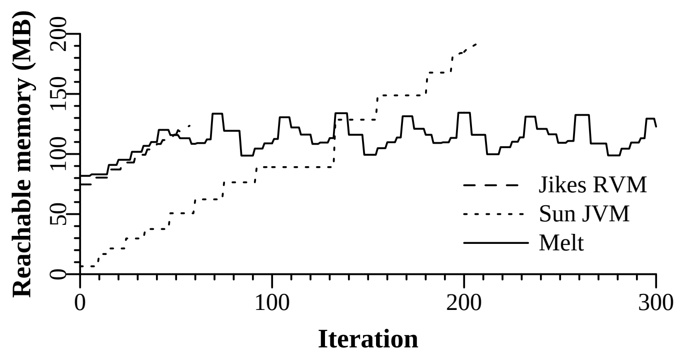

Figure 14. Comparison of reachable memory for the first 300 iterations of EclipseDiff.

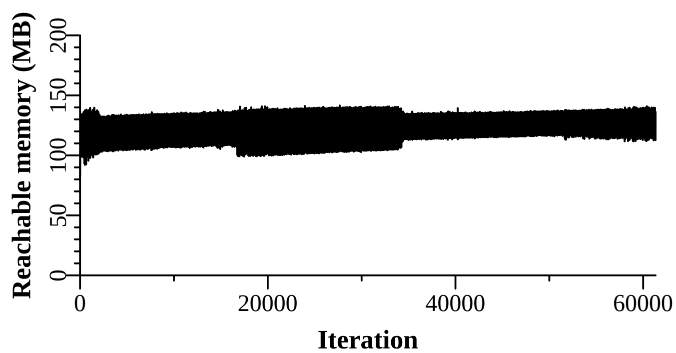

Figure 15. Reachable memory running EclipseDiff with Melt for 24 hours.

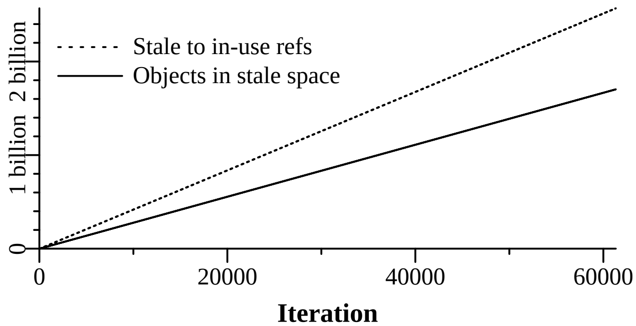

Figure 16. EclipseDiff leak with Melt: stale objects and references from stale to in-use.

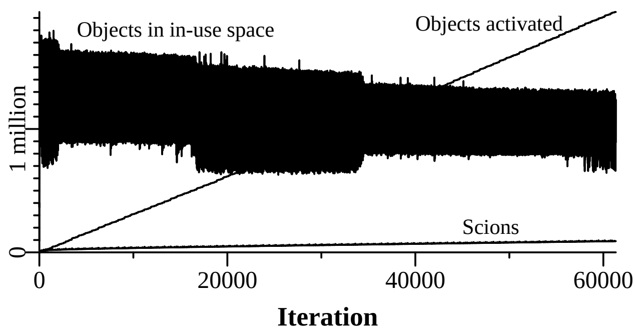

Figure 17. EclipseDiff leak with Melt: in-use objects, objects activated, and scions.

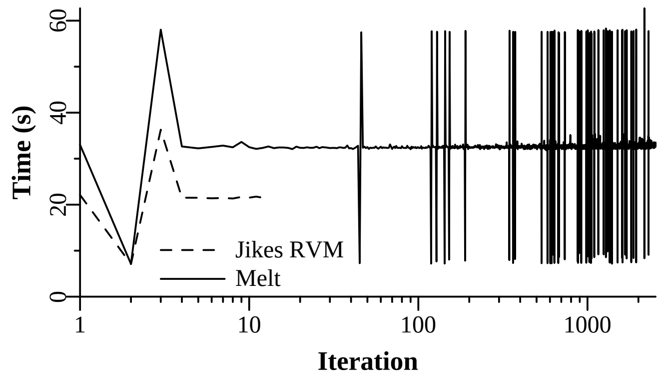

Figure 18. EclipseCP performance over time, with and without Melt (logarithmic x-axis to show behavior of both VMs).

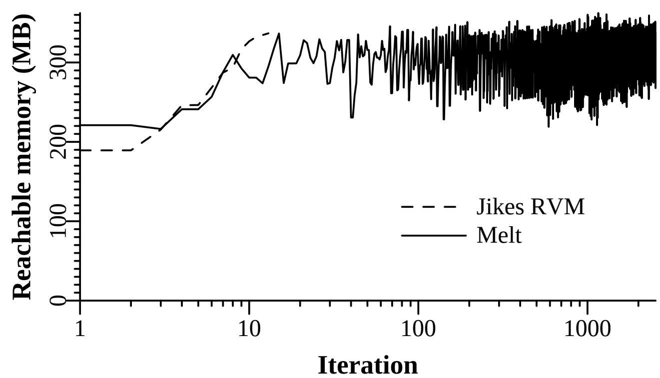

Figure 19. EclipseCP reachable memory over time, with and without Melt (logarithmic x-axis).

in the stale space both grow linearly over iterations and have large magnitudes. This result motivates avoiding a solution that uses time or space proportional to stale objects or references from stale to in-use objects. Figure 17 shows that Melt holds in-use objects relatively constant over iterations. The number of scions grows linearly over time, although it stays small in magnitude: roughly one scion per iteration. This growth occurs because a very small part of the leak is *live*. Each iteration leaks a large data structure, and the root object of this structure remains live, and this object uses an extra scion.

The graph shows that the number of objects activated increases linearly but its magnitude is still small compared with objects in the stale space, i.e., just a few stale objects are activated. Each activated object needs a scion, and many more objects are activated than there are scions, which shows that the application activates the same objects over and over again. Future work could consider a different policy for objects that have been activated. The fact that scions stay relatively small while stale-to-in-use references grow significantly, motivates Melt's use of stub-scion pairs to maintain references from stale to in-use objects.

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

EclipseCP. We reproduce Eclipse bug #155889, which reports a growing leak when the user repeatedly cuts text, saves, pastes the same text, and saves again. An Eclipse plugin we wrote exercises the GUI to perform this cut-paste behavior. Figure 18 shows the run time of each iteration of a cut-save-paste-save of a large block of text, using a logarithmic x-axis since unmodified Jikes runs for a short time before running out of memory. We do not present data for Sun JVM since we could not reproduce the leak with it. The figure shows that Melt adds some overhead to EclipseCP, but it is able to execute with fairly constant long-term performance for nearly 200 times as many iterations as without Melt. We terminate Melt after 24 hours, at which point it has used 39 GB of disk space. We note that the performance fluctuations are due to the application, not Melt, since they occur with unmodified Jikes RVM. Figure 19 shows memory usage over time, with and without Melt. Melt holds memory fairly steady in the long term. The short-term fluctuations are due to Melt moving objects gradually to the stale space each time the heap reaches 80%.

JbbMod. Since SPECjbb2000 (see below) has significant *live* heap growth, Tang et al. modified it by injecting a leak of *dead* (permanently stale) objects [57]. This version, which we call JbbMod, is a very slow-growing leak. Melt runs almost 21 hours (almost 20 times more iterations than without

Melt) before crashing with an apparent heap corruption error, likely due to a bug in Melt. During this time, it keeps performance and memory usage fairly constant. We thus believe that all heap growth is dead and, in lieu of crashing, Melt would run the program as long as disk space allowed.

ListLeak. The first microbenchmark leak is from a post on the Sun Developer Network [56]. It is a very simple and fast-growing leak:

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

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

SwapLeak. This leak also comes from a message posted on the Sun Developer Network [55]. The message asks for help understanding why an attached program runs out of memory. The program first initializes an array of 1,000,000 SObjects, which each contain an inner class Rep. The program then swaps out each SObject's Rep object with a new SObject's Rep object. Intuitively it seems that the second operation should have no net effect on reachable memory. However, as explained by a response to the message, the VM keeps a reference from an inner class object back to its containing object, which causes the swapped-out Rep object and the new SObject to remain reachable. The fix is to make the inner class static, but Melt provides the illusion of a fix without needing to understand or apply the fix.

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

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

Melt tolerates this leak but periodically suffers a huge pause when the hash table grows and re-hashes its elements, which activates all statement objects. Figures 20 and 21 show the performance (logarithmic y-axis) and memory usage of MySQL over time, with and without Melt. Unmodified Jikes RVM and Sun JVM quickly run out of memory, but Melt keeps the program running for 74 times as many iterations as Jikes RVM. When the hash table of statements grows and re-hashes its elements, e.g., at iterations 300 and 600, pause times rise to 30 minutes. Our implementation of Melt is not optimized for activation performance since it does not consider locality when moving objects to disk or activating objects, so a future implementation could potentially do better. Alternatively, an implementation could attempt to recognize that statement objects are live rather than dead.

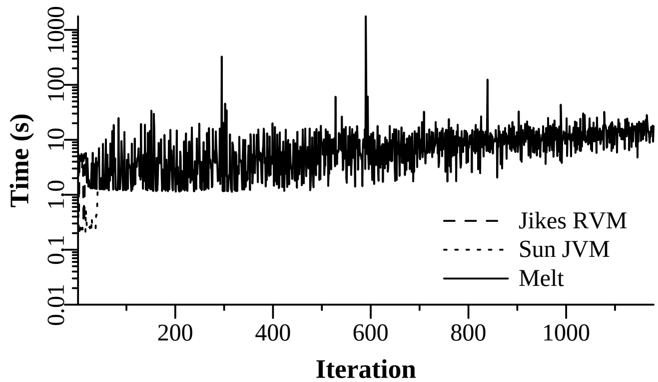

Figure 20. MySQL performance over time, with and without Melt. The y-axis is logarithmic because some pause times are quite high.

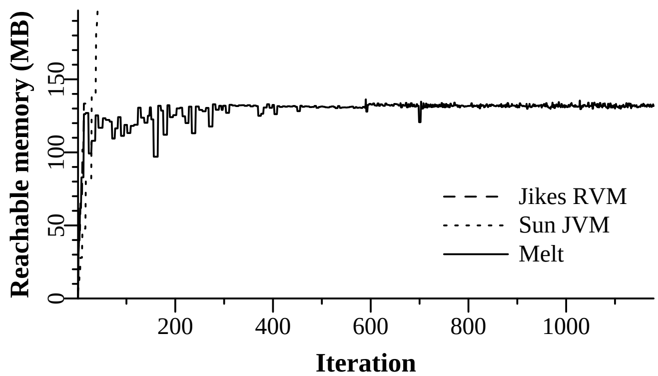

Figure 21. MySQL reachable memory over time, with and without Melt.

Melt terminates with an unrelated corruption error, mostly likely caused by a bug in Melt or perhaps Jikes RVM, after 4 hours and 20 minutes. Melt could tolerate the leak longer if the VM did not crash, albeit with periodic pauses to activate all statement objects.

Delaunay. Next we present a leak in Delaunay, an application that performs a Delaunay triangulation, which generates a triangle mesh, for a set of points, that meets a set of constraints [22]. We obtained the program from colleagues who added a *history directed acyclic graph* (DAG) to reduce algorithmic complexity of the triangulation, but the

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

Table 3. Delaunay run times, stale memory, and activated memory for various input sizes. *OOM* means out of memory.

change inadvertently caused graph components no longer in the graph to remain reachable.

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

Table 3 shows run times for all configurations and how much memory is transferred to and from disk using a maximum heap size of 256 MB and a variety of input sizes with a focus on 21,000-22,000 iterations, when the program exhausts memory. We set the threshold for moving objects to the stale space at 95% to avoid moving objects to the stale space too aggressively. For input sizes $\leq$21,000 iterations, all VMs perform similarly since the program has enough memory. Starting with 22,000 iterations, Melt tolerates the leak while the unmodified VM runs out of memory. The performance of Melt with an in-memory stale space scales well with input size. The on-disk stale space's performance does not scale well because Melt activates many objects from disk, which becomes expensive when the working set of accesses exceeds disk buffering. At some point, Melt is going beyond tolerating the leak, i.e., the heap would not be large enough even if the leak were fixed, as indicated by the increasing amount of activated memory.

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

SPECjbb2000. SPECjbb2000 simulates an order processing system and is intended for evaluating server-side Java performance [54]. It contains a known, growing memory leak that manifests when it runs for a long time without changing warehouses. It leaks because it adds orders to an order list that should have no net growth and does not correctly remove some of them.

Although SPECjbb2000 experiences unbounded heap growth over time, it uses almost all the objects. The program periodically accesses all orders in the order list. It seems unlikely that any system will be able to differentiate useful from useless memory accesses. We note that prior work on staleness-based leak detection diagnoses this leak because a small part of each order's data structure is stale [9]. Melt executes SPECjbb2000 about twice as long as without Melt (1166 vs. 540 iterations) since it finds some stale memory to move to disk. However, performance suffers beginning at about 650 iterations because Melt starts moving many objects that are not permanently stale to disk in order to avoid running out of memory, resulting in significant activation overhead.

DualLeak. This leak comes from an example in an IBM developerWorks column [23]. We call it DualLeak since its 55 sources lines contain two different leaks. The program executes in iterations and exercises both leaks during each iteration. The first leak is slow-growing and occurs because of an off-by-one error that leads to an Integer object not being removed from a Vector on each iteration. The other leak grows more quickly by adding multiple String objects to a HashSet on each iteration.

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

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

Mckoi. We reproduce a memory leak reported on a message board for Mckoi SQL Database, a database management system written in Java [37]. The leak occurs if a program repeatedly opens a database connection, uses the connection, and closes the connection. Mckoi does not properly dispose of the connection thread, leading to a growing number of unused threads. These threads leak memory; most of the leaked bytes are for each thread's stack.

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

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

5. Related Work

Although there is a lot of prior work on detecting leaks, only a few researchers have tried to tolerate leaks. Our leak tolerance approach improves over previous approaches by offering a comprehensive and safe solution that identifies stale objects and handles small leaking objects with time and space proportional to in-use memory.

5.1 Detecting Leaks

Static analysis for C and C++ detects leaks before the program executes but can produce false positives [14, 25]. This prior work focuses on identifying unfreed, unreachable objects whereas our work addresses reachable but dead objects. Dynamic tools for C and C++ track allocations, heap updates, and frees to report unfreed objects [24, 35, 40] or track object accesses to report stale objects [15, 47]. Online leak detectors for managed languages identify heap growth or stale objects to find potential leaks [9, 34, 38, 44, 49, 52].

5.2 Dealing with Memory Pressure

Language features and automatic approaches can help applications experiencing memory pressure.

To help programmers avoid leaks and manage large heaps, the Java language definition provides *weak* and *soft* references. The collector always reclaims weakly-referenced objects, and it reclaims softly-referenced objects if the application experiences memory pressure [19, 20]. Inserting soft and weak references adds to the development burden, and programmers may still forget to eliminate the last strong (not weak or soft) reference.

Static liveness detection of GC roots can reduce the *drag* between when objects die and when they are collected [27] but cannot deal with other dead, but reachable, objects.

Many VMs dynamically size the heap based on application behavior. For example, some approaches adaptively trigger GC or resize the heap in order to improve GC performance and program locality [13, 59, 60, 61]. These approaches do not directly address memory leaks.

When the application's heap size exceeds its working set size, *bookmarking collection* reduces collection overhead [26]. It cooperates with the operating system to *bookmark* swapped-out pages by marking in-memory objects they reference as live. The garbage collector then never visits its bookmarked pages. Bookmarking can compact the heap but cannot move objects referenced by bookmarked pages. It tracks staleness on page granularity. Melt instead uses object granularity, grouping and isolating leaking objects.

General error tolerance approaches, such as *failure-oblivious computing* [50], *DieHard* [5], and *Rx* [48] deal with memory corruption and nondeterministic errors to improve reliability, but they do not handle memory leaks.

5.3 Tolerating Leaks

Several recent publications address the problem of tolerating leaks [11, 21, 41, 42, 57]. Compared to Melt, they offer less coverage, do not scale, or are unsafe.

Leaks in native languages. *Cyclic memory allocation* tolerates leaks in C and C++ by limiting allocation sites to m live objects at a time [41]. Profiling runs determine m for each allocation site, and subsequent executions allocate into m-sized circular buffers. Cyclic memory allocation is *unsafe* since it may overwrite live memory, although failure-oblivious computing [50] mitigates the effects in some cases. In contrast, our approach places no requirements on allocation sites and is always safe.

Plug safely tolerates leaks in C and C++ with an allocator that segregates objects by age and allocation site, increasing the likelihood that leaked and in-use objects will reside on distinct pages [42]. Plug deals with later fragmentation via *virtual compaction*, which maps two or more virtual pages to the same physical page if the allocated slots on the pages do not overlap. Plug's approach helps native languages since objects cannot move, but collectors in managed languages can reorganize objects. In addition, segregating leaked and in-use objects is insufficient for managed languages since tracing collectors by default access the whole heap.

Leaks in managed languages. *Panacea* supports moving stale objects to disk [11, 21]. The approach requires annotations for objects that can be moved to disk, and these objects must be serializable to get put on disk. Panacea does not scale for small, stale objects—which we find are frequent leak culprits—because it uses proxy objects for swapped-out objects. An advantage of Panacea is that it is implemented at the library level and needs no VM modifications.

LeakSurvivor [57] is the closest related work and was developed concurrently with Melt [10]. Both approaches free up virtual and physical memory by transferring highly stale objects to disk, and both preserve safety by returning accessed disk objects to memory. Unlike Melt, LeakSurvivor cannot guarantee space and time proportional to in-use memory because references from stale to in-use objects continue to use space even if the in-use objects become stale. In particular, entries in LeakSurvivor's *Swap-Out Table* (SOT) (similar to Melt's scion table) cannot be eliminated if the target object moves to disk, since incoming pointers from disk are unknown. In contrast, Melt uses two levels of indirection, stub-scion pairs, to eliminate scions referencing objects later moved to the stale space (Section 2.2). For the three leaks evaluated in LeakSurvivor, the SOT grows only slightly, but it is unclear if they grow proportionally to the leak since the experiments are terminated after two hours, before the leaks would have filled the disk. Melt adds less overhead than LeakSurvivor to identify stale objects (6% vs. 21%) since LeakSurvivor accesses an object's header on each read, while Melt uses referenced-based conditional read barriers to avoid accessing object headers in the common case.

5.4 Orthogonal Persistence and Distributed GC

Leak tolerance uses mechanisms that have been used in orthogonal persistence, distributed garbage collection, and other areas. Orthogonal persistence uses object faulting, pointer swizzling, and read barriers to support transparent storage of objects on disk [3, 28, 29, 36, 62]. Pointer swizzling can also be used to support huge address spaces [39, 58]. Our implementation uses swizzling to support a 64-bit disk space on a 32-bit platform. Read barriers are widely used in concurrent garbage collectors [4, 8, 17, 30, 45, 63]. Distributed collectors use stub-scion pairs for references between machines [46]. We use stub-scion pairs to support references from stale to in-use objects. Although leak tolerance borrows existing mechanisms, previous work does not combine these mechanisms in the same way as leak tolerance, i.e., to identify, isolate, and activate stale memory.

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. Melt requires only time and space proportional to in-use memory, rather than leaked memory, and preserves safety by activating stale objects on disk that the program later references. Our Melt implementation adds low enough overhead for deployed use. For growing leaks in real programs, Melt substantially delays crashes due to out-of-memory errors. With plenty of disk space, Melt has the potential to improve the user experience. It buys developers more time to fix leaks and keeps users happy by providing the illusion there is no leak. These attributes make Melt a compelling feature for future production VMs.

Acknowledgments

We thank Jason Davis for the MySQL leak, Patrick Carribault for the Delaunay leak, Maria Jump for the SPECjbb2000 leak, and Yan Tang for the modified SPECjbb2000 leak. Thanks to Eddie Aftandilian, Emery Berger, Steve Blackburn, Curtis Dunham, Daniel Frampton, Robin Garner, David Grove, Samuel Guyer, Xianglong Huang, Maria Jump, Milind Kulkarni, Erez Petrank, Chris Pickett, Dimitrios Prountzos, and Jennifer Sartor for helpful discussions. We thank Emery Berger, Rudy Depena, Tom Horn, Nicholas Nethercote, and the anonymous reviewers for valuable feedback on the paper text.

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

[4] 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.

[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. Wiedermann. 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] M. D. Bond and K. S. McKinley. Tolerating Memory Leaks. Technical Report TR-07-64, University of Texas at Austin, December 2007.

[11] D. Breitgand, M. Goldstein, E. Henis, O. Shehory, and Y. Weinsberg. PANACEA–Towards a Self-Healing Development Framework. In *Integrated Network Management*, pages 169–178, 2007.

[12] 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.

[13] 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.

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

[15] 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.

[16] DaCapo Benchmark Regression Tests. http://jikesrvm.anu.-edu.au/˜dacapo/.

[17] 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):966–975, Nov. 1978.

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

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

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

[21] M. Goldstein, O. Shehory, and Y. Weinsberg. Can Self-Healing Software Cope With Loitering? In *International Workshop on Software Quality Assurance*, pages 1–8, 2007.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] M. Hirzel, A. Diwan, and J. Henkel. On the Usefulness of Type and Liveness Accuracy for Garbage Collection and Leak Detection. *ACM Transactions on Programming Languages and Systems*, 24(6):593–624, 2002.

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

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

[30] A. L. Hosking, N. Nystrom, Q. I. Cutts, and K. Brahnmath. Optimizing the Read and Write Barriers for Orthogonal Persistence. In *International Workshop on Persistent Object Systems*, pages 149–159, 1999.

[31] 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.

[32] Jikes RVM. http://www.jikesrvm.org.

[33] Jikes RVM Research Archive. http://www.jikesrvm.org/-Research+Archive.

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

[35] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise Detection of Memory Leaks. In *International Workshop on Dynamic Analysis*, pages 25–31, 2004.

[36] A. Marquez, S. M. Blackburn, G. Mercer, and J. Zigman. Implementing Orthogonally Persistent Java. In *International Workshop on Persistent Object Systems*, pages 247–261, 2000.

[37] 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.

[38] 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.

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

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

[41] 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.

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

[43] 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.

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

[45] 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.

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

[47] 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.

[48] 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 Systems Principles*, pages 235–248, 2005.

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

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

[51] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: High-Performance Garbage Collection for Memory-Constrained Environments. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 81–98, 2004.

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

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

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

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

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

[57] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages. In *USENIX Annual Technical Conference*, pages 307–320, 2008.

[58] 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.

[59] 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.

[60] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual Memory Support for Garbage-Collected Applications. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 103–116, 2006.

[61] 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.

[62] J. N. Zigman, S. Blackburn, and J. E. B. Moss. TMOS: A Transactional Garbage Collector. In *International Workshop on Persistent Object Systems*, pages 138–156, 2001.

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