Leak Pruning

To appear in ASPLOS 2009. Preliminary draft version (Nov 2008); final version available Jan 2009.

Michael D. Bond Kathryn S. McKinley

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

Abstract

Managed languages improve programmer productivity with type safety and garbage collection, which eliminate memory errors such as dangling pointers, double frees, and buffer overflows. However, programs may still *leak* memory if programmers forget to eliminate the last reference to an object that will not be used again. Leaks slow programs by increasing collector workload and frequency. Growing leaks crash programs. Instead of crashing, *leak pruning* extends program availability by predicting and reclaiming leaked objects at run time. Whereas garbage collection over-approximates live objects using reachability, leak pruning predicts dead objects and reclaims them based on how *stale* they are and the size of stale data structures. Leak pruning *preserves semantics* because it waits for heap exhaustion before reclaiming objects and then poisons references to objects it reclaims. If the program later tries to access these objects, the virtual machine (VM) throws an internal error. We implement leak pruning in a Java VM, show its overhead is low, and evaluate it on 10 leaking programs. Leak pruning does not help two programs, executes four substantial programs 1.6-35X longer, and executes four programs, including two leaks in Eclipse, for at least 24 hours. In the worst case, leak pruning defers fatal errors. In the best case, programs with unbounded memory requirements execute indefinitely and correctly in bounded memory with consistent throughput.

1. Introduction

Managed languages such as Java, C#, Python, and Ruby provide garbage collection and type safety, which eliminate (1) memory corruption errors such as dangling pointers, double frees, and buffer overflows and (2) memory leaks due to unreachable objects. The increasing use of managed languages is due in part to these features. Unfortunately, programs may still leak objects that are reachable, but will not be used again, because garbage collection uses *reachability* to over-approximate *liveness*. A reachable object is not live if the program never uses it again. Computing reachability is relatively straightforward; collectors perform a transitive closure over the object graph from program *roots* (globals, stacks, and registers). Liveness is much harder to determine and is in general undecidable.

Memory leaks hurt performance by consuming unnecessary memory resources, and they increase garbage collection frequency and workload. Leaks occur frequently in managed languages and a number of tools help programmers diagnose them [8, 26, 30, 35, 39, 42]. Leaks are hard to reproduce, find, and fix because they have no immediate symptoms [20]. For example, *when* a leaking Java program exhausts memory depends on the heap size, choice of garbage collector, and nondeterministic factors not directly related to the leak. Despite extensive in-house testing, leaks exist in production software because they are behavior and environment sensitive. Furthermore, attackers can exploit these behaviors to launch denial-of-service attacks.

This paper introduces *leak pruning*, which *preserves semantics*, uses *bounded resources*, and runs leaky programs longer than before or, in some cases, indefinitely. Leak pruning defers out-of-memory errors by predicting which objects are dead and reclaiming them when the program is about to run out of memory. As long as the program does not attempt to access reclaimed objects, it may run indefinitely. If the program attempts to access a reference to reclaimed memory, a leak pruning-enabled VM intercepts the access and throws an error. This behavior preserves semantics since the program already ran out of memory. In the worst case, leak pruning only defers out-of-memory errors. In the best case, it enables leaky programs with unbounded memory requirements to run indefinitely in bounded memory.

The key to leak pruning is accurately predicting which reachable objects are dead. Our dynamic prediction algorithm uses past access behavior by differentiating *in-use* objects that the program referenced recently and *stale* objects that have not been referenced in a while. It piggybacks on the garbage collector. While the collector traces the reachable objects in the heap, it also identifies stale *data structures*, i.e., stale subgraphs. It records the source and target classes of the first reference and the number of stale bytes in the stale

data structure. When the VM runs out of memory, leak pruning *poisons* references to instances of the data structure type consuming the most bytes. Poisoning invalidates and specially marks references. The collector then reclaims objects that were only unreachable from these references. If the program subsequently accesses a poisoned reference, the VM throws an error.

We implement leak pruning in a high-performance Java VM and show that it adds on average 6% to execution time due to its software read barrier (instrumentation at every read [7]). This overhead would drop to zero with hardware read barriers, such as those provided by Azul [13]. Although our implementation is for Java, the general approach is applicable to any language that supports garbage collection. We evaluate how well leak pruning tolerates leaks on 10 leaking programs, including 2 in Eclipse. For two leaks, leak pruning provides no help. It runs two programs 1.6-4.7X longer and two leaks 21-35X longer. The remaining four leaky programs execute for at least 24 hours (then we terminate them). When leak pruning cannot defer an out-of-memory error indefinitely, in all cases but one it is due to growing *live* memory, i.e., the program's working set is growing because the program continues to access leaked memory. Other leak tolerance approaches that preserve semantics cannot tolerate live leaks either.

One objection to error tolerance is that it may encourage poor programming practices. Since modern software is never bug free, error tolerance in general should be viewed as a temporary measure that gives users a better experience, buys developers time to fix bugs, and provides protection against some attacks. Leak pruning may not be appropriate for all programs, e.g., programs that catch out-of-memory errors to abort speculative computation, and should be a configuration parameter at deployment time. Most prior leak tolerance approaches use unbounded amounts of disk space and therefore will eventually exhaust disk space and crash [9, 10, 18, 47]. Leak pruning can be applied after these approaches exhaust their disk space budget or in embedded systems without disks.

The contributions of this paper are (1) leak pruning, a novel semantics-preserving approach for reclaiming memory instead of running out of memory, (2) an algorithm for accurately identifying likely dead objects, and (3) an evaluation of leak pruning's effectiveness on 10 leaks: 5 benchmarks and 5 real applications. Leak pruning's preservation of semantics and low overhead make it a compelling configuration choice for many deployed systems.

2. Related Work

Prior work tolerates memory corruption and concurrency bugs using redundancy, randomness, checkpointing, padding, and ignoring errors, but these approaches do not help memory leaks [4, 38, 40]. One industrial response to leaks is restarting the application, but this mechanism reduces availability and loses application state that may not be recoverable. The closest related work to leak pruning is *cyclic memory allocation*, which limits the amount of live memory produced by any allocation site [32]. Also related is leak tolerance using disk space [9, 10, 18, 47]. *Leak detection* is synergistic with leak tolerance. Detection gives developers more information on how to fix leaks. Tolerance yields a better user experience and protection against some attacks.

Detecting leaks. Static leak detectors for C and C++ identify objects that the programmer forgot to free and are unreachable [11, 21]. Dynamic leak detectors for C and C++ find these objects at run time by tracking allocations, frees, and pointer mutations [20, 28, 31] or by tracking staleness [12, 37]. Managed languages reclaim unreachable objects, but reachable objects that the program will never access again can leak. Leak detectors for managed languages report dynamic heap growth [26, 30, 35, 39, 42] and stale objects [8]. Leak pruning uses object staleness to predict liveness.

Bounding memory usage. Leak pruning is most similar to cyclic memory allocation, which seeks to bound memory usage by controlling the number of live objects produced by an allocation site to m [32]. Off-line profiling determines m, and a modified allocator uses a cyclic buffer. It assumes each allocation site produces a bounded footprint of live objects, but some leaks do not have this property. In contrast, leak pruning predicts liveness based on object and data structure staleness. Cyclic memory allocation may change program semantics since the program is silently corrupted if it uses more than m objects, although *failure-oblivious computing* [40] often mitigates the effects.

Tolerating leaks. *Plug* segregates objects at allocation time and re-maps virtual to physical pages to save physical memory in C and C++ programs, but it does not address challenges presented by garbage collection [33].

LeakSurvivor, *Panacea*, and *Melt* tolerate leaks by transferring potentially leaked objects to disk [9, 10, 18, 47]. They reclaim virtual and physical memory and modify the collector to avoid accessing objects moved to disk. Leak pruning borrows Melt's low-overhead, reference-based read barriers. Although not designed to tolerate leaks, *bookmarking collection* tolerates leaks somewhat by saving physical, not virtual, memory and tracking staleness on page granularity [22].

These approaches preserve semantics since they retrieve objects from disk if the program accesses them. Since they retrieve objects from disk, the prediction mechanisms do not have to be perfect, just usually right. If the systems are too inaccurate, performance will suffer. All will eventually exhaust disk space and crash. Leak pruning, on the other hand, requires perfect prediction to defer a leak successfully. Consequently, it uses a more sophisticated algorithm for predicting dead objects (Section 7 compares leak pruning's

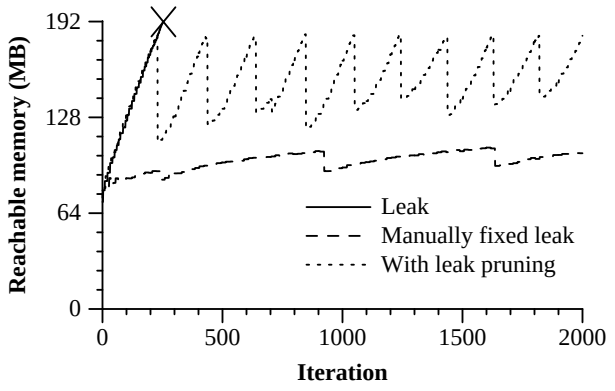

Figure 1. Reachable heap memory for the EclipseDiff leak: An unmodified VM running the leak, a manually fixed version, and the leak running with leak pruning.

prediction algorithm to the algorithm used by prior work). Leak pruning is potentially less tolerant of errors because it must throw an error if it makes a mistake. However, leak pruning uses bounded resources, making it suitable when disk space runs out or no disk is available (e.g., embedded systems).

3. Leak Pruning

This section describes the high-level approach and semantics of leak pruning. Section 4 presents the details of our algorithm and implementation.

3.1 Motivation and Overview

Garbage collection overview. Garbage collection (GC) uses *reachability* as an over-approximation of *liveness*—an object is live if the program will use it again. The VM invokes the collector each time the program fills the heap. A *tracing* collector then performs a transitive closure over the heap starting from the *roots*, which include stack pointers, global variables, and any references in registers.[1] The collector retains objects that are transitively reachable and reclaims all the memory used by unreachable objects. The next collection occurs after the sum of this reachable memory plus new allocation exceeds the available heap memory.

Reachability versus liveness. Reachability often approximates liveness well. However, developers may neglect to remove the last reference to an object or data structure that the program will not use again. Dead-but-reachable objects are *leaks*, and a growing leak has *unbounded* memory requirements. Leaks (1) slow the program down as the heap fills by increasing the frequency and workload of garbage collection and (2) eventually cause the program to throw an out-of-memory error by exhausting memory resources.

Figure 1 shows the memory consumption over time measured in *iterations* (fixed amounts of program work) for a growing leak in Eclipse called EclipseDiff (Section 6 discusses this leak in detail). The graph shows reachable memory at the end of each full-heap collection. The solid line shows that the leak causes reachable memory to grow without bound until it overflows the heap. At 200 MB for this experiment, the VM throws an out-of-memory error (the plotted line reaches only 192 MB since the program did not complete another iteration before exhausting memory).

A larger maximum heap size can help leaky programs. Some systems, such as embedded systems, have hard upper bounds on maximum memory. In systems with virtual memory and swap space, the upper bound on the maximum heap size is effectively physical memory; exceeding physical memory leads to thrashing during GC since the collector's working set is the entire heap [22, 48].

The dashed line shows reachable memory if we modify the Eclipse source to fix the leak (as described in the bug report, which includes a fix). Reachable memory stays fairly constant over time, and Eclipse does not run out of memory. The dotted line shows reachable memory with leak pruning. When the program is about to run out of memory, leak pruning reclaims objects that it predicts are dead. It cannot reclaim all dead objects promptly because objects need time to become stale. Section 6 shows that leak pruning keeps EclipseDiff from running out of memory for over 50,000 iterations (24 hours).

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

3.2 Triggering Leak Pruning

Figure 2 shows a high-level state diagram for leak pruning. Leak pruning's state is based on how close the program is to running out of memory. Leak pruning performs most of its work during full-heap garbage collections, and it changes state after each full-heap GC. State changes depend on how full the heap is at the end of GC.

Initially, leak pruning is INACTIVE and does not observe program behavior. This state has two purposes. First, it avoids the overhead of leak pruning's analysis when the program is not running out of memory. Second, it collects potentially better information by focusing on program behavior that appears to be leaking. Leak pruning remains INACTIVE until reachable memory exceeds "expected memory use," a user-configurable threshold. By default our implementation

[1] Our discussion and implementation use a tracing collector because pure reference-counting collectors are incomplete—they cannot reclaim cycles.

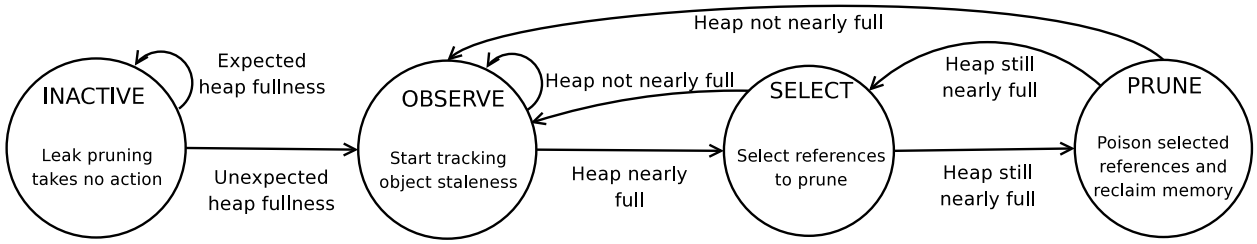

Figure 2. State diagram for leak pruning.

sets this threshold to 50% since users typically execute programs in heaps at least twice as large as maximum reachable memory. Leak pruning is not very sensitive to the exact value of this threshold. If set too low, leak pruning may incur some overhead when the program is not leaking; if set too high, it will have less time to observe program behavior before selecting memory to reclaim.

When memory usage crosses this threshold, leak pruning enters the OBSERVE state, in which it analyzes program reference patterns to choose pruning candidates (described in detail in Section 4). Once leak pruning enters the OBSERVE state, it never returns to the INACTIVE state because it considers the application to be in a permanent unexpected state.

Leak pruning moves from OBSERVE to SELECT when the program has nearly run out of memory, which is user-configurable and 90% by default. In SELECT, leak pruning chooses references to prune, based on information collected during the OBSERVE state (described in detail in Section 4).

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

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

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

Figure 3 shows an example heap when leak pruning enters the PRUNE state. Each circle is a heap object. Each object instance has a name based on its *class*, e.g., b1, b2, b3, and b4 are instances of class B. The selection algorithm uses class to select references to prune (Section 4). The figure shows that objects a1 and e1 are directly reachable from the program roots (registers, stacks, and statics), and other objects are transitively reachable. Suppose leak pruning selects three references to prune, labeled **sel** in the figure: b1 → c1, b3 → c3, and b4 → c4.

3.3 Reclaiming Reachable Memory

During a full-heap collection in the PRUNE state, the collector repeats its analysis, but this time poisons selected references and reclaims all objects reachable only from these references as shown in Figure 4. The collector reclaims objects reachable *only* from pruned references since it does not trace pruned references. The subtree rooted at c4 is not reclaimed because it is transitively reachable from the roots via object e1.

Leak pruning poisons a reference by setting its second-lowest-order bit (Section 4.4). Setting the reference to null is insufficient since that could change program semantics. If the program accesses a poisoned reference, the VM intercepts the access and throws an internal error with an attached out-of-memory error. This behavior preserves semantics since the program previously ran out of memory when it entered the PRUNE state for the first time.

3.4 Exception and Collection Semantics

The Java VM specification says that an OutOfMemoryError can be thrown only at program points responsible for allocating resources, e.g., new expressions or expressions that may trigger to class initialization [27]. Program accesses to pruned memory are at reference loads, which are not memory-allocating expressions. The Java specification however permits InternalError to be thrown asynchronously at

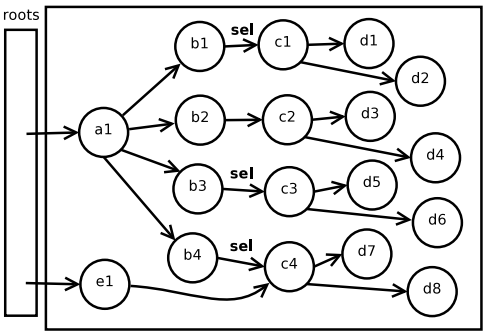

Figure 3. Example heap after the SELECT state. References selected for pruning are marked with **sel**.

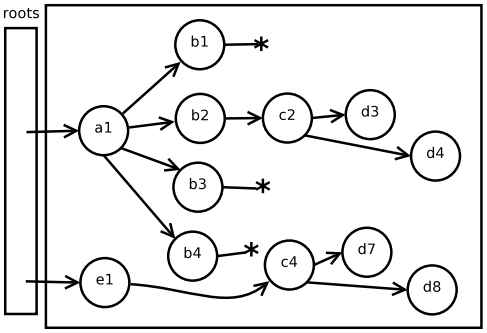

Figure 4. Example heap at the end of GC in PRUNE state. Poisoned references end in an asterisk (*).

any program point. Our implementation thus throws an InternalError if the program accesses a pruned reference.

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

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

3.5 Helping Programmers and Users

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

4. Algorithm and Implementation

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

4.1 Predicting Dead Objects

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

Since leaks add space and time overhead, the prediction algorithm should not make matters even worse. In particular, we should not add space proportional to the objects in the heap. Our algorithm steals three available bits in object headers and two unused lowest-order bits in object-to-object references.

4.2 The OBSERVE State

Tracking staleness. In the OBSERVE state, leak pruning tracks each object's *staleness*, i.e., how long since the program last used it. Our implementation maintains staleness using a three-bit *logarithmic stale counter* in each object's header (an idea borrowed from prior leak detection work [8]). A value k in an object's stale counter means the program last used the object approximately 2^k collections ago. We maintain each stale counter's value by (1) incre-

menting object counters in each collection and (2) inserting instrumentation to clear an object's counter when it is used.

The collector keeps an exact count of the number of full-heap garbage collections. Every full-heap collection i increments an object's stale counter if and only if i evenly divides 2^k, where k is the current value of the counter. In addition, the collector sets the lowest bit of every object-to-object *reference*, which is available since objects are word-aligned. Setting this bit allows for a quick test in application instrumentation of whether the target object's stale counter has been reset since the last collection [9].

The baseline and optimizing compilers (see Section 5) insert *read barriers* (instrumentation at every reference load, e.g., `b = a.f`) [7] that reset the referenced object's (`b`) stale counter. The instrumentation is efficient because it first checks if the lowest bit of the reference (`a.f`) is cleared; if so, it knows a read barrier previously cleared `b`'s stale counter. The following pseudocode shows the read barrier:

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

If a reference's lowest bit is set, the barrier clears this bit and also clears the referenced object's stale counter. Thus, the barrier condition is true at most once for each reference after each full-heap collection, i.e., in the common case, the barrier's body does not execute, so we put it *out-of-line* in a separate method.

The barrier stores the updated reference atomically to avoid overwriting another thread's write. The notation `[iff a.f == t]` indicates the store occurs if and only if the reference slot has not been modified by another thread. If the store fails, the barrier simply continues, which is a valid serialization since another thread will have written a valid value with its lowest bit cleared to `a.f`, and the current thread can safely use the reference `b`. The barrier also clears `b.staleCounter` atomically to avoid losing updates to other bits in the object header (used for locking and hashing in many VMs). Since the barrier condition is usually false, the atomic stores add unnoticeable overhead.

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

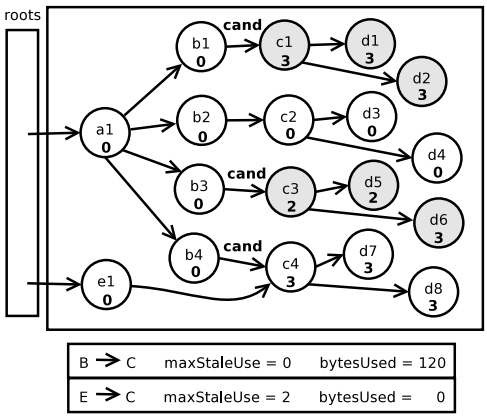

Figure 5. Example heap during the SELECT state.

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

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

4.3 The SELECT State

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

1. The *in-use transitive closure* starts with the roots (registers, stacks, statics) and marks live objects, except for when it encounters a stale reference whose target object has a stale counter at least two greater than its edge table entry's maxStaleUse value. We conservatively use two greater, instead of one, since the stale counters only approximate the logarithm of staleness. These references are *candidates* for pruning. Instead of processing them, we put them on a *candidate queue*.
2. The *stale transitive closure* starts with references in the candidate queue; these references' target objects are the roots of stale data structures. It computes the bytes reachable from each root, i.e., the size of the stale data structure. The stale closure adds this value to bytesUsed for the edge entry for the candidate reference.

At the end of this process, leak pruning iterates over each entry in the edge table, finding the entry with the greatest bytesUsed value, and it resets all bytesUsed values. This edge type is used for pruning in the PRUNE state.

Example. Figure 5 shows the heap and an edge table for Figures 3 and 4 during SELECT. Each object is annotated with the value of its stale counter. The in-use closure adds the references marked **cand** to the candidate queue, but it does not add b2 → c2 since c2's stale counter is less than 2. It also does not add e1 → c4. Its stale counter must be at least 4, i.e., 2 more than the maxStaleUse of 2 for E → C in this example. The stale closure processes the objects reachable only from candidate references, which are shaded gray. Objects c4, d7, and d8 are processed by the *in-use closure* since they are reachable from non-candidate reference e1 → c4. If we suppose each object is 20 bytes, then bytesUsed for B → C is 120 bytes. This edge entry is selected for pruning since it has the greatest value of bytesUsed.

4.4 The PRUNE State

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

4.5 Intercepting Accesses to Pruned References

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

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

The read barrier body checks for a poisoned reference by examining the second-lowest bit. If the bit is set, the barrier throws an InternalError. To help users and developers, it attaches the original OutOfMemoryError that *would* have been thrown earlier.

4.6 Garbage Collection

The implementation's design is compatible with moving and non-moving collectors such as copying, compacting, and mark-sweep, all of which are in use by modern high-performance VMs. We use a parallel, stop-the-world, generational mark-sweep collector since it has high performance and also performs well in tight heaps [5]. This collector allocates objects into a *nursery*; when the nursery fills, the collector traces the live nursery objects and copies them into a mark-sweep *mature space*. When the mature space fills, the collector performs a full-heap collection that traverses and marks all reachable objects and then reclaims all unreachable objects.

4.7 Concurrency and Thread Safety

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

By default, the garbage collector is parallel [5]. It uses multiple collector threads to traverse all reachable objects. The implementation uses a shared pool from which threads obtain local work queues to minimize synchronization and balance load. Because many objects have multiple references to them, the collector prevents more than one thread from processing an object with fine-grained synchronization on the object. We piggyback on these mechanisms to implement the in-use and stale transitive closures. In the stale closure, a single thread processes all objects reachable from a candidate edge. The stale closure is parallel since multiple collector threads can process the closures of distinct candidates simultaneously.

5. Performance of Leak Pruning

This section measures the overhead leak pruning adds to observe and select references for pruning.

VM configurations. We implement leak pruning in Jikes RVM 2.9.2, a high-performance Java-in-Java virtual machine [1, 2, 24]. The DaCapo benchmark regression tests page [15] shows that Jikes RVM performs the same as Sun Hotspot 1.5, and between 15 to 20% worse than Sun 1.6, JRockit, and J9 1.9, all configured for high performance. Our performance measurements are therefore relative to an excellent baseline. We will make our implementation publicly available on the Jikes RVM Research Archive [25] in January 2009.

By default, Jikes RVM uses two compilers. It uses a baseline non-optimizing compiler to generate machine code when it first loads a class. Over time, it identifies hot methods and recompiles them with increasingly more aggressive compiler optimizations. This default execution behavior is called *adaptive* compilation. Because Jikes RVM uses timer-based sampling to identify hot methods, adaptive compilation is nondeterministic. To achieve determinism, we use *replay* compilation [23, 34, 41], which uses profile informa-

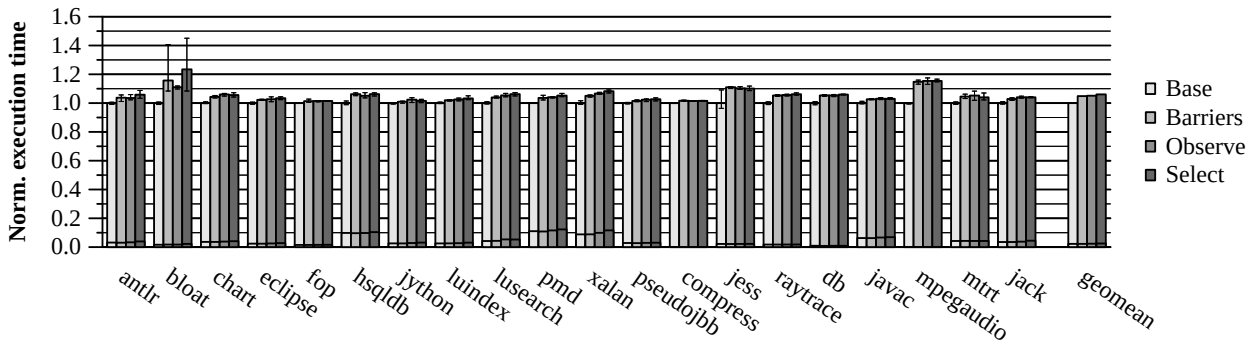

Figure 6. Application execution time overhead of leak pruning. Sub-bars are GC time.

tion from a previous adaptive execution to compile the same methods identically and deterministically. It thus avoids high variability due to sampling-driven compilation. For replay, we execute the benchmark *twice* in one JVM instance. The first run compiles all methods while running the program. The second executes only the application. This second execution is representative of steady-state application behavior. We use this configuration to measure overheads only. We use adaptive compilation when evaluating leaky programs.

Jikes RVM's Memory Management Toolkit (MMTk) [5] supports a variety of garbage collectors with most functionality residing in shared code. Our implementation resides almost exclusively in this shared code. To add support for leak pruning to another collector requires adding a method that specifies which space(s) contain objects that leak pruning should track. Because the VM is written in Java, VM objects reside in the heap. For simplicity and efficiency, we do not prune VM objects or objects directly pointed to by roots.

Benchmarks. We measure leak pruning's overhead on the DaCapo benchmarks version 2006-10-MR1, a fixed-workload version of SPECjbb2000 called pseudojbb, and SPECjvm98 [6, 43, 44].

Platform. 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 processor has a 64-byte L1 and L2 cache line size, a 16-KB 8-way set associative L1 data cache, a 12-Kμops L1 instruction trace cache, and a 1-MB unified 8-way set associative L2 on-chip cache.

Application overhead. Leak pruning adds overhead because it inserts read barriers into application code and tracks staleness and selects references to prune during garbage collection. Using replay compilation, Figure 6 shows application and collection times without compilation. Execution times are normalized to *Base*, which is unmodified Jikes RVM. Each bar is the mean of five trials; the error bars show the range across trials. Replay compilation keeps variability low in all cases except for bloat; we observe this variability in other versions, and it is unrelated to leak pruning. To control the memory size, we fix the heap at two times the

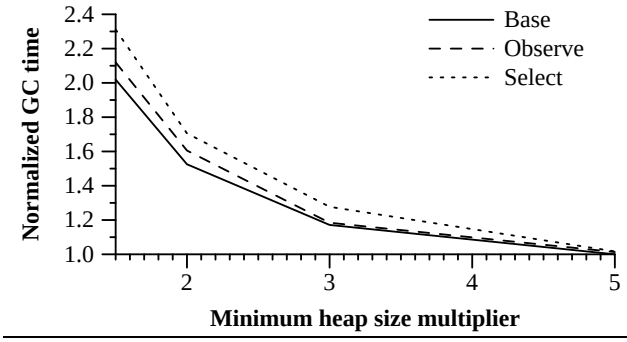

Figure 7. Normalized collection times for leak pruning across heap sizes (y-axis starts at 1.0).

minimum in which each benchmark can run. The sub-bars at the bottom are the fraction of time spent in collection.

The *Barriers* configuration includes only leak pruning's read barriers. Leak pruning does not need barriers in the INACTIVE state, so non-leaking programs would not experience this overhead. For simplicity, our implementation uses all-the-time read barriers, but a production version of leak pruning could trigger recompilation of all methods with read barriers when leak pruning entered the OBSERVE state. On average, barriers slow execution by 6%. This overhead is comparable to barrier overheads for concurrent, incremental, and real-time collectors [3, 16, 36]. With the increasing importance of concurrent software and the advent of transactional memory hardware, future general-purpose hardware is likely to provide read barriers with no overhead, and Azul hardware has them already [13].

Observe shows the overhead of the OBSERVE state. For these experiments, we force the OBSERVE state all the time. This configuration adds updating of each object's staleness header bits during collection and updating of maxStaleUse for edge types that become stale but are used later. Similarly, the *Select* configuration represents the overhead of always being in the SELECT state: performing the stale trace, updates to bytesUsed in the edge table, and selection of an edge type to prune. These configurations add no noticeable overhead since they mainly add collection overhead, which accounts for 2% of overall execution time on average.

Leak (LOC)	Leak pruning's effect	Reason
EclipseCP (2.4M)	Runs >100X longer	All reclaimed?
EclipseDiff (2.4M)	Runs >200X longer	Almost all reclaimed
ListLeak (9)	Runs indefinitely	All reclaimed
SwapLeak (33)	Runs indefinitely	All reclaimed
MySQL (75K)	Runs 35X longer	Most reclaimed
SPECjbb2000 (34K)	Runs 4.7X longer	Some reclaimed
JbbMod (34K)	Runs 21X longer	Most reclaimed
Mckoi (95K)	Runs 1.6X longer	Some reclaimed
DualLeak (55)	No help	None reclaimed
Delaunay (1.9K)	No help	Short-running

Table 1. Ten leaks and leak pruning's effect on them.

Garbage collection overhead. Figure 7 plots the geometric mean of normalized GC time over all the benchmarks as a function of heap sizes ranging from 1.5 to 5 times the minimum heap size in which each benchmark executes. The smaller the heap size, the more often the program exhausts memory and invokes the collector. Observe adds up to 5% overhead to mark the lowest bit of references and update staleness. Selecting references to prune every collections adds 9% more, for a total of 14%.

Compilation overhead. Inserting read barriers adds compiler overhead by increasing the code size and thus work for downstream optimizations. To mitigate this overhead, the compilers insert only the conditional test and a method call for the barrier's body. We measure compilation time using the first run of replay methodology. Read barriers increase code size by 10% on average and 15% at most (for javac). Inserting read barriers adds 17% to compilation time on average and at most 34% (for raytrace). In practice this overhead is negligible because compilation accounts for just 4% of overall execution time, and long-running programs are likely to spend an even smaller fraction of total time compiling.

6. Effectiveness of Leak Pruning

We evaluate the 10 leaks summarized in Table 1. Four are reported leaks from open-source programs; one is a leak in an application written by our colleagues; two are leaks in a benchmark program; and three are third-party microbenchmarks. The table shows lines of code and leak pruning's effect.

Leaky program executing with leak pruning fall into three categories: four execute for at least 24 hours, four execute longer than without leak pruning, and two do not execute any longer. Leak pruning fails to execute programs indefinitely when: (1) it prunes a reference that the program uses again, or (2) some or all of a program's heap growth is not dead. For the five leaks leak pruning cannot execute indefinitely other than short-running Delaunay (MySQL, SPECjbb2000, JbbMod, Mckoi, and DualLeak), leak pruning fails because some or all of the program's heap growth is *live*. In some cases, the memory is live because the programmer intentionally accesses leaked objects, e.g., SPECjbb2000 processes all objects in a list including those that the programmer intended to remove. In other cases, the program inadvertently accesses objects it no longer needs due to the data structure implementation. For example, when MySQL causes the size of one of its hash tables to grow, it accesses all the elements to re-hash them.

Other leak tolerance approaches that preserve semantics also cannot tolerate live leaks since the memory is in use [9, 10, 18, 47]. Leak pruning and Melt [9] perform about the same on all the leaks except one, JbbMod. (Our leaks are the same as in Melt since we obtained them from the Melt authors.) Melt (and LeakSurvivor [47]) can run it until the disk fills, while leak pruning runs it 21 times longer, presumably because leak pruning fails to identify and prune some fraction of the leak. On the other hand, leak pruning runs some leaks indefinitely without using disk space. A best-of-both-worlds approach could store leaks to disk and prune leaks to avoid running out of disk space. Here we evaluate the most challenging case: identifying and pruning leaks without using any disk space to increase the time until heap exhaustion.

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

EclipseCP. Eclipse is a popular integrated development environment (IDE) with 2 million lines of Java source [17]. Bug report #155889 states that when the user repeatedly cuts text, saves the file, pastes the text, and saves again, memory leaks. We reproduce this EclipseCP (cut-paste) leak by writing a plugin that repeatedly exercises this sequence with about 3 MB of text. Each instance of cut-save-paste-save is an iteration. Leak pruning primarily prunes two types of references: org.eclipse.jface.text.DefaultUndoManager$TextCommand → String and org.eclipse.jface.text.DocumentEvent → String. The String objects point to large char[] objects containing the cut-paste text. Later, leak pruning selects 19 additional types of edges before the 24-hour limit.

Figure 8 shows reachable memory over iterations of EclipseCP using a logarithmic x-axis. Without leak pruning, it quickly runs out of memory after 11 iterations. Leak pruning reclaims enough reachable but dead memory to keep it running for 1,115 iterations, when it reaches the 24-hour limit. Average reachable memory grows very slowly, but we do not know if (1) the growth represents edge types with relatively small bytesUsed values that will eventually grow large enough to be reclaimed; (2) the growth is live, and leak pruning will be unable to reclaim all of it; or (3) the growth is due to object caching (common in Eclipse) that will get flushed. Figure 9 shows EclipseCP the time spent in each iteration. Leak pruning initially added high overhead (*Old*

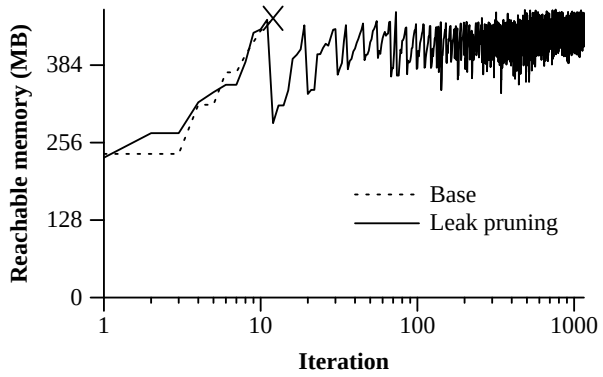

Figure 8. Reachable memory for EclipseCP with and without leak pruning (logarithmic x-axis).

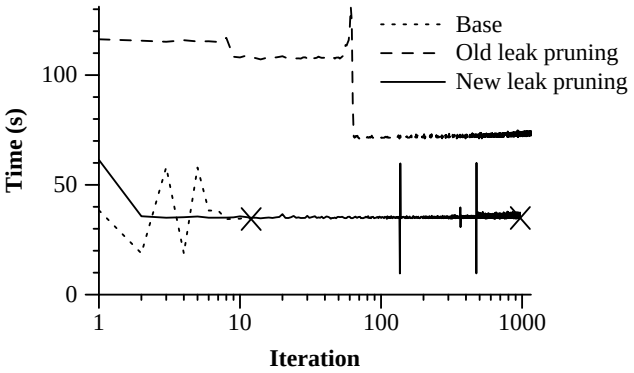

Figure 9. Time per iteration for EclipseCP with and without leak pruning (logarithmic x-axis).

leak pruning) because of an inefficiency in our implementation's read barriers that impacted EclipseCP significantly. We recently fixed this inefficiency (*New leak pruning*), but with this fix, leak pruning runs out of memory on EclipseCP before 24 hours. We will investigate for the final paper. Leak pruning keeps performance steady in the long term and runs over 100 times longer.

EclipseDiff. Eclipse bug #115789 performs a structural recursive compare (*diff*) and leaks memory in Eclipse 3.1.2. EclipseDiff reproduces it with a plugin that repeatedly performs structural diffs. The program leaks because each diff creates an entry in a component called NavigationHistory that points to objects of type ResourceCompareInput. The entries in the NavigationHistory and the ResourceCompareInput are not dead since Eclipse traverses the list and accesses them. However, a large subtree with the diff results is rooted at each ResourceCompareInput object and is dead. Leak pruning selects and prunes several edge types with source type ResourceCompareInput.

EclipseDiff with leak pruning should eventually exhaust memory since some heap growth is live, but the subtree rooted at each ResourceCompareInput is comparatively much larger, so leak pruning turns a fast-growing leak into a very

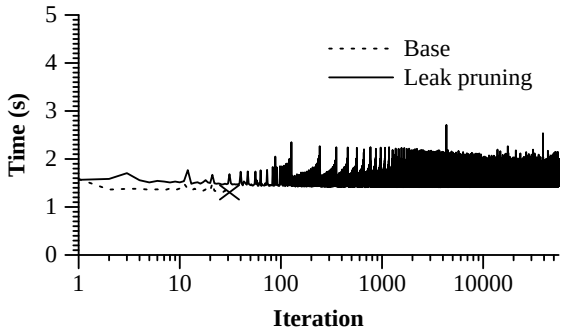

Figure 10. Time per iteration for EclipseDiff (logarithmic x-axis).

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

Figure 1 (page 3) shows reachable memory in the heap with and without leak pruning for its first 2,000 iterations. Figure 10 plots time for each iteration for all 55,780 iterations, using a logarithmic x-axis. Selection and pruning during GC extend some iterations up to 2X, but long-term throughput stays consistent.

ListLeak. ListLeak is a simple and fast-growing leak posted on the Sun Developer Network [46] that repeatedly adds, but does not remove, new objects to a linked list. To measure progress, we count every megabyte of allocation as an iteration. Each time leak pruning enters the SELECT state, leak pruning correctly selects a single reference with edge type java.util.LinkedList$Entry → java.util.LinkedList$Entry. Pruning this reference disconnects the stale part of the linked list, and the collector reclaims it.

SwapLeak. SwapLeak is also from a post on the Sun Developer Network [45]. It replaces a collection of objects with new objects but inadvertently retains the old objects. We put this behavior in a loop to create a growing leak. Leak pruning repeatedly identifies the correct edge type to prune to reclaim the old objects. Diagnosing this leak is challenging since objects are kept live by invisible references from an instance of an inner class back to its parent class instance, but leak pruning provides an automatic fix without needing to understand the leak.

MySQL. The MySQL leak is a simplified version of a JDBC application from a colleague. The application eventually exhausted memory unless it acquired 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 and re-hashes its elements. Although the hash table and statements are live, the

statements each reference a dead data structure with many more bytes, so leak pruning can significantly increase the lifetime of this leaky program. It correctly selects and prunes several types of references from statement objects, allowing the leak to execute 35 times longer.

SPECjbb2000. SPECjbb2000 is a Java benchmark that simulates an order processing system [44]. It has a known, growing leak that manifests when the program is run for a long time without changing warehouses. It adds but does not remove some orders. We count 100,000 SPECjbb2000 transactions as an iteration. Leak pruning cannot tolerate SPECjbb2000's leak indefinitely because the program accesses orders in the order list, keeping them live. However, leak pruning can still reclaim some memory. This leak grows very slowly. Leak pruning prunes 82 distinct edge types, most near the end of the run, sometimes netting fewer than 100 bytes. For example, leak pruning deletes character set objects in the class libraries that the application is not using. The program ultimately accesses a pruned reference. A production implementation of leak pruning could avoid pruning references that do not yield many bytes since they give little benefit.

JbbMod. Because SPECjbb2000 has significant *live* heap growth, Tang et al. modified it so much of its heap growth is stale [47]. We call this version JbbMod, and leak pruning runs it for about 10 hours before exhausting memory, executing 20X more iterations. Since other leak tolerance approaches handle this leak until they exhaust disk [9, 47], we believe that leak pruning fails to identify and prune some small fraction of leaked objects, and we plan to confirm this for the final paper.

Mckoi. Mckoi SQL Database is a database management system written in Java [29]. We reproduce a leak reported on a Mckoi message board: if a program repeatedly opens, uses, and closes a connection, the program leaks memory. Mckoi does not properly dispose of a thread associated with each connection. Our implementation currently cannot reclaim a thread's stack because it is a VM object, but we could modify it to detect leaked threads and prune them and their stacks. Without these modifications, leak pruning runs Mckoi 60% longer by reclaiming dead memory referenced by leaked threads.

DualLeak. DualLeak is a microbenchmark leak from an IBM developerWorks column [19]. We call it DualLeak because it has two different leaks in its 55 lines of source. The first leak is due to an off-by-one error that inadvertently leaves an Integer in a Vector on each iteration. The second leak adds multiple String objects to a HashSet. Leak pruning cannot reclaim any memory because all of the heap growth is live. The program accesses all the Integers in the Vector on each iteration. And when the HashSet grows, it accesses all the Strings in order to re-hash them.

Delaunay. Delaunay is an application provided by colleagues that generates a triangle mesh for a set of points, but inadvertently causes graph components removed from the graph to remain reachable. Unlike the other leaks, Delaunay is short-running: it generates a mesh and exits. It uses more memory than it needs to, but not an unbounded amount and we therefore omit it from the remainder of the results. Leak pruning does not have enough time to observe the leak and prune profitable references.

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

Table 2. Effectiveness of several selection algorithms. *Base* is unmodified Jikes RVM. The other columns are selection algorithms for leak pruning.

7. Accuracy and Sensitivity

This section examines implementation considerations. It shows that (1) the prediction mechanism in leak pruning is more accurate than just staleness or ignoring data structure sizes, (2) leak pruning is effective given various memory bounds imposed by the system on the application, (3) its space overhead is small, and (4) completely exhausting memory before pruning can initially degrade performance significantly.

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

Most stale. In the SELECT state, this algorithm identifies the highest staleness level of any object. In the DELETE state, it prunes all references to every object with this staleness level. This is effectively the same algorithm used by approaches that move objects to disk [9, 10, 18, 47].

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

Table 2 shows the effectiveness of these prediction algorithms measured in iterations. For example, EclipseCP with *Indiv refs* terminates after 41 iterations because the algorithm selects and prunes highly stale String → char[] references. The program later tries to use one of these references.

Leak	Live mem	Heap size / Longevity increase								Edge types
						Default				
EclipseCP	˜300M	300M	NH	400M	5.9X	500M	>100X	1000M	?	2,203
EclipseDiff	˜84M	100M	23X	150M	>2h	200M	>200X	500M	>2h	1,817
ListLeak	<40M	25M	NH	50M	>2h	100M	>25,000X	200M	>2h	56
SwapLeak	˜64M	75M	>2h	100M	>2h	200M	>2,200X	500M	>2h	83
MySQL	<40M	50M	NH	100M	42X	200M	35X	500M	>2h	230
SPECjbb2000	˜50M	55M	14X	65M	3.7X	75M	4.7X	100M	>4h	197
JbbMod	˜50M	55M	137X	65M	150X	75M	21X	100M	>2h	209
Mckoi	˜40M	32M	2.6X	48M	2.0X	64M	1.6X	128M	1.5X	308
DualLeak	<40M	32M	1.3X	48M	1.2X	64M	NH	128M	NH	69

Table 3. Leak pruning's effectiveness with various maximum heap sizes. *NH* means "no help." *>2h* ($> 4h$) means the program ran past a 2-hour (4-hour) limit. The last column is entries in the edge table.

In contrast, our default algorithm prunes reference types org.eclipse.jface.text.DefaultUndoManager$TextCommand → String and org.eclipse.jface.text.DocumentEvent → String, automatically reclaiming the growing, leaked Strings without accidentally deleting other live Strings. In general, our algorithm matches or outperforms the others since (1) it considers references' types (unlike *Most stale*) and (2) it considers data structures (unlike *Individual references*, which tries to identify each leaked reference type).

Varying the Heap Size. This section evaluates leak pruning's effectiveness in various maximum heap sizes since its actions are sensitive to heap fullness. Table 3 shows how leak pruning compares to unmodified Jikes RVM for a variety of heap sizes. The *Default* column shows the heap size we use by default for other experiments. It is based on the non-leaked memory (*Live mem* column): 1.5-2X, with larger sizes chosen for faster-growing leaks. Simple leaks with little non-leaked memory like ListLeak have arbitrary default heap sizes.

We select one heap size larger and two sizes smaller than the default because smaller heaps stress leak pruning more: it has less time and information to make good decisions. We impose a 2-hour time limit since there are many experiments; *>2h* means it appears leak pruning will perform similarly to the *Default* column. SPECjbb2000 needs 4 hours to run out of memory with a 100 MB heap. Strangely, we had trouble reproducing EclipseCP's leak when using leak pruning with a 1000M heap, but we will investigate for the final paper. In general, leak pruning performs similarly across heap sizes. Results vary in some cases because leak pruning makes different decisions about what to reclaim when there are no good candidates (the heap is full of live heap growth): it may reclaim memory that is used soon, or it may reclaim some part of the application or libraries that will not be used again. For example, EclipseDiff at the lowest heap size prunes live reference incorrectly since it does not have enough ceiling room to differentiate the growing, leaked memory from other stale memory in the heap.

Space overhead. Our implementation adds space overhead to store information about edge types in the edge table. For simplicity, it uses a fixed-size table with 16K slots using closed hashing [14]. Each slot has four words—source class, target class, maxStaleUse, and bytesUsed—for a total of 256K. A production implementation could size the table dynamically according to the number of edge types. The last column of Table 3 shows the number of edge types used by our implementation when it runs each leak. We simply measure the number of edge types in the edge table at the end of the run, since we never remove an edge type from the table. Eclipse is complex and uses a few thousand edge types; the database and JBB leaks are real programs but less complex and store hundreds of types; and the microbenchmark leaks store fewer than 100 edge types.

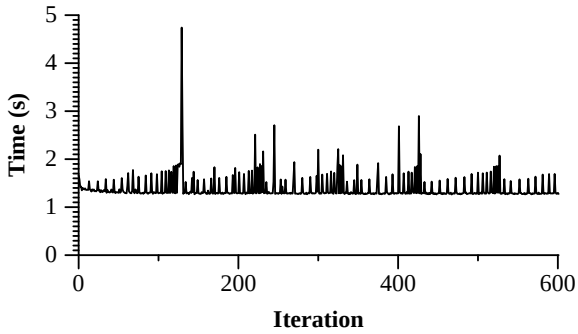

Figure 11. Time per iteration for EclipseDiff when it must exhaust memory prior to pruning.

Full heap threshold. By default, our implementation starts pruning references when the heap is 90% full (Section 3.2). However, the user can specify that leak pruning wait to prune references until the heap is 100% full, i.e., when it is just about to throw an out-of-memory error. Figure 11 shows the throughput of EclipseDiff for its first 600 iterations using a 100% heap fullness threshold. The first spike, at about 125 iterations, occurs because Eclipse slows significantly as GCs become very frequent: each GC reclaims only a small fraction of memory, so the next GC occurs soon after. Later spikes are smaller because leak pruning prunes references when the heap is only 90% full (since the program has already exhausted memory once); some of the overhead is due to the overhead of selecting and pruning references. The spike is about 2.5X greater than the other spikes, which may

be a reasonable trade-off in order to run the program as long as possible before commencing pruning.

8. Conclusion

Leak pruning is an automatic approach for bounding the memory consumption of programs with leaks, in many cases increasing availability significantly. It prunes references when the program runs out of memory and intercepts any future program accesses to these references. Leak pruning adds little or no overhead when there is no leak. If memory leaks, it preserves semantics while adding overhead low enough for production use. It improves the user experience while buying developers time to fix bugs, making it a compelling feature for production systems.

Acknowledgments

We thank Jason Davis for the MySQL JDBC leak; Patrick Carribault for the Delaunay mesh leak; Maria Jump for the original SPECjbb2000 leak; and Yan Tang for the modified SPECjbb2000 leak. Thanks to Eddie Aftandilian, Steve Blackburn, Curtis Dunham, Sam Guyer, Milind Kulkarni, Martin Rinard, Jennifer Sartor, and Craig Zilles for helpful discussions. We thank Nick Nethercote, Don Porter, and the anonymous reviewers for their helpful comments on the text.

References

[1] ALPERN, B., ATTANASIO, C. R., BARTON, J. J., BURKE, M. G., CHENG, P., CHOI, J.-D., COCCHI, A., FINK, S. J., GROVE, D., HIND, M., HUMMEL, S. F., LIEBER, D., LITVINOV, V., MERGEN, M., NGO, T., RUSSELL, J. R., SARKAR, V., SERRANO, M. J., SHEPHERD, J., SMITH, S., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. The Jalapeño Virtual Machine. *IBM Systems Journal 39*, 1 (2000), 211–238.

[2] ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., AND SWEENEY, P. F. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2000), pp. 47–65.

[3] BACON, D., CHENG, P., AND RAJAN, V. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *ACM Symposium on Principles of Programming Languages* (2003), pp. 285–298.

[4] BERGER, E. D., AND ZORN, B. G. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM Conference on Programming Language Design and Implementation* (2006), pp. 158–168.

[5] BLACKBURN, S. M., CHENG, P., AND MCKINLEY, K. S. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ACM International Conference on Software Engineering* (2004), pp. 137–146.

[6] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2006), pp. 169–190.

[7] BLACKBURN, S. M., AND HOSKING, A. L. Barriers: Friend or Foe? In *ACM International Symposium on Memory Management* (2004), pp. 143–151.

[8] BOND, M. D., AND MCKINLEY, K. S. Bell: Bit-Encoding Online Memory Leak Detection. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 61–72.

[9] BOND, M. D., AND MCKINLEY, K. S. Tolerating Memory Leaks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2008). To appear.

[10] BREITGAND, D., GOLDSTEIN, M., HENIS, E., SHEHORY, O., AND WEINSBERG, Y. PANACEA–Towards a Self-Healing Development Framework. In *Integrated Network Management* (2007), pp. 169–178.

[11] CHEREM, S., PRINCEHOUSE, L., AND RUGINA, R. Practical Memory Leak Detection using Guarded Value-Flow Analysis. In *ACM Conference on Programming Language Design and Implementation* (2007), pp. 480–491.

[12] CHILIMBI, T. M., AND HAUSWIRTH, M. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2004), pp. 156–164.

[13] CLICK, C., TENE, G., AND WOLF, M. The Pauseless GC Algorithm. In *ACM/USENIX International Conference on Virtual Execution Environments* (2005), pp. 46–56.

[14] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 2nd ed. The MIT Press, McGraw-Hill Book Company, 2001, ch. 11.

[15] DACAPO BENCHMARK REGRESSION TESTS. http://-jikesrvm.anu.edu.au/˜dacapo/.

[16] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM 21*, 11 (Nov. 1978), 966–975.

[17] ECLIPSE.ORG HOME. http://www.eclipse.org/.

[18] GOLDSTEIN, M., SHEHORY, O., AND WEINSBERG, Y. Can Self-Healing Software Cope With Loitering? In *International Workshop on Software Quality Assurance* (2007), pp. 1–8.

[19] GUPTA, S. C., AND PALANKI, R. Java memory leaks – Catch me if you can, 2005. http://www.ibm.com/-developerworks/rational/library/05/0816_GuptaPalanki/-index.html.

[20] HASTINGS, R., AND JOYCE, B. Purify: Fast Detection of Memory Leaks and Access Errors. In *Winter USENIX Conference* (1992), pp. 125–136.

[21] HEINE, D. L., AND LAM, M. S. A Practical Flow-Sensitive

and Context-Sensitive C and C++ Memory Leak Detector. In *ACM Conference on Programming Language Design and Implementation* (2003), pp. 168–181.

[22] HERTZ, M., FENG, Y., AND BERGER, E. D. Garbage Collection without Paging. In *ACM Conference on Programming Language Design and Implementation* (2005), pp. 143–153.

[23] HUANG, X., BLACKBURN, S. M., MCKINLEY, K. S., MOSS, J. E. B., WANG, Z., AND CHENG, P. The Garbage Collection Advantage: Improving Program Locality. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004), pp. 69–80.

[24] JIKES RVM. http://www.jikesrvm.org.

[25] JIKES RVM RESEARCH ARCHIVE. http://www.jikesrvm.-org/Research+Archive.

[26] JUMP, M., AND MCKINLEY, K. S. Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages. In *ACM Symposium on Principles of Programming Languages* (2007), pp. 31–38.

[27] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*, 2nd ed. Prentice Hall PTR, 1999.

[28] MAEBE, J., RONSSE, M., AND BOSSCHERE, K. D. Precise Detection of Memory Leaks. In *International Workshop on Dynamic Analysis* (2004), pp. 25–31.

[29] 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.

[30] MITCHELL, N., AND SEVITSKY, G. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming* (2003), pp. 351–377.

[31] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation* (2007), pp. 89–100.

[32] NGUYEN, H. H., AND RINARD, M. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ACM International Symposium on Memory Management* (2007), pp. 15–29.

[33] NOVARK, G., BERGER, E. D., AND ZORN, B. G. Plug: Automatically Tolerating Memory Leaks in C and C++ Applications. Tech. Rep. UM-CS-2008-009, University of Massachusetts, 2008.

[34] OGATA, K., ONODERA, T., KAWACHIYA, K., KOMATSU, H., AND NAKATANI, T. Replay Compilation: Improving Debuggability of a Just-in-Time Compiler. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2006), pp. 241–252.

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

[36] PIZLO, F., FRAMPTON, D., PETRANK, E., AND STEENSGAARD, B. Stopless: A Real-Time Garbage Collector for Multiprocessors. In *ACM International Symposium on Memory Management* (2007), pp. 159–172.

[37] QIN, F., LU, S., AND ZHOU, Y. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *International Symposium on High-Performance Computer Architecture* (2005), pp. 291–302.

[38] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In *ACM Symposium on Operating Systems Principles* (2005), pp. 235–248.

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

[40] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND BEEBEE, W. Enhancing Server Availability and Security through Failure-Oblivious Computing. In *USENIX Symposium on Operating Systems Design and Implementation* (2004), pp. 303–316.

[41] SACHINDRAN, N., MOSS, J. E. B., AND BERGER, E. D. MC^2: High-Performance Garbage Collection for Memory-Constrained Environments. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004), pp. 81–98.

[42] SCITECH SOFTWARE. .NET Memory Profiler. http://www.-scitech.se/memprofiler/.

[43] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECjvm98 Documentation*, release 1.03 ed., 1999.

[44] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECjbb2000 Documentation*, release 1.01 ed., 2001.

[45] SUN DEVELOPER NETWORK FORUM. Java Programming [Archive] - garbage collection dilema (sic), 2003. http://-forum.java.sun.com/thread.jspa?threadID=446934.

[46] SUN DEVELOPER NETWORK FORUM. Reflections & Reference Objects - Java memory leak example, 2003. http://-forum.java.sun.com/thread.jspa?threadID=456545.

[47] TANG, Y., GAO, Q., AND QIN, F. LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages. In *USENIX Annual Technical Conference* (2008), pp. 307–320.

[48] YANG, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. CRAMM: Virtual Memory Support for Garbage-Collected Applications. In *USENIX Symposium on Operating Systems Design and Implementation* (2006), pp. 103–116.