Leak Pruning *

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, because garbage collection uses reachability to over-approximate live objects, 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 eventually crash programs.

This paper introduces *leak pruning*, which keeps programs running by predicting and reclaiming leaked objects at run time. It predicts dead objects and reclaims them based on observing data structure usage patterns. Leak pruning *preserves semantics* because it waits for heap exhaustion before reclaiming objects and *poisons* references to objects it reclaims. If the program later tries to access a poisoned reference, the virtual machine (VM) throws an error. We show leak pruning has low overhead in a Java VM and evaluate it on 10 leaking programs. Leak pruning does not help two programs, executes five substantial programs 1.6-81X longer, and executes three programs, including a leak in Eclipse, for at least 24 hours. In the worst case, leak pruning defers fatal errors. In the best case, it keeps leaky programs running with preserved semantics and consistent throughput.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Debuggers, Memory management, Run-time environments

General Terms Reliability, Performance, Experimentation

* This work is supported by an Intel Ph.D. fellowship, NSF CCF-0811524, NSF CCF-0429859, NSF CNS-0719966, NSF EIA-0303609, Intel, IBM, Cisco, and Microsoft. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

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 [7, 19, 22, 27, 30, 32]. Leaks are hard to reproduce, find, and fix because they have no immediate symptoms [16]. 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 input and environment sensitive.

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, the leak pruning-enabled VM intercepts the access and throws an error. This behavior preserves semantics since the program already ran out of memory. In the worst case, leak pruning only defers out-of-memory errors. In the best case, it enables leaky programs with unbounded reachable memory to run indefinitely in bounded memory. In this case, leak pruning provides the illusion that the garbage collector is liveness-based rather than reachability-based.

Prior work tolerates leaks by identifying *stale* objects not used in a while and offloading them to disk [8, 9, 15, 35]. These systems tolerate mispredictions by retrieving objects from disk. Staleness alone is too imprecise for leak pruning's more aggressive approach, as we show experimentally in Section 6.1.

Leak pruning uses a new, dynamic prediction algorithm that considers both staleness and data structure usage. Our algorithm piggybacks on the garbage collector to identify stale *data structures*, i.e., stale subgraphs in the heap. It records the source and target classes of the first reference into a stale subgraph and the size of the subgraph. 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 3-5% to execution time due to its software read barrier (instrumentation at every read [6]). Although our implementation is for Java, the approach is applicable to other garbage-collected languages. We evaluate ten leaking programs, including two leaks in Eclipse. For two leaks, leak pruning provides no help. It executes two programs 1.6-4.7X longer and three leaks 21-81X longer. The remaining three leaky programs execute for at least 24 hours, when we terminate them. In all cases but one, when leak pruning cannot defer an out-of-memory error indefinitely, the program's working set is growing and thus the objects are *live*. 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 it should be a configuration parameter at deployment time.

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

2. Motivation, Background, and Semantics

This section gives motivation, background, and semantics for leak pruning. Section 3 overviews the approach and Section 4 presents algorithmic and implementation details.

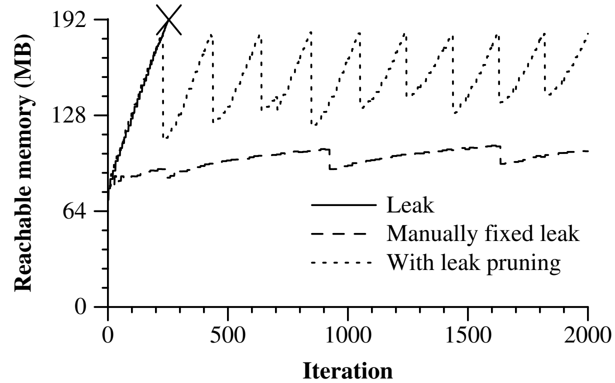

Figure 1. Reachable heap memory for the EclipseDiff leak: an unmodified VM running the leak and a manually fixed version, and a VM with leak pruning running the leak.

Even well-tested and widely used applications can contain leaks. Consider the example in Figure 1, which shows the memory consumption over time measured in *iterations* (fixed amounts of program work) for a 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 with a 200 MB maximum heap size. The solid line shows that the leak causes reachable memory to grow without bound until the VM throws an out-of-memory error. The dashed line shows reachable memory if we modify the source code to fix the leak, resulting in fairly constant reachable 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).

Garbage collection and memory exhaustion. Garbage collection (GC) reclaims only unreachable memory. *Reachability* approximates *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*[1] collector performs a transitive closure over the heap starting from the *roots*, which include stack pointers, global variables, and references in registers. The collector retains all transitively reachable objects and reclaims all memory used by unreachable objects. The next collection occurs after the sum of this reachable memory plus new allocation exceeds the available heap memory.

When an application exceeds the available heap memory and triggers a collection or exhausts memory, is not well defined because of collector and VM implementation details, including object header sizes, collector meta-data, and choice of collector algorithm. Increasing the maximum heap

[1] Our discussion and implementation use a tracing collector. Reference counting must also trace to collect cycles.

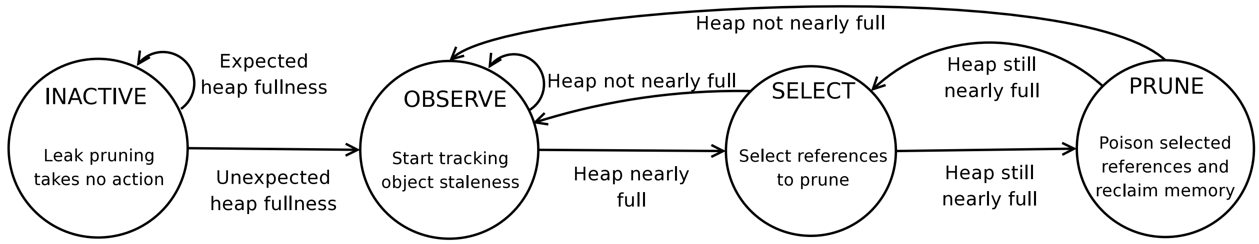

Figure 2. State diagram for leak pruning.

size helps some leaky programs. However, many embedded systems have hard upper bounds on maximum memory. Even with virtual memory and swap space on stock hardware, physical memory sizes effectively limit the heap's upper bound because exceeding this bound causes the collector to thrash since its working set is the entire heap [18, 37].

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*, which (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.

Leak pruning seeks to close the gap between liveness and reachability by providing functionality and performance consistent with GC that is based on liveness instead of 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, leak pruning *poisons* references to these objects and reclaims them. If the application subsequently attempts to read a poisoned reference, the VM throws an error, giving the original out-of-memory error as the cause. Since the program has *executed beyond* an out-of-memory error, throwing an error does not violate semantics. Leak pruning's goal is to defer out-of-memory errors indefinitely by eliminating space and time overheads due to leaks.

Exception and collection semantics. The Java VM specification says an OutOfMemoryError may be thrown only at program points responsible for allocating resources, e.g., new expressions or expressions that may trigger class initialization [20]. In general, programs will access pruned references at other points. However, the specification permits InternalError to be thrown asynchronously at any program point. Our implementation thus throws an InternalError if the program accesses a pruned reference.

When the VM runs out of memory, leak pruning records and defers the error. However, if the application can catch and handle the out-of-memory error, then deferring the error violates semantics. Catching out-of-memory errors is uncommon since these errors are not easy to remedy. In Java, a regular try { ... } catch (Exception ex) { ... } will not catch an OutOfMemoryError since it is on a different branch of the Throwable class hierarchy. Some applications, such as Eclipse, catch all errors in an outer loop and allow other components to proceed, but the Eclipse leaks we evaluate cannot perform 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 the program exhausts 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 semantics. 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 developers and users in order to avoid exhausting other resources while tolerating memory leaks.

3. Leak Pruning Overview

Figure 2 shows a high-level state diagram for leak pruning. State changes are based on how close the program is to running out of memory. Leak pruning performs most of its work during full-heap garbage collections. It changes state (or stays in the same state) depending on how full the heap is at the *end* of every full-heap collection.

3.1 Triggering Leak Pruning

Initially, leak pruning is INACTIVE and does not observe program behavior. This state avoids the overhead of leak pruning's analysis when the program is not running out of memory. Subsequent analysis focuses on the most recent behavior. Leak pruning remains INACTIVE until reachable memory exceeds "expected memory use," a configurable threshold. We use a 50% default threshold 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 and then analyzes program reference patterns to choose pruning candidates. Once leak pruning enters the OBSERVE state, it never returns to INACTIVE because it permanently considers the application to be in an unexpected state. Leak pruning moves from OBSERVE to SELECT when the program has nearly run out of memory, which is configurable and 90% by default. The SELECT state chooses references to prune, based on information collected during the OBSERVE state.

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) moving to PRUNE when the heap is still 100% full after a collection and the VM is about to throw an out-of-memory error or (2) moving to the PRUNE state after finishing a collection in the SELECT state. In either case, after entering PRUNE once, leak pruning always enters PRUNE on the *next* collection after entering SELECT, since the program has exhausted memory at least once. We believe (2) is more appealing since it avoids the VM grinding to a halt before pruning can commence. Option (2) does not generally violate program semantics because the VM has flexibility in how it reports memory usage, as discussed in Section 2. Programmers should consider the "nearly full" threshold to be the maximum heap size and "full" to be the extra headroom to perform GC efficiently. We use (2) by default and also evaluate (1).

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, identifying more references to prune.

Figure 3 shows an example heap after SELECT, when entering 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.

3.2 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 via object e1.

Leak pruning poisons a reference by setting its second-lowest-order bit (Section 4.3). 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 whose getCause() method returns the original OutOfMemoryError that *would have been thrown previously*. This behavior preserves semantics since the program previously ran out of memory when it entered the PRUNE state for the first time. To help programmers, leak pruning optionally reports (1) an out-of-memory "warning" when the program first runs out of memory and (2) the data structures it prunes.

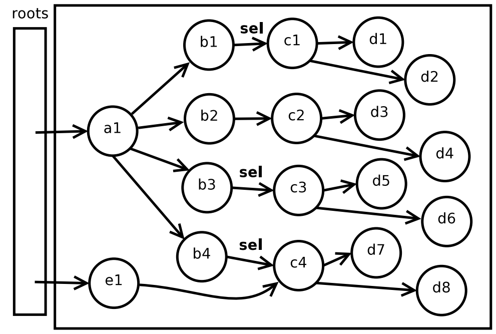

Figure 3. Example heap after the SELECT state. References selected for pruning are marked with **sel**.

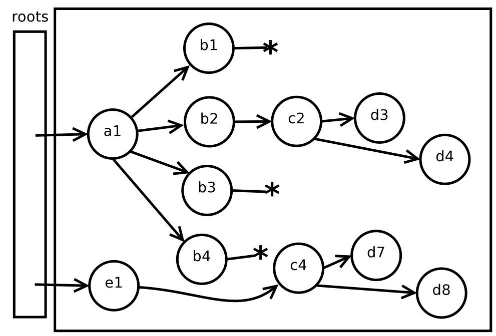

Figure 4. Example heap at the end of GC in the PRUNE state. Poisoned references end in an asterisk (*).

4. Algorithm and Implementation

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

Our prediction algorithm has the following objectives: (1) perfect accuracy, (2) high coverage, and (3) low time and space overhead. If prediction 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. 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 2).

4.1 The OBSERVE State

Tracking staleness. The OBSERVE state 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 [7]. A value k in an object's stale counter means the program last used the object approximately 2^k collections ago. We maintain each stale counter's value by (1) incrementing object counters in each collection and (2) inserting instrumentation that clears an object's counter when the program uses it.

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 instrumentation to test quickly whether the target object's stale counter has been reset since the last collection [8].

We modify the VM's just-in-time compiler to insert *read barriers* [6] at reference loads, e.g., `b = a.f`, that set `b`'s stale counter to zero, as shown in the following pseudocode.

```
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. The instrumentation is efficient because it takes no action if the lowest bit of the reference (`a.f`) is cleared. Since the VM initializes the bit to zero for all newly allocated objects, the barrier condition is true at most once for each reference after each full-heap collection. Since the barrier's body does not execute in the common case, we force the compiler to put it *out-of-line* in a separate method.

The barrier updates the reference *atomically* with respect to the read 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 atomic update fails, the barrier simply continues, which is a valid serialization because another thread has written a valid reference to `a.f`, and the current thread can safely use `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, these atomic updates add unnoticeable overhead.

Edge table. The OBSERVE state starts maintaining an *edge table* to track the staleness of heap references based on type. For a stale edge in the heap, $src \rightarrow tgt$, the table records the Java class of the source and target objects: $src_{class} \rightarrow tgt_{class}$. Each entry summarizes an equivalence relationship over object-to-object references: two references are equivalent if their source and target objects each have the same class. Each edge entry $src_{class} \rightarrow tgt_{class}$ records bytesUsed (for use 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 value. We record in maxStaleUse the all-time maximum value of *tgt*'s stale counter when a barrier accesses a reference $src_{class} \rightarrow tgt_{class}$. The read barrier executes the following code as part of its out-of-line cold path.

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

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

4.2 The SELECT State

A full-heap collection in SELECT chooses *one* edge type for pruning during a subsequent GC in the PRUNE state. It divides the regular transitive closure, which marks all reachable objects, into the following 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 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 add them to a *candidate queue*.
2. The *stale transitive closure* marks objects live, starting with references in the candidate queue. These references point to *stale roots* of data structures. The stale closure

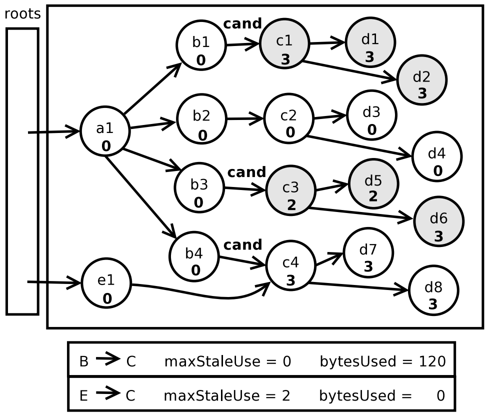

Figure 5. Example heap during the SELECT state.

computes the bytes reachable from each stale root, i.e., the size of the stale data structure, and adds this value to bytesUsed for the stale root's edge entry.

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 resets all bytesUsed values. The PRUNE state then prunes data structures that match this edge type.

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 since its stale counter would need to be at least 4 (2 more than the maxStaleUse of 2 for E → C). 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.

4.3 The PRUNE State

The PRUNE state performs only the in-use closure, during which it prunes all references corresponding to the selected edge type and whose target objects have staleness values that are at least two more than the entry's maxStaleUse. The collector poisons each reference in the candidate set by setting its *second-lowest* bit, as well as its lowest bit. The collector does not trace the reference's target. Future collections see the reference is poisoned and do not dereference it.

4.4 Intercepting Accesses to Pruned References

To intercept program accesses to pruned references, the barrier also checks for poisoned references. The following check is performed at the beginning of the barrier's cold path:

```
if (b & 0x2) {  // Check if poisoned
  InternalError err = new InternalError();
  err.initCause(avertedOutOfMemoryError);
  throw err;
}
/* rest of read barrier cold path */
```

If the reference is poisoned, the barrier throws an InternalError with the original OutOfMemoryError attached.

4.5 Concurrency and Thread Safety

Our implementation supports multithreaded programs executing on multiple processors. Above, we discussed how atomic updates in the read barrier preserve thread safety. The edge table is a global structure that can be updated by multiple threads in read barriers or during collection. We need global synchronization on the edge table only when adding a new edge type, which is rare, and we never delete an edge table entry. When updating an entry's data, our implementation should use fine-grained synchronization to protect the entry. However, our prototype implementation does not synchronize these updates since we expect conflicts to be rare, and edge selection is not sensitive to exact values of bytesUsed and maxStaleUse.

By default, the garbage collector is parallel [4]. 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 presents our performance evaluation methodology and shows that the overheads of leak pruning are low.

VM configurations. We implement leak pruning in Jikes RVM 2.9.2,[2] a high-performance Java-in-Java virtual machine [1, 2]. As of August 2008, Jikes RVM performs the same as Sun Hotspot 1.5, and 15 to 20% worse than Hotspot 1.6, JRockit, and J9 1.9, all configured for high performance.[3] Our performance measurements are therefore relative to an excellent baseline. We have made our implementation publicly available on the Jikes RVM Research Archive.[4]

Jikes RVM recompiles hot methods with increasingly aggressive optimizations. Because timer-based sampling iden-

[2] `http://www.jikesrvm.org`

[3] `http://jikesrvm.anu.edu.au/~dacapo/`

[4] `http://www.jikesrvm.org/Research+Archive`

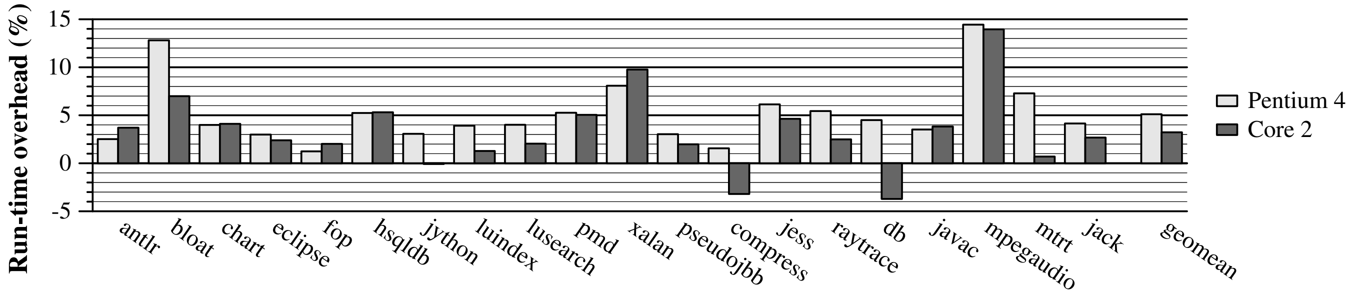

Figure 6. Run-time overhead of leak pruning on two platforms.

tifies hot methods, compilation decisions are nondeterministic. To achieve determinism in performance experiments, we use *replay* compilation [5, 14] to produce the same compilation decisions on different runs. Replay executes two iterations. The first includes compilation. We report the second, which executes only the application and is representative of steady-state application behavior.

Our experiments add leak pruning to a parallel, stop-the-world, generational mark-sweep collector in the Memory Management Toolkit (MMTk) [4]. MMTk supports a variety of garbage collectors with most functionality residing in shared code. Our implementation resides almost exclusively in this shared code, allowing leak pruning to support other collectors by specifying which space(s) contain objects that leak pruning should track.

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 [5, 33, 34].

Platform. All experiments execute on a dual-core 3.2 GHz Pentium 4 system. 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. Additionally, we measure read barrier overheads on a Core 2 Quad 2.4 GHz system. 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. Both systems have 2 GB of main memory and run Linux 2.6.20.3.

Application overhead. Leak pruning adds overhead because it inserts read barriers into application code, tracks staleness, and selects references to prune during garbage collection. Using replay compilation, Figure 6 includes application and collection, but not compilation overheads. Each bar is the median overhead of five trials. To control the memory size, we fix the heap at two times the minimum in which each benchmark can run. The two bars are overhead on the Pentium 4 and Core 2, respectively.

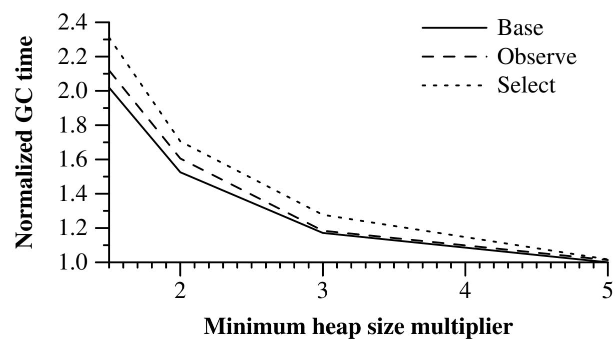

Figure 7. Normalized collection times across heap sizes (y-axis starts at 1.0).

The bars show the overhead of exercising leak pruning: even though these benchmarks do not leak memory, we force leak pruning to be in the SELECT state continuously. However, we find that virtually all run-time overhead comes from the overhead of read barriers; tracking staleness and selecting references add negligible overhead. On average, overhead is 5% on the Pentium 4 and 3% on the Core 2.

Non-leaking programs do not need barriers as long as leak pruning remains in the INACTIVE state. For simplicity, our implementation uses all-the-time barriers, but a production implementation should trigger recompilation of all methods with read barriers only when leak pruning enters the OBSERVE state. With the increasing importance of concurrent software, future general-purpose hardware is likely to provide read barriers with no overhead, and Azul hardware has them already [12].

Garbage collection overhead. Figure 7 plots the geometric mean of normalized GC time on the P4 (Core 2 times are similar) over all the benchmarks as a function of heap sizes 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. Base is GC time on unmodified Jikes RVM. *Observe* forces leak pruning to be in the OBSERVE state all the time, which involves maintaining each object's staleness bits dur-

Leak (LOC)	Effect	Reason
EclipesDiff (2.4M)	Runs >200X longer	Almost all reclaimed
ListLeak (9)	Runs indefinitely	All reclaimed
SwapLeak (33)	Runs indefinitely	All reclaimed
EclipseCP (2.4M)	Runs 81X longer	Almost 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.

ing collection and updating maxStaleUse for edge types that are used after being stale for a while. *Observe* adds up to 5% overhead. *Select* adds the rest of leak pruning's functionality without actually pruning references: performing the stale trace and selection of an edge type to prune. This configuration adds up to 9% more to GC time, for a total of 14%.

Compilation overhead. Inserting read barriers adds compiler overhead by bloating the intermediate representation (IR) and thus increasing 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 iteration of replay compilation. 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. Read barriers increase code size by 10% on average and 15% at most (for javac).

6. Tolerating Leaks

We evaluate 10 leaks, summarized in Table 1. Four are reported leaks from open-source programs (EclipseDiff, EclipseCP, MySQL, Mckoi); one is a leak in an application written by our colleagues (Delaunay); two are leaks in a benchmark program (SPECjbb2000, JbbMod); and three are third-party microbenchmarks (ListLeak, SwapLeak, DualLeak). The table shows lines of code and leak pruning's effect. Each program executes in a heap chosen to be about twice the size needed to run the program if it did not leak. We evaluate four other heap sizes for each leak and find leak pruning's effectiveness is generally not sensitive to maximum heap size, except that it sometimes fails to identify and prune the right references in tight heaps.

The programs fall into three categories: three execute for at least 24 hours, four execute longer with leak pruning than without, and two do not execute longer. Leak pruning fails to execute JbbMod indefinitely because it fails to select and prune key reference types. It fails to execute EclipseCP, MySQL, SPECjbb2000, Mckoi, and DualLeak indefinitely because some or all of their heap growth is *live*. In some cases, 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 rehash them.

Other leak tolerance approaches that preserve semantics also cannot tolerate live leaks since the memory is in use [8, 9, 15, 35]. Leak pruning and Melt [8] perform about the same on all the leaks except JbbMod and EclipseCP, as described below. However, while disk-based approaches fail when they run out of disk space, leak pruning can run some leaks indefinitely. Leak pruning and disk-based approaches are complementary, and a combined approach could get the benefits of both. Here we evaluate the most challenging case for leak pruning: identifying and pruning leaks without using any disk space.

Next we describe each leak and leak pruning's effect on it. For space, some descriptions are short; our prior work presents more leak details [8].

EclipseDiff. Eclipse is a popular integrated development environment (IDE) with 2 million lines of Java source.[5] Bug report #115789 states that performing a structural recursive compare (*diff*) 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, dead subtree with the diff results is rooted at each ResourceCompareInput object. Leak pruning correctly selects and prunes several edge types with source type ResourceCompareInput. We reported a fix for this leak [7], which developers applied in time for Eclipse 3.2.

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 slow-growing leak. We run EclipseDiff with leak pruning for 24 hours, and it does not run out of memory. Figure 1 shows reachable memory in the heap with and without leak pruning for its first 2,000 iterations. Figure 8 plots time for each iteration for 55,780 iterations, using a logarithmic x-axis. Leak pruning occasionally doubles an iteration's execution time, but long-term throughput is constant.

EclipseCP. Eclipse 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

[5] `http://www.eclipse.org/`

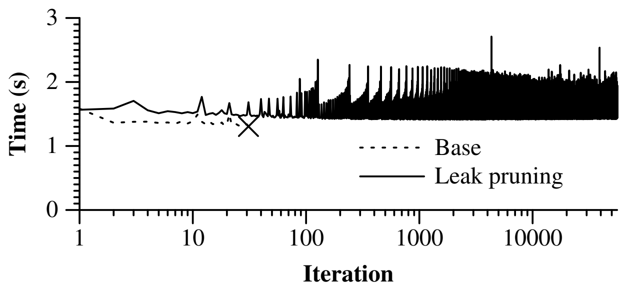

Figure 8. Time per iteration for EclipseDiff (logarithmic x-axis).

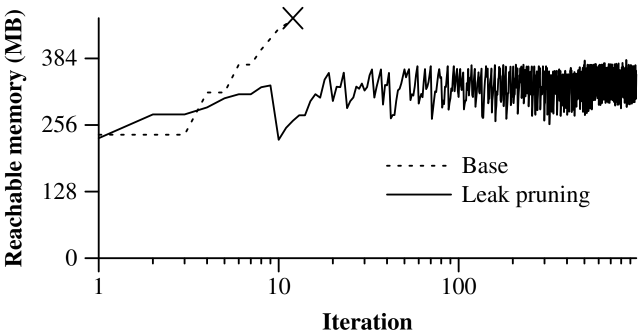

Figure 9. Reachable memory for EclipseCP with and without leak pruning (logarithmic x-axis).

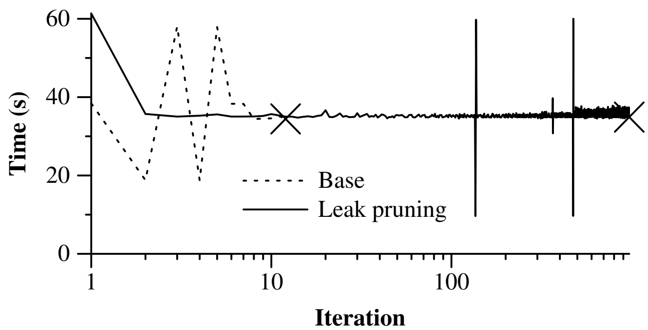

Figure 10. Time per iteration for EclipseCP with and without leak pruning (logarithmic x-axis).

(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.

Figure 9 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 971 iterations (9.5 hours). However, steady-state reachable memory slowly increases over time, either due to objects that our algorithm fails to prune, or object caches (common in Eclipse) that Eclipse would eventually flush before running out of memory. Initially, leak pruning repeatedly prunes the reference types org.eclipse.jface.text.DefaultUndoManager$TextCommand → String and org.eclipse.jface.text.DocumentEvent → String. Eventually, after about 490 iterations, space is so tight that the SELECT state chooses another reference type, and it ultimately reclaims over 100 different reference types before EclipseCP uses a reclaimed instance, terminating the program.

ListLeak, SwapLeak, *and* DualLeak. These leaks are simple and fast-growing examples posted to the Sun Developer Network[6] and IBM developerWorks.[7] Leak pruning tolerates ListLeak and SwapLeak indefinitely by repeatedly selecting and pruning the correct reference type. It cannot tolerate DualLeak, which involves *live* heap growth.

MySQL. The MySQL leak is a simplified version of a JDBC application from a colleague that exhausts memory unless it reacquires a connection periodically. The leak occurs because the JDBC library keeps already-executed SQL statements in a collection unless the connection or statements are explicitly closed. MySQL 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, each statement references a dead data structure with relatively many bytes, so leak pruning can significantly increase MySQL's lifetime. It correctly selects and prunes several types of references pointing from statement objects, allowing the program to execute 35 times as many iterations.

SPECjbb2000. SPECjbb2000 is a Java benchmark that simulates an order processing system [34]. We count 100,000 SPECjbb2000 transactions as an iteration. The program has a known, growing leak that manifests when it is run for a long time without changing warehouses. The leak occurs because it never removes some orders from an order processing list. 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.

JbbMod. Because SPECjbb2000 has significant *live* heap growth, Tang et al. modified it to make much of its heap growth stale [35]. We call this version JbbMod, and leak pruning runs it for about 10 hours before exhausting memory, executing 20X more iterations. We note Melt [8] and LeakSurvivor [35] tolerate this leak until they exhaust the disk. To determine why leak pruning fails sooner, we modified Melt to report the bytes used by different types of highly

[6] `http://forum.java.sun.com/thread.jspa?threadID=456545` and `http://forum.java.sun.com/thread.jspa?threadID=446934`

[7] `http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/index.html`

stale objects that it moves to disk. *Leak pruning* repeatedly selects and prunes spec.jbb.Orderline → java.lang.String → char[]. In addition to these types, *Melt* transfers many objects of types spec.jbbOrder, java.util.Date, and java.lang.Object[] to disk. From examining leak pruning's diagnostic output, it appears that objects of type Object[] point to Order objects, which point to Object[] and Date. Leak pruning does not prune references from Object[] to Order because this reference type's maxStaleUse value is high (5). Tolerating this leak longer would require a different policy, e.g., categorizing references some way other than source-target type, or periodically decaying each reference type's maxStaleUse value to account for possible phased behavior.

Mckoi. Mckoi SQL Database is a database management system written in Java. This leak[8] is primarily a thread leak. Our current implementation cannot reclaim a thread's stack, although it could be modified to do so. Leak pruning runs Mckoi 60% longer by selecting and pruning dead memory *referenced by* the leaked threads' stacks.

Delaunay. Delaunay is short running, so it is not clear if it truly leaks memory or simply keeps some memory reachable longer than it should. Unlike the other leaks, Delaunay does not use an unbounded amount of memory. Leak pruning does not have time to observe it and prune references.

6.1 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 algorithm is effectively the same as those that move objects to disk [8, 9, 15, 35].

Individual references. This algorithm modifies our default algorithm by eliding the candidate queue and the stale transitive closure from the SELECT state. The resulting algorithm prunes individual stale references rather than stale subtrees.

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, but live, String → char[] references. 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 String objects without deleting other live String objects. In general, our algorithm matches or outperforms the others since it considers reference types (unlike *Most stale*) and data structures (unlike *Individual references*).

[8] http://www.mckoi.com/database/mail/subject.jsp?id=2172

Leak	Base	*LS* [35] & *Melt* [8] Most stale	Indiv refs	*Leak pruning* Default	Default edge types
EclipseDiff	259	228	3,380	≥55,780	1,817
ListLeak	110	108	≥2.7M	≥2.7M	56
SwapLeak	5	5	11	≥11,368	83
EclipseCP	11	10	41	971	2,203
MySQL	18	35	114	634	230
SPECjbb2000	135	97	625	632	197
JbbMod	204	41	911	4,267	209
Mckoi	44	47	71	72	308
DualLeak	145	149	144	143	69

Table 2. Iterations executed by leak programs using leak pruning with several prediction algorithms. *Base* is unmodified Jikes RVM; *Most stale* is the algorithm used by LeakSurvivor [35] and Melt [8]; *Indiv refs* does not consider data structures; and *Default* is leak pruning's algorithm.

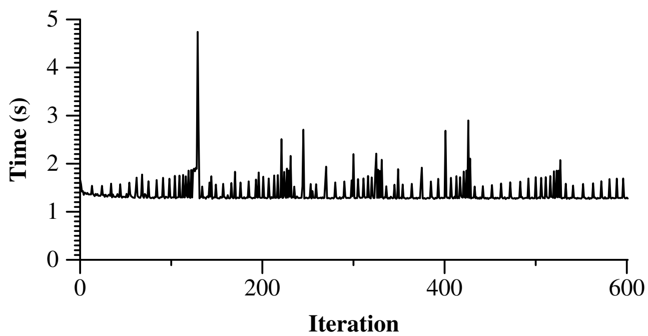

Figure 11. Time per iteration for EclipseDiff when it must exhaust memory prior to pruning.

6.2 Space Overhead

Our implementation adds space overhead to store information about edge types in the edge table. For simplicity, it uses a fixed-size table with 16K slots using closed hashing [13]. 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 2 shows the number of edge types used by leak pruning for each leak, measured at the end of the run because the table never shrinks. 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.

6.3 Full Heap Threshold

By default, our implementation starts pruning references when the heap is 90% full (Section 3.1). Optionally, it can wait to prune until the heap is 100% full, i.e., when the VM 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 successive pruning occurs when the heap is only 90% full (since the program has already exhausted memory once). The spike is about 2.5X taller than the other spikes, which may be a reasonable tradeoff to execute programs as long as possible before commencing pruning.

7. 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 [3, 29, 31]. One industrial response to leaks is restarting the application, but this mechanism reduces availability and loses application state that may not be recoverable. The prior work most closely related to leak pruning is *cyclic memory allocation* [25] and offloading leaks to disk [8, 9, 15, 35].

Detecting leaks. Static leak detectors for C and C++ identify objects that the programmer forgot to free and are unreachable [10, 17]. Dynamic leak detectors for C and C++ find these objects at run time by tracking allocations, frees, and pointer mutations [16, 21, 24] or by tracking staleness [11, 28]. Leak detectors for managed languages report dynamic heap growth [19, 22, 27, 30, 32] and stale objects [7]. Leak pruning uses staleness to predict liveness.

Tolerating leaks. *Cyclic memory allocation* seeks to bound memory usage by controlling the number of live objects produced by an allocation site to m [25]. Off-line profiling determines m and a modified allocator uses a cyclic buffer. Cyclic memory allocation assumes each allocation site produces a bounded footprint of live objects, but some leaks do not have this property. Cyclic memory allocation may change program semantics since the program is silently corrupted if it uses more than m objects, although *failure-oblivious computing* [31] often mitigates the effects.

Plug segregates objects at allocation time and re-maps virtual to physical pages to save physical memory in C and C++ programs [26]. It differs from leak pruning by using disk space and not addressing challenges presented by garbage collection.

Prior work considers memory management for programs with large working sets executing on Lisp machines [23, 36]. Solutions include incremental copying collection, segregating objects based on expected lifetime, hardware support for data type tags, and even avoiding garbage collection altogether by using tertiary memory storage.

LeakSurvivor, Panacea, and *Melt* tolerate leaks by transferring potentially leaked objects to disk [8, 9, 15, 35]. 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* may tolerate some leaks by saving physical, not virtual, memory and tracking staleness on page granularity [18].

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 to keep performance from suffering. All will eventually exhaust disk space and crash. Leak pruning requires perfect prediction and uses a more precise algorithm for predicting dead objects, as shown in Section 6.1. Leak pruning is less tolerant of errors because it must throw an error if it makes a mistake. However, it bounds memory usage, making it suitable when disk space runs out or no disk is available, e.g., in embedded systems.

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 likely leaked data structures when a program runs out of memory. It preserves semantics by intercepting any future accesses to pruned objects. Leak pruning adds overhead low enough for deployed use. It improves the user experience while buying developers time to fix bugs, making it a compelling feature for production systems.

Acknowledgments

We thank Partick Carribault, Jason Davis, Maria Jump, and Yan Tang for memory leaks. Thanks to Eddie Aftandilian, Steve Blackburn, Cliff Click, Curtis Dunham, Sam Guyer, Milind Kulkarni, Martin Rinard, Jennifer Sartor, and Craig Zilles for helpful discussions and ideas. 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] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] BOND, M. D., AND MCKINLEY, K. S. Tolerating Memory Leaks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2008), pp. 109–126.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] GEORGES, A., EECKHOUT, L., AND BUYTAERT, D. Java Performance Evaluation through Rigorous Replay Compilation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2008), pp. 367–384.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] MOON, D. A. Garbage Collection in a Large Lisp System. In *ACM Conference on LISP and Functional Programming* (1984), pp. 235–246.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] 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.

[32] SCITECH SOFTWARE. .NET Memory Profiler. `http://www.scitech.se/memprofiler/`.

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

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

[35] 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.

[36] WHITE, J. L. Address/Memory Management For A Gigantic LISP Environment or, GC Considered Harmful. In *ACM Conference on LISP and Functional Programming* (1980), pp. 119–127.

[37] 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.