Detecting memory leaks in managed languages with Cork*

Maria Jump[1] and Kathryn S McKinley[2]

[1] *King's College*
[2] *The University of Texas at Austin*

SUMMARY

A *memory leak* in a managed program occurs when the program inadvertently maintains references to objects that it no longer needs. Memory leaks cause systematic heap growth which degrades performance and results in program crashes after perhaps days or weeks of execution. Prior approaches for detecting memory leaks rely on heap differencing or detailed object statistics which store state proportional to the number of objects in the heap. These overheads preclude their use on the same processor for deployed long-running applications.

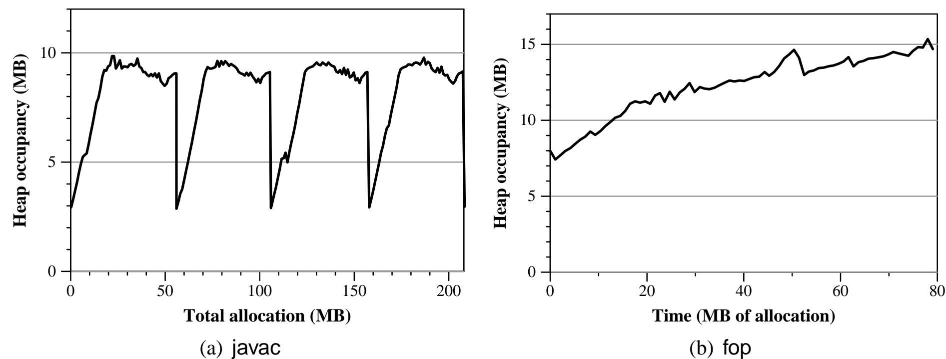
This paper introduces Cork as a tool that accurately identifies heap growth caused by leaks. It is space efficient (adding less than 1% to the heap) and time efficient (adding 2.3% on average to total execution time). We implement this approach of examining and summarizing the class of live objects during garbage collection in a *class points-from graph* (*CPFG*). Each node in the *CPFG* represents a class and and edges between nodes represent references between objects of the specific classes. Cork annotates nodes and edges with the corresponding volume of live objects. Cork identifies growing data structures across multiple collections and computes a *class slice* to identify leaks for the user. We experiment with two functions for identifying growth and show that Cork is accurate: it identifies systematic heap growth with no false positives in 4 of 15 benchmarks we tested. Cork's slice report enabled us to quickly identify and eliminate growing data structures in large and unfamiliar programs, something their developers had not previously done.

KEY WORDS: memory leaks, runtime analysis, dynamic, garbage collection

1 Introduction

Memory-related bugs are a substantial source of errors, especially for languages with explicit memory management. For example, C and C++ memory-related errors include (1) *dangling pointers* – dereferencing pointers to objects that the program previously freed, (2) *lost pointers* – losing all pointers to objects that the program neglects to free, and (3) *unnecessary references* – keeping pointers

*Correspondence to: Department of Mathematics and Computer Science, King's College, 133 North River Street, Wilkes-Barre, PA 18711, USA.
Contract/grant sponsor: This work is supported by NSF SHF-0910818, NSF CCF-0811524, NSF CNS-0719966, NSF CCF-0429859, IBM, and CISCO. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.



(a) javac (b) fop

Figure 1. Heap-Occupancy Graphs

to objects the program never uses again. Garbage collection precludes the first two errors, but not the last. Since garbage collection is conservative, it cannot detect or reclaim objects referred to by unnecessary references. Thus, a *memory leak* in a garbage-collected language occurs when a program maintains references to objects that it no longer needs, preventing the garbage collector from reclaiming space. In the best case, unnecessary references degrade program performance by increasing memory requirements and consequently collector workload. In the worst case, a leaking, growing data structure will cause the program to run out of memory and crash. In long-running applications, small leaks can take days or weeks to manifest. These bugs are notoriously difficult to find because the allocation that finally exhausts memory is not necessarily related to the source of the heap growth.

To demonstrate leaks, we measure heap composition using heap-occupancy graphs [8, 21, 39]. A heap-occupancy graph plots the total heap occupancy on the y-axis over time measured in allocation on the x-axis by collecting the entire heap very frequently (i.e., every 10K of allocation). Figure 1 shows the heap occupancy graphs for javac from SPECjvm [37] and fop from DaCapo [9]. A heap-occupancy curve with an overall positive slope clearly indicates systematic heap growth. Figure 1(a) shows four program allocation phases for javac that reach the same general peaks and indicate that javac uses about the same maximum amount of memory in each phase and no phase leaks memory to the next. On the other hand, Figure 1(b) shows that the heap occupancy graph for fop continue to grow during execution. Such growth indicates the presence of potential leaks, but they do not pinpoint the source of the leak.

Previous approaches for finding leaks use heap diagnosis tools that rely on a combination of heap differencing [15, 16, 30] and allocation and/or fine-grain object tracking [10, 13, 14, 19, 29, 35, 36, 40, 41]. These techniques can degrade performance by a factor of two or more, incur substantial memory overheads, rely on multiple executions, and/or offload work to a separate processor. Additionally, they yield large amounts of low-level details about individual objects. For example, if a `String` leaks, they can report individual `String`s numbering in the 10,000s to 100,000s and allocation sites numbering in the 100s to 1000s. Interpreting these reports requires a lot of time and expertise. Some prior work reports *stale* objects that the program is no longer accessing [10, 14, 29]. A problem with

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

using staleness is that data structures such as arrays and hash tables occasionally touch all the objects when the data structure grows, which defeats leak detection based only on stale objects. Prior work thus lacks efficiency and precision. Our approach is efficient because it piggyback on the garbage collector and it is precise because it identifies growing data structures that cause heap growth

We introduced *Cork*, an accurate, scalable, and online memory leak detection tool for typed garbage-collected languages in prior work [23]. This paper extends our prior work with more rigorous descriptions and a comparison of leak identification algorithms. It describes in more detail our pruning techniques and experimental results, including describing the leaks Cork finds in Java programs and how to fix them.

Cork uses a novel approach to summarize, identify, and report data structures with systematic heap growth. We show that it provides both efficiency and precision. For performance efficiency, Cork piggybacks on full-heap garbage collections. As the collector scans the heap, Cork summarizes the dynamic object graph by summarizing objects by their user-defined class in *class points-from graph* (*CPFG*). The nodes of the graph represent the volume of live objects of each class. The edges represent the points-from relationship between classes weighted by volume. At the end of each collection, the *CPFG* completely summarizes the live-object points-from relationships in the heap.

For space efficiency, Cork stores class nodes together with their global *type information block* (TIB). The TIB, or equivalent, is a required implementation element for managed languages, such as Java and C#, that instructs the compiler on how to generate correct code and instructs the garbage collector on how to scan objects. The number of nodes in the *CPFG* scales with the number of loaded classes. While the number of edges between classes are quadratic in theory, programs implement simpler class relations in practice; we find that the edges are linear in the number of classes.

Cork uses multiple *CPGF*s to detect and report a dynamic *class slice* with systematic heap growth. We show that even with multiple *CPFG*, Cork never adds more than 0.5% to heap memory. Additionally, this paper compares two heuristics for detecting leaks and controlling for natural variations in the heap object graphs: *Slope Ranking* and *Ratio Ranking*. We find that although slope ranking is more principled, ratio ranking works better. Slope ranking computes the slope between heap summaries. When the heap volume fluctuates a lot, it requires more summaries to accurately find slow leaks. Ratio ranking instead accumulates cumulative heap growth statistics from each summary, which makes it accurate even when the heap size fluctuates a lot and the leak is slow. We store points-from instead of points-to information to efficiently compute the candidate class slice from a growing node. We demonstrate that the construction and comparison of *CPFG*s across multiple collections adds on average 2.3% to total time to a system with a generational collector.

We apply Cork to 15 Java programs: one from an Eclipse developer report and the others from the DaCapo b.050224 [9] and SPECjvm [37, 38] benchmarks. Cork precisely identifies and reports unbounded heap growth in four of them:

Eclipse configured to exercise a known leak in comparing files (Eclipse bug #115789) grows 2.97MB every 64MB of allocation. Due to the size and complexity of Eclipse and our lack of experience with the implementation, we needed about three and a half days to find and fix this leak using the Cork report.

fop grows 4.8MB every 64MB of allocation in a data structure which is in use throughout the entire program. While Cork precisely pinpoints a single growing data structure, it does not sample or track individual access to heap objects [14] so it cannot distinguish between a growing data

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

structure which will never be used again from one that is still in use. Regardless, even systematic heap growth in a data structure that is still in use is a cause for concern as it can affect application reliability and performance.

jess grows 45KB every 64MB of allocation. Cork precisely pinpoints a single growing data structure that contains a growing number of objects that are never used again.

jbb2000 grows 127KB every 64MB of allocation. Cork's precision pays off allowing us to quickly fix the memory leak that had eluded developers for many years. We found it and fixed it in a day with Cork's report.

We confirm there are no additional memory leaks in the other 11 benchmarks by examining their heap composition graphs [8], showing Cork is accurate on all the programs we tested. In practice, Cork's novel summarization technique is efficient and precisely reports data structures responsible for systematic heap growth. Its low space and time overhead makes it appealing for periodic or consistent use in deployed production systems.

2 Related Work

The problem of detecting memory leaks falls in three categories: static analysis detection, heap differencing, and online staleness detection. Compile-time static analysis can find double free and missing frees [20] and is complimentary to our work. Offline diagnostic tools accurately detect leaks using a combination of heap differencing [15, 16, 30] and fined-grained allocation/usage tracking [13, 19, 35, 36, 40, 41]. These approaches are expensive and often require multiple executions and/or separate analysis to generate complex reports full of low-level details about individual objects. In contrast, Cork's completely online analysis reports summaries of objects by class while concisely identifying the dynamic data structure containing the growth. Other online diagnostic approaches rely on detecting when objects exceed their expected lifetimes [29] and/or detecting when an object becomes *stale* [10, 14]. This work differentiates in-use objects from those not in-use. We instead detect growing data structures, which finds leaks when the program keeps touches these objects, such as when growing a hash table. Approaches based on staleness miss these cases.

Static approaches, for example Heine and Lam [20], rely on compile-time analysis to detect memory leaks. Here a pointer analysis identifies potential memory leaks in C and C++ using the object ownership abstraction. They find double frees and missing frees that occur when the program overwrites the most recent pointer to an object or data structure without first freeing it. It does not find growing data structures and thus static approaches are complementary to our work. The challenge in implementing our approach for C and C++ is connecting the allocation type to memory, since *malloc* is untyped. Their static analysis of ownership types could provide similar information to explicit types used in Java.

The closest related work is Leakbot which combines offline analysis with online diagnosis to find data structures with memory leaks [18, 24, 26]. Leakbot uses JVMPI to take heap snapshots offloaded to another processor for analysis (we call this *offline* analysis since it is not using the same resources as the program although it may occur concurrently with program execution). By offloading the expensive part of the analysis to another processor, Leakbot minimizes the impact on the application while maintaining detailed per-object information. It then relies on an additional processor to perform heap differencing across multiple copies of the heap–a memory overhead potentially 200% or more that is

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

proportional to the heap–and ranking which parts of the *object* graph may be leaking. Leakbot produces very detailed object-level statistics, which depend precision of the heap snapshots. Cork, on the other hand, summarizes object instances in a *CPFG* graph that preserves a subset object class information, which minimizes the memory overhead (less than 0.5%). Cork is thus time and space efficient enough to run continuously and concurrently with the application.

Several completely online instance-based approaches for finding memory leaks exist for C/C++ and Java. Qin et al. detect memory leaks in C/C++ by looking for objects that exceed their expected lifetimes [29]. They use special hardware to detect and eliminate false positives, which gives them low time overhead and greater accuracy, but space overhead grows proportionally to the number of objects.

Relying only on software, other online techniques detect memory leaks identifies *stale* objects as those that have not been accessed in a long time [10, 14]. Chilimbi and Hauswirth introduced this technique for C/C++ where they added per-object bookkeeping information to track stale objects [14]. Per-object bookkeeping information does not translate well to Java where even the smallest application creates millions of distinct objects, making per-object tracking too expensive in both space and time. Bond and McKinley address this expense by significantly reducing the space overhead of identifying stale objects as likely leaks by introducing a statistical approach for storing per-object information in a single bit [10]. Using this technique, combined with an offline processing step, they detect memory leaks by reporting allocation and last-use sites of stale objects. Although they achieve space efficiency, tracking per-object information adds overheads of 45% on average which they reduce to 14% with sampling losing accuracy. For finding memory leaks, differentiating in-use objects from those not-in-use adds additional information and is complimentary to finding heap growth. However, staleness can miss leaking objects. For example, when the program leaks objects in a hash table, the hash table eventually exceeds its size. Rehashing all the objects touches them and defeats staleness-based approaches. Cork finds the data structures and their allocation sites responsible for systematic heap growth, which will eventually crash the program.

3 An Example Memory Leak

Figure 2 shows a simple order processing system that includes a memory leak. *NewOrder* inserts new `Order` into the `allOrdersHT` hashmap and into the `newOrderQ`, as shown in Figure 2(a). In Figure 2(b), *ProcessOrders* processes the `newOrderQ` one order at a time. It removes each order from the `newOrderQ` and fills it. If the customer is a `Company` (subtype of `Order`), it then issues a bill, putting it on the `billingQ`, and ships the order to the customer. In Figure 2(c), when the customer sends a payment, *ProcccessBill* removes the order from the `billingQ` and the `allOrdersHT` hashmap. However, if the customer is a `Person` (subtype of `Order`), *ProcessOrders* calls *ProcessPayment* with the customer-provided payment information and ships the order. *ProcessOrders* should, but does not, remove the order from `allOrdersHT` which results in a memory leak. Figure 2(d) lists abbreviations and statistics for these classes. We use this memory leak as a running example throughout this paper.

4 Finding Leaks with Cork

This section overviews how Cork identifies candidate leaks by examining the objects in the heap, finding growth, and reporting the corresponding class for growing objects to the user along with their allocation site and the data structure which contains them. For clarity of exposition, we describe Cork in the context of a full-heap collector.

Copyright © 2009 John Wiley & Sons, Ltd.

```
1 NewOrder(Order n) {
2   int id = getOrderId();
3   allOrdersHM.add(id, n); // insert into HashMap
4   newOrderQ.add(n);       // insert into NewOrderQ
5 }
```

(a) Incoming order

```
1 ProcessOrders() {
2   while (! newOrderQ.isEmpty()) {
3     Order n = newOrderQ.getNext();
4     newOrderQ.remove(n);  // remove from NewOrderQ
5     FillOrder(n);
6     if (n.getCustomer() instanceof Company) {
7       IssueBill(n);      // insert into BillingQ
8       ShipOrder(n);
9     } else if (n.getCustomer() instanceof Person) {
10      ProcessPayment(n);
11      ShipOrder(n);
12      // A MEMORY LEAK!! -- not removed from HashMap
13 }}}
```

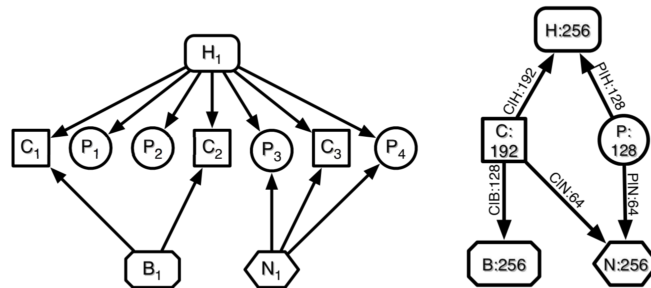
(b) Processing orders

```
1 ProcessBill(int orderId) {
2   Order n = allOrdersHT.get(orderId);
3   billingQ.remove(n);          // remove from Billing Q
4   allOrdersHM.remove(orderId); // remove from HashMap
5 }
```

(c) Process bills

Class	Variable	Symbol	Size in Bytes
HashMap	allOrdersHM	H	256
PriorityQueue	newOrderQ	N	256
Queue	billingQ	B	256
Company : Order	n	C	64
Person : Order	n	P	32

(d) Object statistics



(e) Object points-to graph (f) Class points-from graph

Figure 2. Order Processing System

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
1  void scanObject(TraceLocal trace,
2                  ObjectReference object) {
3    MMType type = ObjectModel.getObjectType(object);
4    type.incVolumeTraced(object);         // added
5    if (!type.isDelegated()) {
6      int references = type.getReferences(object);
7      for (int i = 0; i < references; i++) {
8        Address slot = type.getSlot(object, i);
9        type.pointsTo(object, slot);      // added
10       trace.traceObjectLocation(slot);
11   }} else {
12     Scanning.scanObject(trace, object);
13 }}
```

Figure 3. Object Scanning

4.1 Building the Class Points-From Graph

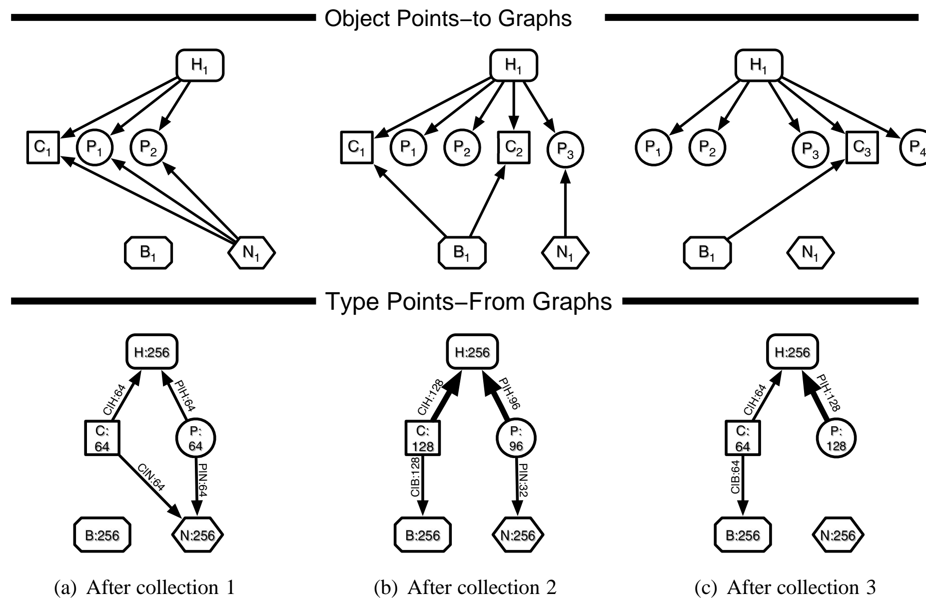
To detect growth, Cork summarizes the heap in a *class points-from graph* (*CPFG*). The *CPFG* consists of *class nodes* and *reference edges*. The class node, annotated with volume of instances of that class, represents the total volume of objects of class c (V_c). The reference edges are directed edges from class node c' to class node c and are annotated with the volume of objects of class c' that are referred to by objects of class c ($V_{c'|c}$). Looking at our example leak, Figure 2(e) shows a heap consisting of an object points-to graph, i.e., objects and their pointer relationships, for objects of classes H, C, P, B, and N from the example order-processing system. Each vertex represents a different object class in the heap and each arrow represents a reference between two objects classes. Figure 2(f) shows the class points-from graph that Cork computes.

To minimize the costs associated with building the *CPFG*, Cork piggybacks its construction on the scanning phase of garbage collection which detects live objects by starting with the roots (statics, stacks, and registers) and performing a transitive closure through all the live object references in the heap. For each live, reachable object o, Cork determines the object's class c_o and increments the corresponding class node by the object's size. Then for each reference from object o to object o', it increments the reference edge from c' to c by the size of o'. At the end of the collection, the *CPFG* completely summarizes the volumes of all classes and references that are live at the time of the collection.

Figure 3 shows the modified scanning code from MMTk, a memory-management toolkit which implements a number of garbage collection algorithms [6, 7]. Cork requires two simple additions that appear in lines 4 and 9. Assume *scanObject* is processing an object of class B that refers to an object of class C from Figure 2(e). It takes a reference and the object as parameters and finds the object class. Line 4 increments the volume of class B (V_B) for our example. Since the collector *scans*, i.e., detects liveness of, an object only once, Cork increments the total volume of this class only once per object instance. Next, *scanObject* determines if it needs to scan each referent of the object. As it iterates through the fields (slots), the added line 9 resolves the referent class of each outgoing reference ($B \rightarrow C$) and increments the volume of the appropriate edge ($B \leftarrow C$) in the graph ($V_{C|B}$). Thus, this step increments the edge volume for all references to an object, not just the first one. Because this step adds

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls



(a) After collection 1 (b) After collection 2 (c) After collection 3

Figure 4. Comparing Class Points-From Graphs to Find Heap Growth

an additional class look up for each reference, it also introduces the most overhead. Finally, *scanObject* enqueues those objects that have not yet been scanned in line 10. The additional work of the garbage collector depends on whether it is moving objects or not, and is orthogonal to Cork.

At the end of scanning, the *CPFG* completely summarizes the live objects in the heap. Figure 2(f) shows the *CPFG* for our example. Notice that the reference edges in the *CPFG* point in the opposite direction of the references in the heap. Also notice that, in the heap, objects of class C are referenced by H, B, and N represented by the outgoing reference edges of C in the *CPFG*. Since C has multiple references to it in the heap, the sum of the weights of its outgoing reference edges is greater than its class node weight. Cork uses volume rather than simple count to detect heap growth in order to capture not only when the number of instances of a class increase, but also when the number stays constant but the size of the the instances grow, as may be the case with arrays. Volume gives a heavier weight to larger classes which tend to make the heap grow faster than smaller classes. Cork differences the *CPFG* volumes from distinct collections to determine where growth is occurring in the heap.

4.2 Finding Heap Growth

At the end of each collection, Cork compares the *CPFG* for the current collection with data from previous collections. We define those class nodes whose volume increases across several collections as *candidates*. For each candidate, Cork follows growing reference edges through the *CPFG* to enumerate the class data structures involved in the growth.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls

For example, Figure 4 shows the *CPGF*s created during three collections of our example program. Figure 4(a) represents an initial state of the system after three orders arrive, but have not yet been processed. Figure 4(b) shows four orders processed: two billed and two completed. Notice that the program removes orders from individuals (P) from all the processing queues (B, N), but not from the hashmap (H) resulting in the memory leak shown in Figure 2(b). Comparing the *CPFG* from the first two collections shows both C and P objects are potentially growing (depicted with bold arrows). To be sure, we need more history. Figure 4(c) represents the state at the next collection, at which point it becomes clearer that the volume of P objects is monotonically increasing, whereas the volume of C objects is simply fluctuating. In practice, we find that class volume *jitters*, i.e., it fluctuates with high frequency. We say that a class whose volume monotonically increases shows *absolute growth* and one whose volume fluctuates but still increases shows *potential growth*. Cork detects both absolute and potential growth.

To detect systematic heap growth, Cork compares the *CPFG* from the current collection with data from previous collections and ranks each node according to how likely it is that a particular class is a candidate. Additionally it ranks edges in a similar fashion. We examine two different methods for ranking candidates: slope ranking and ratio ranking. Although slope ranking is more principled, ratio ranking is more space efficient and better captures slow growth when the volume of objects fluctuates a lot between collections.

4.2.1 Slope Ranking

Recall from Section 1, a positive slope in a heap-occupancy graph clearly indicates systematic heap growth. The Slope Ranking Technique (SRT) uses the insight that a growing class must contribute to the overall positive slope in the heap-occupancy graph. The more a class contributes, the higher the likelihood that it leaks. Thus, SRT ranks candidates according to the portion of the overall heap growth that each class contributes. In this configuration, Cork stores *CPFG*s from each collection and calculates the rate of change, or *slope*, between the current collection and previous collections. Slope for class c at collection i is calculated as $s_{c_i} = \delta v_c / \delta A$, where v_c is the volume of class c live in the heap and A is the total volume allocated. A class node is classified a candidate if it is growing more often than it is shrinking. SRT uses the percentage of the overall growth caused by the candidate leaking class c to calculate rank r_{c_i} for collection i such that $r_{c_i} = r_{c_{i-1}} + p_{c_i} * s_{c_i} / S$, where p is the number of phases (or collections) that c has been growing and S is the rate of change of the total heap ($S_i = \delta V_i / \delta A$). SRT reports classes with positives ranks ($r_i > 0$) as candidates.

While SRT is based on the principle of heap occupancy, it does not detect growth well. One reason problem is that SRT depends on the number of collections in the window. Larger windows are more likely to detect absolute growth. However, large windows are more likely to show false positives, since for example, the heap always grows at the beginning of any application run. If we choose smaller window sizes and accumulate rank over time, SRT misses slow growing leaks. Rank depends both on the slope of the class (s_{c_i}) and the overall slope (S_i), since heap fluctuations may cause either one to be negative, SRT is not accurate enough.

4.2.2 Ratio Ranking

The Ratio Ranking Technique (RRT) ranks class nodes according to the ratio of volumes between two consecutive collections, accumulates the differences (adding or subtracting, as appropriate) over all collections, and reports classes with ranks above a rank threshold ($r_c > R^c_{thres}$) as candidate leaks.

Copyright © 2009 John Wiley & Sons, Ltd.

```
1  public double calculateRatioRank() {
2    if (thisVolume > 0) {
3      if (thisVolume > maxVolumeTraced * (1 - decayFactor)) {
4        // growth phase detected
5        leakPhases++;
6        if (thisVolume > maxVolumeTraced)
7          maxVolumeTraced = thisVolume;
8        // calculate rank
9        if (thisVolume > lastVolume) {
10         rank += leakPhases * (thisVolume/lastVolume - 1);
11       } else {
12         rank -= leakPhases * (lastVolume/thisVolume - 1);
13       }
14     } else {
15       // non-growth phase detected
16       reset();
17     }
18     if (leakPhases >= MAX_LEAK_PHASES &&
19         rank > RANK_THRESHOLD) {
20       // report candidate
21       findSlice();
22   }
23   return rank;
24 }
```

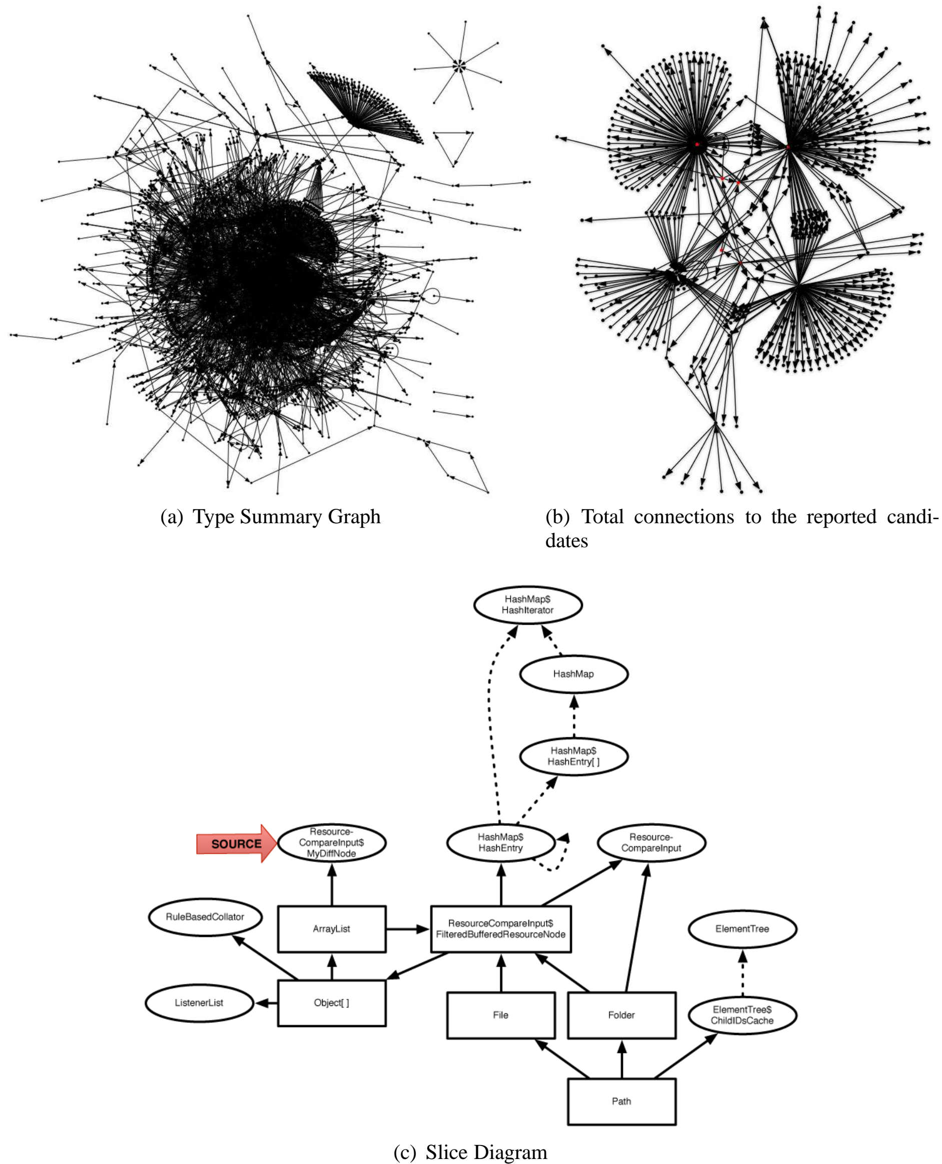
Figure 5. Ratio Ranking Technique Algorithm

Figure 5 shows the ranking algorithm for the RRT. Assume that `thisVolume` represents the volume from this collection and `lastVolume` the volume from the previous collection. RRT uses a `decayFactor`, f, where $0 < f < 1$ to adjust for jitter and detect potential growth. RRT considers only those class nodes whose volumes satisfy $V_{C_i} > V_{C_{i-1}} * (1 - f)$ (line 3) on consecutive collections as potential candidates. The decay factor keeps class nodes that shrink a little in this collection but which show potential growth. We find that the decay factor is increasingly important as the size of the leak decreases.

To rank class nodes, RRT uses the ratio of volumes between two consecutive collections Q such that $Q > 1$. Since $Q > 1$, then $Q - 1$ represents the percentage change in the volume between this collection and the previous collection. Each class node's rank r_c is calculated by accumulating the percent change multiplied by the number of phases (or collections) that c has been potentially growing such that absolute growth is rewarded and decay is penalized (lines 10 and 12). Higher ranks represent a higher likelihood that the corresponding volume of the class grows without bound. RRT reports candidates that show potential growth for at least two (2) phases and never reports a class the first time it appears in the graph (lines 18-22).

Cork reports candidate leaks and their ranks back to the user. Next, we describe how Cork correlates the candidate leaks back to the data structure that contains them and the allocation sites that allocated them.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls



(a) Type Summary Graph

(b) Total connections to the reported candidates

(c) Slice Diagram

Figure 6. Pruning the summary graph

4.3 Correlating to Data Structures and Allocation Sites

Reporting a low-level class such as `String` as a potential leak is not very useful. To demonstrate the complexity of the *CPFG*, Figure 6(a) shows one instance of the *CPFG* of Eclipse from DaCapo. A

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

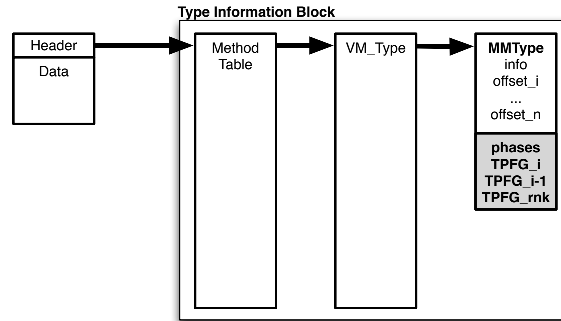


Figure 7. Type Information Block (TIB)

node exists for each class in the heap and an edge exists between any two classes if a corresponding reference exists in the heap. Notice that the complete *CPFG* is large and complex. Even if we refine this graph to simply include the candidate class nodes and all the classes that point to them, the graph still exhibits a fair amount of complexity, as shown in Figure 6(b). Cork automatically prunes the graph and isolates the growing data structure by constructing a *slice* through the *CPFG* that only includes the growing classes and references.

A *slice* through the *CPFG* is the set of all paths originating from class node c_0 such that the rank of each reference edge $r_{c_k \to c_{k+1}}$ on the path is positive. A slice defines the growth originating at class node c_0 following a sequence of class nodes $\{c_0, c_1, \ldots, c_n\}$ and a sequence of reference edges (c_k, c_{k+1}) where class node c_k points to c_{k+1} in the *CPFG*. Figure 6(c) shows the reported slice for Eclipse (Section 5.5.4 discusses this leak in more detail). The slice contains candidate leak classes and the dynamic data structure containing them. Cork also reports class allocation sites. However, unlike some more expensive techniques, it does not find the specific allocation site(s) responsible for the growth. Instead, it reports all allocation sites for each candidate class. As each allocation site is compiled, Cork assigns it a unique identifier, and constructs a map (a *SiteMap*) to it from the appropriate class. For each leaking class, Cork searches the map to find allocations sites for that class. For each class, the SiteMap includes the method and byte-code index or line for each allocation site.

4.4 Implementation Efficiency and Scalability

We implement several optimizations to make Cork's implementation scalable and efficient in both time and space. First, we limit the number of *CPFG*s that we keep. For SRT, the size of the window we use determines how many *CPGF*s are required. For RRT, only 3 *CPGF*s were required: $CPFG_i$, $CPFG_{i-1}$, and $CPFG_{rank}$, where $CPFG_{rank}$ is a graph which stores node and edge rank rather than volume. Cork piggybacks class nodes on the VM's global *type information block* (TIB). This structure or an equivalent is required for a correct implementation of managed languages such as Java or C#. Figure 7 shows the modified TIB from Jikes RVM. Notice that every live object of a class ($object_L$) points to the TIB corresponding to its class. The TIB consists of three different parts. The first is the method

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

table which stores pointers to code for method dispatch. The method table points to a corresponding `VM_Type` which stores field offsets and type information used by the VM for efficient type checking and is used by the compiler for generating correct code. Finally, the `VM_Type` points to a corresponding `MMType` used by the memory management system to do correct allocation and to identify references during garbage collection. Recall from Figure 3 that object scanning resolves the `MMType` of each object (line 3). Cork stores *CPFG* class node data for each *CPFG* in the corresponding `MMType`, adding only one word per stored *CPFG*. One additional word stores the number of consecutive phases that a class node shows potential growth. Thus, the class nodes scale with the type system of the VM.

While the number of reference edges in the *CPFG* are quadratic in theory, one class does not generally reference all other classes. Programs implement a much simpler class hierarchy, and we find reference edges are linear with respect to the class nodes. This observation motivates a simple edge implementation consisting of a pool of available edges. New edges are allocated only when the edge pool is empty. As *CPFG*s expire because they are in graphs outside the history window, we return the edges to the pool for reuse. New edges are added to the *CPFG* by removing them from the edge pool and adding them to the list of reference edges kept with node data. We encode a pointer to the edge list with the node data which eliminates the need for adding any extra words to the `MMType` structure. We further reduce the space required for reference edges by pruning those that do not grow.

4.5 Cork in Other Collectors

Cork performs its analysis on each full heap garbage collection in our implementation. The frequency could be increased by performing more full heap collections, or decreased by only performing it on a subset of full heap collections. We did not find either option necessary.

Since Cork's implementation piggybacks on live-heap scanning during garbage collection, it is compatible with any mark-sweep or copying collector, i.e., a *tracing* collector. Cork and thus can be added as described to any collector that does periodic whole-heap collections. To find leaks in our benchmarks, Cork needed approximately six full heap collections during which heap growth occurs. An incremental collector that never collects the entire heap may add Cork by defining intervals and combining statistics from multiple collections until the collector has considered the entire heap (i.e., an interval). Cork would then compute difference statistics between intervals to detect leaks.

4.6 Cork in Other Languages

Cork's heap summarization, the *CPFG*, relies on the garbage collector's ability to determine the class of an object. We exploit the object model of managed languages, such as Java and C#, by piggybacking on their required global class information to keep space overheads to a minimum. There are, however, other implementation options. For garbage-collected languages that lack user-defined class information, such as Standard ML, other mechanisms may be able to provide equivalent information. Previous work provides some suggestions for functional languages that tag objects [32, 33, 34]. For example, type-specific tags could be used to index into a hashmap for storing class nodes. Alternatively, objects could be tagged with allocation and context information allowing Cork to summarize the heap in an *allocation-site points-from graph*. These techniques, however, would come at a higher space and time overhead.

5 Results

This section presents overhead and qualitative results for Cork. Section 5.1 describes our methodology, Section 5.2 shows that Cork has a very small space overhead, and Section 5.3 shows that Cork adds

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

very little to total execution time, although it does slow down garbage collections. Section 5.4 shows that ratio ranking has few false positives and higher accuracy that slope ranking, and that furthermore, a variety of reasonable values for the decay factor and the rank threshold give similarly accurate results. Applying Cork to four commonly used benchmarks, Cork finds heap growth in four benchmarks: fop, jess, SPECjbb2000 and Eclipse, and the data structure reports enabled us to fix them very quickly, even though we were not previously familiar with these applications.

5.1 Methodology

We implement Cork in MMTk, a memory management toolkit in Jikes RVM version 2.3.7. MMTk implements a number of high-performance collectors [6, 7] and Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler [2, 4]. We use configurations that precompile as much as possible, including key libraries and the optimizing compiler (the *Fast* build-time configuration), and turn off assertion checking. For measurement purposes, we remove the nondeterministic behavior of the adaptive compilation system by applying replay compilation [22].

Eeckhout et al. [17] show that adaptive compilation in Jikes RVM obscures application behavior in performance measurements. Thus, for our overhead measurements, we factor out compilation using *replay compilation* [22]. Replay compilation deterministically applies the optimizing compiler to frequently executed methods chosen by the adaptive compiler in previous (offline) runs. We factor out the adaptive compiler by running each benchmark multiple times. The first run uses replay compilation to give a realistic mixture of optimized and unoptimized code. Then we turn off compilation and flush all compiler objects from the heap. During the second run, we measure and report application performance.

For performance results, we explore the time-space trade-off by executing each program on moderate to large heap sizes, ranging from 2.5X to 6X the smallest size possible for the execution of the program. We execute timing runs five times in each configuration and choose the best execution time (i.e., the one least disturbed by other effects in the system). We perform separate runs to gather overall and individual collection statistics. We perform all of our performance experiments on a 3.2GHz Intel Pentium 4 with hyper-threading enabled, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, and 1GB of main memory, running Linux 2.6.0.

For SPECjvm and DaCapo benchmarks, we use the standard large inputs. Since SPECjbb2000 measures throughput as operations per second for a duration of 2 minutes for an increasing number of warehouses (1 to 8) and each warehouse is strictly independent, we change the default behavior. To perform a performance-overhead comparison, we use pseudojbb, a variant of SPECjbb2000 that executes 10,000 transactions. For memory-leak analysis, we configure SPECjbb2000 to run only one warehouse for one hour. For Eclipse, we use the DaCapo benchmark for general statistics and performance-overhead comparisons and version 3.1.2 to reproduce a documented memory leak by repeatedly comparing two directory structures (Eclipse bug #115789).

5.2 Space Overhead

We evaluate our techniques using the SPECjvm [37], DaCapo b.050224 [9], SPECjbb2000 [38], and Eclipse [43]. Table I shows benchmark statistics including the total volume allocated (column 1) and number of full-heap collections in both a whole-heap (column 2) and a generational (column 3) collector in a heap that is 2.5X the minimum size in which the benchmark can run. Column 4 reports

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Benchmark	Alloc MB	# of colltn whl	gen	# of classes bm	+VM
Eclipse	3839	73	11	1773	3365
fop	137	9	0	700	2292
pmd	518	36	1	340	1932
ps	470	89	0	188	1780
javac	192	15	0	161	1753
jython	341	39	0	157	1749
jess	268	41	0	152	1744
antlr	793	119	6	112	1704
bloat	710	29	5	71	1663
jbb2000	**	**	**	71	1663
jack	279	47	0	61	1653
mtrt	142	17	0	37	1629
raytrace	135	20	0	36	1628
compress	106	6	3	16	1608
db	75	8	0	8	1600
Geomean	303	27	n/a	104	1813

Table I. Benchmark Characteristics. **Volumes for SPECjbb2000 depend on how long the warehouse to runs.

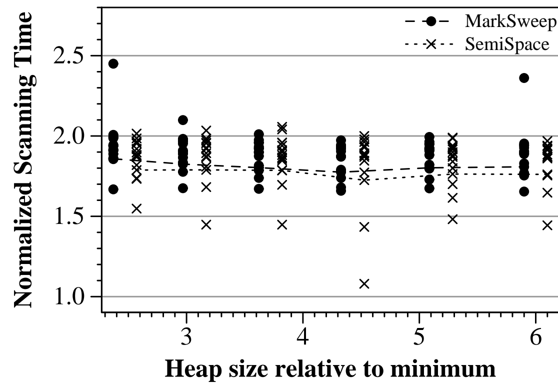
the number of classes (*bm*) in each benchmark. However since Jikes RVM is a Java-in-Java virtual machine, Cork analyzes the virtual machine along with the benchmark during every run. Thus column 5 (+*VM*) is the actual number of classes potentially analyzed at each collection.

The heuristics we introduced in Section 4.4 keep Cork space efficient. Table II(a) reports *CPFG* space overhead statistics. Columns one and two (*# of classes*) report the average and maximum number of classes in the heap during any particular garbage collection. We notice that an average of 44% of all classes used by programs are present in the heap at a time. This feature reduces the number of potential candidates that Cork must analyze.

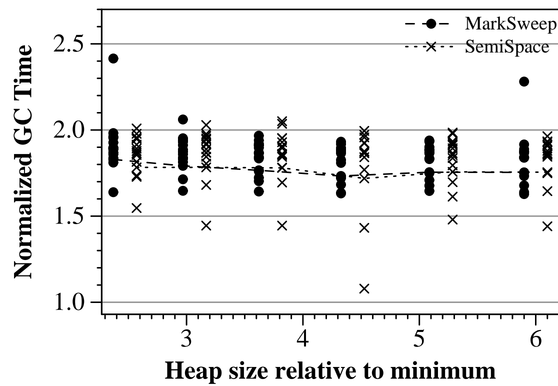
Table II(a) shows the average (column 3) and maximum (column 4) number of reference edges per class node in the *CPFG*. We find that most class nodes have a very small number of outgoing reference edges (2 on average). The more prolific a class is in the heap, the greater the number of reference edges in its node (up to 406). We measure the average and maximum number of reference edges in any *CPFG* (columns 5 and 6) and the percent of those our heuristics prune because their ranks drop below zero ($r_e < 0$) (column 7). These results demonstrate that the number of references edges is linear in the number of class nodes in practice.

Finally, Table II(b) shows the space requirements for the type information block before (*TIB*) and the overhead added by Cork (*TIB*+*Cork*). While Cork adds significantly to the TIB information, it adds only modestly to the overall heap (0.145% on average and never more than 0.5% as shown in column 5). For the longest-running and largest program, Eclipse, Cork has a tiny space overhead (0.004%).

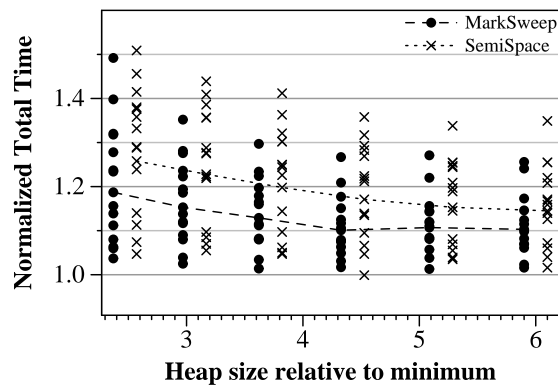
Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls



(a) Scan Time



(b) GC Time

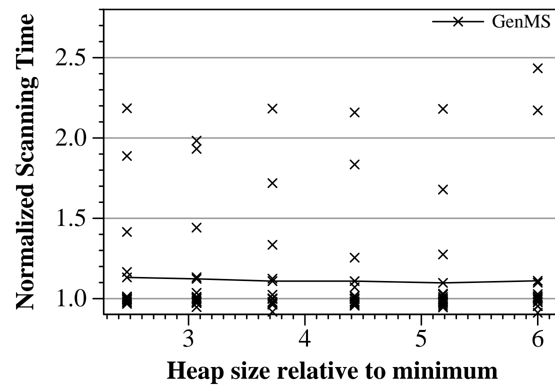


(c) Total Time

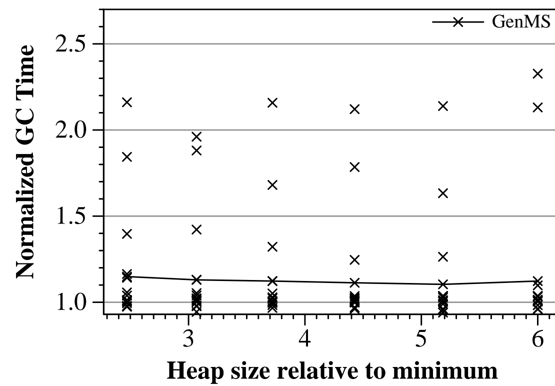
Figure 8. Geometric Mean Overhead Graphs over all benchmarks for whole-heap collector

Copyright © 2009 John Wiley & Sons, Ltd.

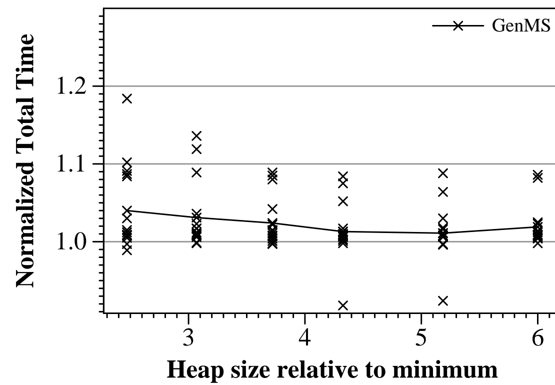
Prepared using speauth.cls



(a) Scan Time



(b) GC Time



(c) Total Time

Figure 9. Geometric Mean Overhead Graphs over all benchmarks for generational collector

Copyright © 2009 John Wiley & Sons, Ltd.

	(a) Class Points-From Statistics								(b) Space Overhead				
	# of classes		# edges per class		# edges per TPFG		% pru-	TIB		TIB+Cork		
Benchmark	avg	max	avg	max	avg	max	ned	MB	%H	MB	%H	Diff
Eclipse	667	775	2	203	4090	7585	42.2	0.53	0.011	0.70	0.015	0.167
fop	423	435	3	406	1559	2623	45.2	0.36	0.160	0.55	0.655	0.495
pmd	360	415	3	121	967	1297	66.0	0.30	0.031	0.44	0.186	0.155
ps	314	317	2	93	813	824	66.3	0.28	0.029	0.39	0.082	0.053
javac	347	378	3	99	1118	2126	45.8	0.28	0.071	0.43	0.222	0.151
jython	351	368	2	114	928	940	66.2	0.28	0.041	0.39	0.112	0.071
jess	318	319	2	89	844	861	66.0	0.27	0.049	0.38	0.143	0.094
antlr	320	356	2	123	860	1398	55.8	0.27	0.016	0.39	0.282	0.266
bloat	345	347	2	101	892	1329	50.6	0.26	0.017	0.38	0.064	0.047
jbb2000	318	319	2	110	904	1122	59.0	0.26	**	0.38	**	**
jack	309	318	2	107	838	878	66.2	0.26	0.042	0.37	0.131	0.089
mtrt	307	307	2	91	820	1047	57.5	0.26	0.081	0.37	0.258	0.177
raytrace	305	306	2	91	814	1074	56.1	0.26	0.085	0.37	0.272	0.187
compress	286	288	2	89	763	898	60.9	0.25	0.105	0.36	0.336	0.231
db	289	289	2	91	773	787	66.1	0.25	0.160	0.35	0.467	0.307
Geomean	342	357	2	116	1000	1303	57.4	0.29	0.048	0.41	0.168	0.145

Table II. Cork Statistics. **Volumes for SPECjbb2000 depend on how long the warehouse runs.

5.3 Performance Overhead Results

Cork's time overhead comes from constructing the *CPFG* during scanning and from differencing between *CPGF*s to find growth at the end of each collection phase. Figures 8 and 9 graphs the normalized geometric mean over all benchmarks to show overhead in scan time, collector (GC) time, and total time. In each graph, the y-axis represents time normalized to the unmodified Jikes RVM using the same collector, and the x-axis graphs heap size relative to the minimum size each benchmark can run in a mark-sweep collector. Each x represents one program.

For mark-sweep (MarkSweep) and copying (SemiSpace) whole-heap collectors, Figure 8 shows that the scan time overhead is 80.8% to 85.5% and 76.1% to 78.8%; collector time is 75.4% to 82.9%; and total time is 10.3% to 25.8% respectively. These overheads represents the worst case since Cork analyzes the heap at every collection. Whole-heap collector overheads can easily be reduced by analyzing the heap every nth collection or by using a generational collector that performs whole-heap collections less frequently. In fact, this configuration may find leaks faster. Similarly, including Cork in a high-performing generation collector with many less full-heap collections significantly reduces these overheads by performing Cork's analysis less frequently. Figure 9 shows Cork's average overhead in a generational collector to be 11.1% to 13.2% for scan time; 12.3% to 14.9% for collector time; and 1.9% to 4.0% for total time. Individual overhead results range higher, but Cork's average overhead is low enough to consider using it online in a production system.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

Benchmark	(a) SRT	(b) RRT Decay Factor 0%	5%	10%	15%	20%	25%	(c) RRT Rank Thres 0	50	100	200
Eclipse bug #115789	6	0	6	6	**6**	6	6	12	6	**6**	6
fop	2	2	2	2	**2**	2	2	35	2	**2**	1
pmd	0	0	0	0	**0**	0	0	11	2	**0**	0
ps	0	0	0	0	**0**	0	0	3	0	**0**	0
javac	2	0	0	0	**0**	0	0	71	2	**0**	0
jython	0	0	0	0	**0**	0	1	3	0	**0**	0
jess	2	0	1	1	**1**	1	2	9	1	**1**	1
antlr	0	0	0	0	**0**	0	0	9	0	**0**	0
bloat	3	0	0	0	**0**	0	0	33	0	**0**	0
jbb2000	1	0	4	4	**4**	4	4	10	6	**4**	4
jack	0	0	0	0	**0**	0	0	9	0	**0**	0
mtrt	0	0	0	0	**0**	0	0	3	2	**0**	0
raytrace	0	0	0	0	**0**	0	0	4	0	**0**	0
compress	0	0	0	0	**0**	0	0	4	0	**0**	0
db	0	0	0	0	**0**	0	0	2	0	**0**	0

Table III. Number of classes reported in at least 25% of garbage collection reports: (a) From Slope Ranking Technique. (b) Varying the *decay factor* from Ratio Ranking Technique ($R^t_{thres} = 100$). We choose a decay factor $f = 15\%$. (c) Varying the *rank threshold* from Ratio Ranking Technique ($f = 15\%$). We choose rank threshold $R^t_{thres} = 100$.

5.4 Achieving Accuracy

Cork's accuracy depends on its ability rank and report growing classes. Table III(a) shows the number of candidates that are reported using slope ratio (SRT). While it accurately identifies growth in fop, jess, jbb2000, and Eclipse bug #115789, it also falsely identifies heap growth in javac and bloat, programs that do not display systematic heap growth. This result is mainly due to very erratic growth patterns in both programs. Ratio ranking (RRT) offers a more robust heuristics for ranking classes. By increasing the rank when the class grows and decreasing it when it shrinks, RRT more accurately captures growth across many collections without depending upon window size.

For the RRT, we experiment with different sensitivities for both the decay factor f and the rank threshold R_{thres}. Table III(b) shows how changing the decay factor changes the number of reported classes. We find that the detection of growing classes is not sensitive to changes in the decay factor ranging from 5 to 20%. We choose a moderate decay factor ($f = 15\%$) for which Cork accurately identifies the only growing data structures in our benchmarks without any false positives. Table III(c) shows how increasing the rank threshold eliminates false positives from our reports. Additionally we experiment with different rank thresholds and find that a moderate rank threshold ($R_{thres} = 100$) is sufficient to eliminate any false positives. We discuss the differences in the number of reported classes between SRT and RRT as we discuss each benchmark in the next section

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

5.5 Finding and Fixing Leaks

Cork identifies heap growth in four of our benchmarks: fop, jess, SPECjbb2000, and Eclipse. Each section first describes the benchmark, demonstrates how Cork found the growing class and data structure, and concludes with an analysis of the growth.

5.5.1 fop

The program fop (Formatting Objects Processor) is from the DaCapo benchmark suite. It uses the standard XSL-FO file format as input, lays the contents out into pages, and then renders it to PDF. Converting a 352KB XSL-FO file into a 128KB PDF generates the heap occupancy graph in Figure 10, which clearly demonstrates an overall monotonic heap growth. While we limit the heap size to 15MB, this size is simply a function of the input size. Given any heap size, we can give fop an input that will cause the heap to expand and crash. fop shows aggressive heap growth (7.5% compared to the 0.19% in SPECjbb2000).

Cork analyzes fop and Figure 10(a) shows the RRT reports. Both SRT and RRT report `ArrayList` and `Object[]` as candidates for growth. Since `ArrayList` is implemented as `Object[]`, we focus just on `ArrayList` for our analysis. We begin our exploration by examining the slices of the *CPFG* to determine what is keeping the `ArrayList` alive. Figure 10(a) shows part of the slice for `ArrayList` . It shows that `ArrayLists` are nested in a data structure. Finally, Cork lists the allocation sites for all the class giving the user a starting point for debugging. Because the allocation sites are numerous, it is not useful to explore `ArrayList`. We go to secondary allocations sites: `WordArea` and `LineArea.`

Next we explore fop's implementation. fop performs two passes over a single complex data structure built with `ArrayList`: the first pass builds the formatting object tree where `ArrayList` contains different formatting object which themselves can contain one or more `ArrayList.` Once fop encounters an end of page sequence, it begins rendering during a second pass over the data structure it built during parsing. Thus, rendering uses the entire data structure. While our analysis accurately pinpoints the source of the growth, fop does not have a memory leak because it uses the entire heap. The developers of fop agree with this analysis, that the heap growth that fop experiences is partly inherent to the formatting process and partly caused by poor implementation choices [3]. Cork identifies this problem.

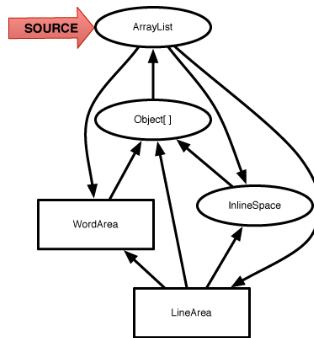
5.5.2 jess

From the SPECjvm benchmark suite, jess is a Java Expert Shell System based on NASA's CLIPS. It grows of 45KB every 64MB. In an expert system, the input is a set of facts and a set of rules. Each fact represents an existing relationship and each rule a legal way of manipulating facts. The expert system then reasons by using rules to *assert* new facts and *retrace* existing facts. As each part of a rule *matches* existing facts, the rule *fires* creating new facts and removing the rule from the set of activated rules. The system continues until the set of activated rules becomes empty.

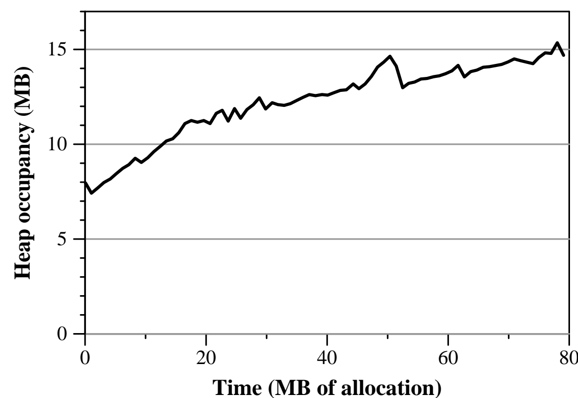
RRT reports `Value` as the overwhelmingly growing class. The slice of the *CPFG* is diagrammed in Figure 11(a) where the square node represents the reported class. Correlating it to the implementation, jess compiles all the rules into a single set of nodes. Fact assertion or retraction is then turned into a *token*, which is fed to the input nodes of the network. Then the nodes may pass the token on to its children or filter it out. As tokens are propagated through the network, rules create new facts. Each new fact is stored in a `Value` in a `ValueVector` implemented as `Value[]`. `ValueVector` is stored in a `ValueVector[]` in a `Token`. A global `TokenVector` implemented as `Token[]`, stores the

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls



(a) Slice Diagram



(b) Heap Occupancy Graphs for fop

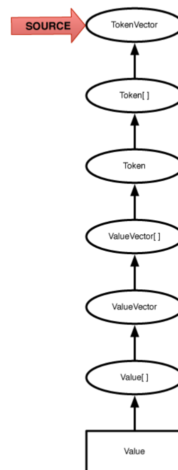
Figure 10. Fixing jess

tokens in the system. Interestingly, SRT reported only two of the classes both of which are in the slice reported by RRT: `Value` and `Value[]`. These facts are part of the input.

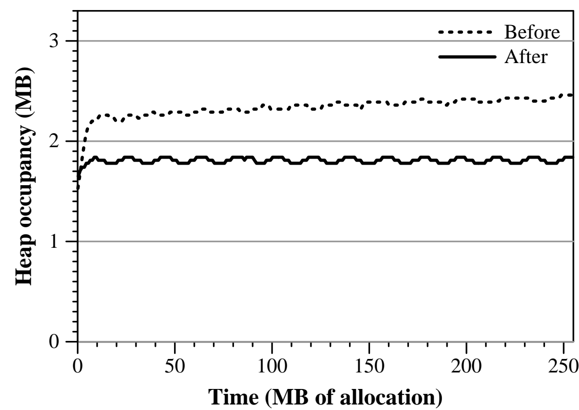
Examining the input for jess, we find the benchmark iterates over the same problem several times. The developer made it artificially more complex by introducing distinct facts in the input file representing the same information for each iteration. Thus, with each iteration, the number of facts to test increases which triggers more allocation. This complexity is documented in the input file. In order to remove the memory leak, we eliminated the artificial complexity from the input file. Figure 11(b) shows both the original heap occupancy graph and the resulting heap occupancy graph. The heap growth, and thus the memory leak, is gone.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls



(a) Slice Diagram



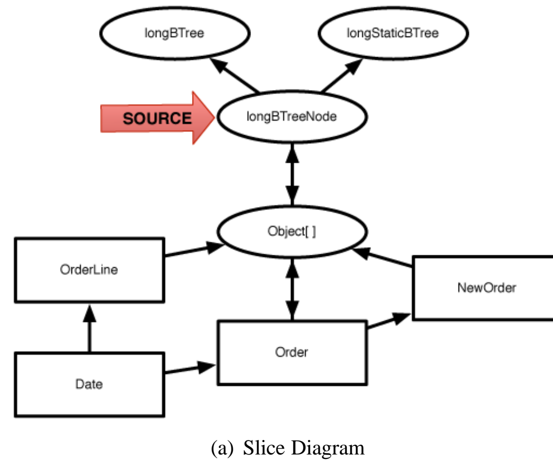
(b) Heap Occupancy Graphs for jess

Figure 11. Fixing jess

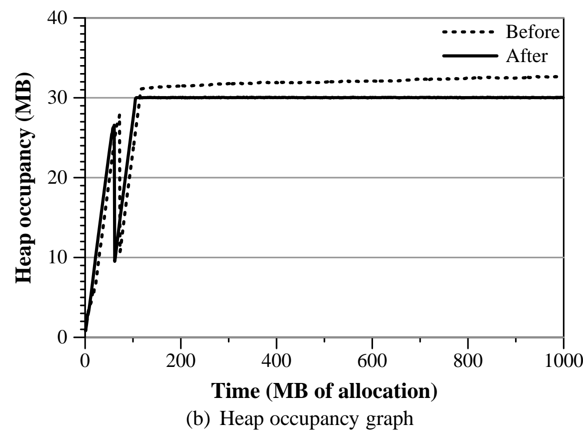
5.5.3 SPECjbb2000

The SPECjbb2000 benchmark models a wholesale company with several warehouses (or districts). Each warehouse has one terminal where customers can generate requests: e.g., place new orders or request the status of an existing order. The warehouse executes operations in sequence, with each operation selected from the list of operations using a probability distribution. It implements this system entirely in software using Java classes for database tables and Java objects for data records (roughly 25MB of data). The objects are stored in memory using `Bret` and other data structures.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls



(a) Slice Diagram



(b) Heap occupancy graph

Figure 12. Fixing SPECjbb2000

RRT analysis reports four candidates: `Order`, `Date`, `NewOrder`, and `OrderLine`. The rank of the four corresponding class nodes oscillates between collections making it difficult to determine their relative importance. Examining the slices of the four reported class nodes reveals the reason. There is an interrelationship between all of the candidates and if one is leaking then the rest are as well. The top of Figure 12(a) shows the Cork slice report where the shaded nodes are growing. Notice that despite the prolific use of `Object[]` in SPECjbb2000, its class node volume jitters to such a degree that it never shows sufficient growth to be reported as leaking. Since the slice includes all class nodes with $r_t > R^t_{thres}$ and reference edges with $r_e > 0$, the slice sees beyond the `Object[]` to the containing data structures.

Copyright © 2009 John Wiley & Sons, Ltd.

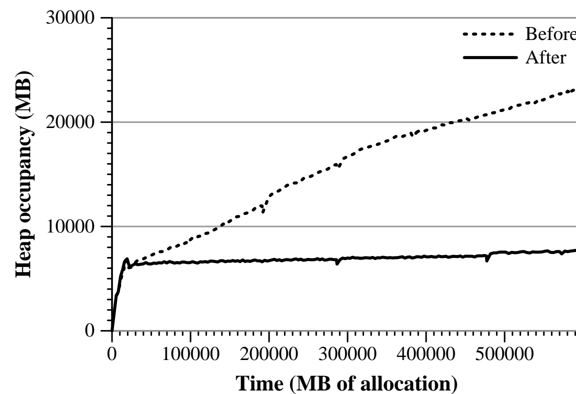


Figure 13. Heap occupancy graph: fixing Eclipse bug #115789

We correlate Cork's results with SPECjbb2000's implementation. We find that orders are placed in an `orderTable`, implemented as a `BTree`, when they are created. When they are completed during a *DeliveryTransaction*, they are not properly removed from the `orderTable`. By adding code to remove the orders from the `orderTable`, we eliminate this memory leak. Figure 12(b) shows the heap occupancy, before and after the bug fix, running SPECjbb2000 with one warehouse for one hour. It took us only a day to find and fix this bug in this large program that we had never studied previously.

5.5.4 Eclipse

Eclipse is a widely-used integrated development environment (IDE) written in Java [43]. Eclipse is big, complex, and open-source. Eclipse bug #115789 documents an unresolved memory leak in the Eclipse bug repository from September 2005. We manually compared the contents of two directory structures multiple times to cause the bug to be triggered at a much higher frequency that would be triggered during regular use. In this way we could isolate this bug from other potential bugs in the system. Both RRT and SRT reported six candidates: `File, Folder, Path, ArrayList, Object[],` and `ResourceCompareInput$FilteredBufferedResourceNode.` Figure 6(c) shows the growth slice for the candidates, the close interrelationship between them, and several possible roots of the heap growth.

Correlating Cork's results with the Eclipse implementation showed that upon completion, the differences between the two directory structures are displayed in the `Compare- EditorInput` which is a dialog that is added to the `NavigationHistory`. Further scrutiny showed that the `NavigationHistoryEntry` managed by a reference counting mechanism was to blame. When a dialog was closed, the `NavigationHistoryEntry` reference count was not decremented correctly resulting in the dialog never being removed from the `NavigationHistory`. The `CompareEditorInput` stores the differences of the two directory structures in a linked list of `ResourceCompareInput%MyDiffNode`. Figure 13(b) shows the heap occupancy graphs before

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

and after fixing the memory leak. This bug took us about three and a half days to fix, the longest of any of our benchmarks, due to the size and complexity of Eclipse and our lack of expertise on the implementation details.

6 Conclusion

This paper introduces a novel and efficient way to summarize the heap to identify types of objects which cause systematic heap growth, the data structures which contains them, and the allocation site(s) which allocate them. We implement this approach in Cork, a tool that identifies growth in the Java heap and reports slices of a summarizing class points-from graph. Cork calculates this information by piggybacking on full-heap garbage collections. We show that Cork adds only 2.3% to total time on moderate to large heaps in a generational collector. Cork precisely identifies data structures with unbounded heap growth in four popular benchmarks: fop, jess, jbb2000, and Eclipse and we use its reports to analyze and eliminate memory leaks. We were able to fix these leaks from the reports, even though we had no prior experience with the code of these applications. Cork is highly-accurate, low-overhead, scalable, and is the first tool to find memory leaks with low enough overhead to consider using in production VM deployments.

ACKNOWLEDGEMENTS

We would like to thank Steve Blackburn, Robin Garner, Xianglong Huang, and Jennifer Sartor for their help, input, and discussions on the preliminary versions of this work. Additional thanks go to Ricardo Morin and Elena Ilyina (Intel Corporation), and Adam Adamson (IBM) for their assistance with confirming the memory leak in jbb2000, and Mike Bond for his assistance with Eclipse.

REFERENCES

1. E. E. Aftandilian and S. Z. Samuel Z Guyer. Gc assertions: Using the garbage collector to check heap properties. In *ACM Conference on Programming Language Design and Implementation*, pages 235–244, Dublin, Ireland, June 2009.
2. B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, Colorado, USA, November 1999.
3. The Apache XML Project. *Using FOP Documentation*, release 0.20.5 edition, 2005.
4. M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65, Minneapolis, Minnesota, USA, October 2000.
5. M. Arnold, M. Vechev, and E. Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 143–162, Nashville, Tennessee, USA, October 2008.
6. S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: The Performance Impact on Garbage Collection. In *International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, New York, New York, USA, June 2004.
7. S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with JMTk. In *International Conference on Software Engineering*, pages 137–146, Scotland, United Kingdom, May 2004. IEEE Computer Society.
8. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. Technical report, October 2006. http://www.dacapobench.org.

Copyright © 2009 John Wiley & Sons, Ltd.

9. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, October 2006. http://www.dacapobench.org.
10. M. D. Bond and K. S. McKinley. Bell: Bit-Encoding Online Memory Leak Detection. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, San Jose, California, USA, October 2006.
11. M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 109–126, Nashville, Tennessee, USA, October 2008.
12. M. D. Bond and K. S. McKinley. Leak pruning. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, pages 277–288, Washington, DC, USA, March 2009.
13. J. Campan and E. Muller. Performance Tuning Essential for J2SE and J2EE: Minimize Memory Leaks with Borland Optimizeit Suite. White Paper, Borland Software Corporation, March 2002.
14. T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection using Adaptive Statistical Profiling. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, pages 156–164, Boston, Massachusetts, USA, October 2004.
15. W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution Patterns in Object-Oriented Visualization. In *USENIX Conference on Object-Oriented Technologies and Systems*, pages 219–234, Santa Fe, New Mexico, USA, April 1998.
16. W. De Pauw and G. Sevitsky. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency: Practice and Experience*, 12(12):1431–1454, November 2000.
17. L. Eeckhout, A. Georges, and K. De Bosschere. How Java Programs Interact with Virtual Machines at the Microarchitecture Level. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, California, USA, October 2003.
18. S. C. Gupta and R. Palanki. Java Memory Leaks – Catch Me If You Can: Detecting Java Leaks using IBM Rational Application Developer 6.0. Technical report, IBM, August 2005.
19. R. Hastings and B. Joyce. Purify: A Tool for Detecting Memory Leaks and Access Errors in C and C++ Programs. In *USENIX Conference*, pages 125–138, Berkley, California, USA, January 1992.
20. D. L. Heine and M. S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *ACM Conference on Programming Language Design and Implementation*, pages 168–181, San Diego, California, USA, June 2003.
21. M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stevanović. Error Free Garbage Collection Traces: How to Cheat and Not Get Caught. In *ACM Conference on Measurement & Modeling Computer Systems*, pages 140–151, Marina Del Rey, California, USA, June 2002.
22. X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 69–80, Vancouver, BC, Canada, October 2004.
23. M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Java. In *ACM Symposium on the Principles of Programming Languages*, pages 31–38, Nice, France, January 2007.
24. N. Mitchell, May 2006. Personal communication.
25. N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 245–260, Montreal, Quebec, Canada, October 2007.
26. N. Mitchell and G. Sevitzky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377, Darmstadt, Germany, July 2003. Springer-Verlag.
27. H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *International Symposium on Memory Management*, pages 15–30, Montreal, Quebec, Canada, June 2007.
28. G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *ACM Conference on Programming Language Design and Implementation*, pages 397–407, Dublin, Ireland, June 1009.
29. F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Symposium on High Performance Computer Architecture*, pages 291–302, Cambridge, Massachusetts, USA, February 2002. IEEE Computer Society.
30. QuestSoftware. JProbe Memory Debugger: Eliminate Memory Leaks and Excessive Garbage Collection. http://www.quest.com/jprobe/profiler.asp.
31. D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. pages 194–203, Atlanta, Georgia, USA, November 2007.
32. N. Röjemo. Generational Garbage Collection without Temporary Space Leaks. In *International Workshop on Memory Management*, 1995.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls

33. N. Röjemo and C. Runciman. Lag, Drag, Void and Use – Heap Profiling and Space-Efficient Compilation Revised. In *ACM International Conference on Functional Programming*, pages 34–41, Philadelphia, Pennsylvania, USA, May 1996.
34. C. Runciman and N. Röjemo. Heap Profiling for Space Efficiency. In E. M. J. Launchbury and T. Sheard, editors, *Advanced Functional Programming, Second International School-Tutorial Text*, pages 159–183, London, United Kingdome, August 1996. Springer-Verlag.
35. M. Serrano and H.-J. Boehm. Understanding Memory Allocation of Scheme Programs. In *ACM International Conference on Functional Programming*, pages 245–256, Montréal, Québec, Canada, September 2000.
36. R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic Removal of Array Memory Leaks in Java. In *International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 50–66, London, United Kingdom, 2000. Springer-Verlag.
37. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.
38. Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.
39. D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, Ameherst, Massachusetts, USA, 1999.
40. Sun Microsystems. Heap Analysis Tool. https://hat.dev.java.net/.
41. Sun Microsystems. HPROF Profiler Agent. http://java.sun.com/j2se/ 1.4.2/docs/guide/jvmpi/jvmpi.html.
42. Y. Tang, Q. Gao, and F. Qin. Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *USENIX Conference*, pages 307–320, Boston, Massachusetts, USA, July 2008.
43. The Eclipse Foundation. Eclipse Homepage. http://www.eclipse.org.
44. G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *International Conference on Software Engineering*, pages 151–160, Leipzig, Germany, May 2008.

99

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using speauth.cls