Cork: Dynamic Memory Leak Detection for Java *

Maria Jump and Kathryn S. McKinley

Technical Report TR-06-07
Department of Computer Sciences
The University of Texas at Austin
Austin, TX, 78712, USA
{mjump,mckinley}@cs.utexas.edu

Abstract

Despite all the benefits of garbage collection, memory leaks remain a problem for Java programs. A *memory leak* in Java occurs when a program inadvertently maintains references to objects that it no longer needs, preventing the garbage collector from reclaiming space. At best, leaks degrade performance. At worst, they cause programs to run out of memory and crash. Small continuous leaks in long-running programs are notoriously hard to find and can crash the program only after days or weeks of execution.

We introduce Cork, a low-overhead, accurate technique for detecting memory leaks in Java programs. Cork identifies overall monotonic heap growth by piggybacking on the garbage collector. On each full-heap collection, Cork builds a summary *type points-to* graph annotated with type volumes. Cork identifies potentially leaking types that grow over multiple collections. Cork reports the *slice* in the type points-to graph that is growing (i.e., the data structure that points to the leaking type). We implement Cork in MMTk for Jikes RVM, where it adds an average overhead of 2.4% for moderate heap sizes and 1.7% for large heap sizes to SPECjvm and DaCapo benchmarks using a generational mark-sweep collector. Cork exactly identifies a single growing data structure in each of three popular benchmarks (fop, _202_jess, and SPECjbb2000). Due to the precision of Cork's report, we eliminated these leaks in _202_jess and SPECjbb2000, whereas their developers had not previously done so. Cork is the first tool to find leaks in Java with low enough overhead to consider using online.

1. Introduction

Memory-related bugs are a substantial source of errors, but are especially problematic for languages with explicit memory management such as C and C++. For these languages, memory-related errors include (1) dereferencing a pointer to memory that the program previously freed (dangling pointer), (2) losing a pointer to an object that the program neglects to free (lost pointer), and (3) keeping a pointer to an object the program will never use again (unnecessary reference).

Garbage-collected languages solve the first two memory errors, but not the last. The garbage collector eliminates the dangling pointer error since a pointer to an object prevents the collector from reclaiming it. Additionally, the collector eliminates those memory leaks caused lost pointers since it reclaims objects that do not have pointers to them. Unfortunately, garbage collection is conservative and therefore cannot detect, much less reclaim, memory referred to by unnecessary references. Thus, a *memory leak* in a garbage-collected language occurs when a program inadvertently maintains references to objects that it no longer needs, preventing the garbage collector from reclaiming space.

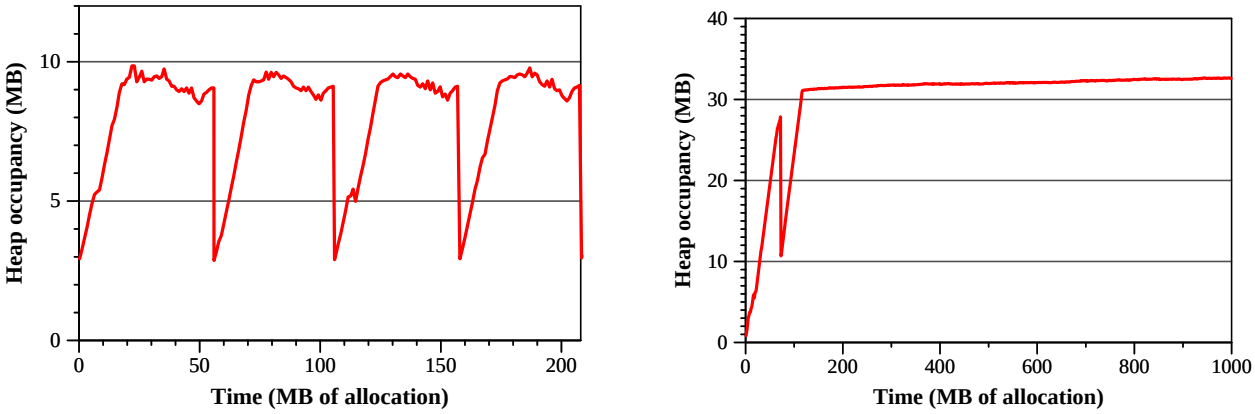
In the best case, unnecessary references to individual objects simply degrade program performance by increasing its memory requirements and consequently the collector workload. In the worst case, unnecessary references refer to a growing data structure, parts of which are no longer in use. These types of leaks can eventually cause the program to run out of memory and crash. In long-running programs, such as server applications, small leaks can take days or weeks to manifest making these bugs notoriously difficult to find.

Heap-occupancy graphs [18, 24] reveal the underlying problem of systematic heap growth, but not the solution. A heap occupancy graph plots the total heap occupancy (y-axis) over time (x-axis) measured in allocation by collecting the entire heap very frequently (every 10K of allocation in our graphs). Figure 1 shows the heap occupancy graphs of _213_javac from SPECjvm and SPECjbb2000. The graph for _213_javac shows four program allocation phases that reach the same general peaks which indicates _213_javac uses about the same maximum amount of memory in each phase and no phase leaks memory to the next. There is no leak. On the other hand, SPECjbb2000 running one warehouse for long periods of time shows memory requirements continue to grow until the end of execution. Allowed to run for days, it would run out of memory and crash. There is a leak. Although these graphs reveal potential leaks, they do not pinpoint the source of the leak.

Previous approaches to finding memory leaks use heap diagnosis tools that rely on a combination of heap differencing [11, 12, 13, 19, 20] and allocation and/or fine-grain usage tracking [9, 10, 15, 16, 21, 25, 26] which makes them very expensive. These techniques tend to yield large amounts of low-level details about individual objects that require a lot of time and expertise to interpret.

To address these shortcomings, this paper introduces *Cork*, a low-overhead, accurate technique for detecting potential memory leaks in Java programs. Cork identifies overall monotonic heap growth and reports the data structure(s) that generates it to the user. Cork piggybacks on full-heap garbage collection to compute this information. As the garbage collector scans the heap, Cork builds a

* This work is supported by NSF CCR-0311829, NSF ITR CCR-0085792, NSF CCR-0311829, NSF CISE infrastructure grant EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.



(a) _213_javac

(b) SPECjbb2000 running one warehouse for 1 hour

Figure 1. Example Heap Occupancy Graphs

summary *type points-to* graph. The nodes of the graph represent the volume of live objects of each type in the heap. The edges represent the points-to relationship between two types and are weighted by volume. At the end of each collection, the type points-to graph completely summarizes the live-object points-to graph. By comparing type points-to history over a series of collections, Cork detects systematic heap growth, but does not require growth after every collection. Since the leaking object type could be prolific (e.g., a *String*), Cork computes the parts of the type points-to graph that cause the leak (e.g., the data structure referring to *String*), and reports it to the user. Cork also reports the allocation points for these types. Cork does not analyze whether the program is in fact using the entire data structure.

We implement Cork in MMTk [6, 7], a memory-management toolkit in Jikes RVM [2, 4], a Java-in-Java virtual machine. Cork operates only during full-heap collections. Thus, we first use Cork to detect memory leaks in short-running programs using a whole-heap collector with frequent collections (every 4MB of allocation). In this setting, Cork's average overhead varies between 35% and 45.5% depending on the heap size.

For SPECjvm and DaCapo benchmarks, we show configurations where Cork reports all growing data structures and does not report any false positives. For all configurations, Cork reports orders of magnitude fewer types than previous work [19, 20, 25, 26].

Cork identifies unbounded heap growth in three commonly used benchmarks: fop, _202_jess, and SPECjbb2000. fop grows 4.8MB every 64MB of allocation but actually uses the entire growing data structure. Since Cork does not sample or track individual accesses to heap objects [10], it finds these types of false positives. However, this heap growth is cause for concern, and is a limitation of the formatting process and the implementation choices in fop.

On the other hand, _202_jess and SPECjbb2000 both leak memory: 45KB every 64MB and 127KB every 64MB respectively. Cork's output precisely pinpoints the single data structure responsible for the growth. For _202_jess and SPECjbb2000, this precision pays off: we correct the leaks and show that the resulting heap occupancy graphs are flat. The developers of these programs knew of these leaks, but had not fixed them. In the case of SPECjbb2000, the leak remained elusive for years, yet we found it and fixed it in a day.

Cork reports leaks that manifest during an instrumented execution. To demonstrate that Cork could be used online in long-running programs, we incorporate it in a generational collector where it executes during infrequent full-heap collections which lowers its overhead to between 1.7% and 2.4% on average for moderate to large heaps, but grows as high as 12.9% for very tight heaps. Less frequent full-heap collections means that Cork needs to run longer before it has enough fodder to accurately identify memory leaks. However, this low overhead combined with Java's dynamic class loading feature make it possible to consider using Cork online to find leaks, and then eliminate them by loading the corrected class without restarting.

Cork substantially increases the accuracy and reduces the cost of finding and fixing memory leaks due to unbounded data structure growth as compared to prior work. In practice, Cork reports leaks precisely without any false positives, and thus significantly improves accuracy over prior work. With modest extensions, this approach can be adapted by any garbage collected system, including Boehm-Demer-Weiser's conservative collector [8].

2. Related Work

A number of offline diagnosis tools help the user look inside the heap to determine the root cause of a memory leak. They rely on a combination of heap differencing [11, 12, 13, 20] and allocation and/or usage tracking at a fine level of detail [9, 15, 16, 21, 25, 26]. The drawbacks of these approaches are that they are very expensive and tend to yield large amounts of low-level details about individual objects that require a lot of time and expertise to interpret.

An example online instance-based approach for C and C++ is work by Chilimbi and Hauswirth [10]. They present an online, low-overhead memory leak detector for C. It uses per-instance bookkeeping information to identify *stale* objects as those which have not been accessed in a long time and thus may be leaks. Because C programs often allocate big hunks of memory and then divide them up [5], their approach often attains low overhead. In Java, even the smallest application creates millions of distinct objects, making per-instance object tracking too expensive. However, their sampling mechanism differentiates in-use objects from not-in-use, which adds accuracy to their reports.

An example of a completely static analysis is Heine and Lam's [17] work in which a static pointer analysis identifies potential memory leaks in C and C++ through the object ownership abstraction. This work finds double frees and missing frees that occur when the program overwrites the last pointer to an object or

data structure without first freeing it. It does not find growing data structures and thus is complementary to our approach. However, the challenge 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 as types do in Java.

An example of the heap differencing approach to detect memory leaks in Java is Leakbot [19]. Leakbot combines offline analysis with online diagnosis to find data structures which potentially have memory leaks. The offline analysis takes two heap snapshots and does a complete heap differencing to find parts of the graph which may be leaking. It then identifies the data structure(s) which contains these potential leaks. Feeding this information back into the online system, Leakbot then adds expensive object-instance instrumentation only on those types that have already been identified as potentially leaking. Leakbot requires two program executions, both of which include substantial overheads. Cork attains its low overhead and high accuracy by piggybacking on the garbage collector and summarizing the structure versus tracking object instances. Cork also uses simpler heuristics to prune growing types. Compared with their user reports, Cork produces more accurate reports with orders of magnitude fewer reported types by tracking all potentially growing types and pruning the growing types reports based on the type points-to graph.

Compared to Cork, the prior work on detecting heap growth reports many more false positives and has much higher overhead.

3. An Example Memory Leak

We use Figure 2 as a running example throughout the paper. It shows an order processing system for a small business with a leak. *NewOrder* inserts new orders into the `allOrdersHT` hash table and into the `newOrderQ` as shown in Figure 2(a). *ProcessOrders* processes the `newOrderQ` one order at a time as shown in Figure 2(b). It removes each order from the `newOrderQ` and fills it. Then if the customer is a Company, it issues a bill (putting it on the `billingQ`) and ships the order to the customer. When the customer sends a payment, *ProccessBill* removes the order from the `billingQ` and the `allOrdersHT` hash table, as shown in Figure 2(c). However, if the customer is not a company, *ProcessOrders* calls *ProcessPayment* with the customer-provided payment information and only then ships the order. However, *ProcessOrders* should, but does not remove the order from the `allOrdersHT` hash table which results in a memory leak. Figure 2(d) lists the abbreviations and statistics for the different types used in our example.

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

(a) Incoming order

```
1 ProcessOrders() {
2   while (! newOrderQ.isEmpty()) {
3     Order n = newOrderQ.getNext();
4     newOrderQ.remove(n); // removed from NewOrder Q
5     FillOrder(n);
6     if (n.getCustomer() instanceof Company) {
7       IssueBill(n);      // inserts onto Billing Q
8       ShipOrder(n);
9     } else {
10      ProcessPayment(n);
11      ShipOrder(n);
12      // A MEMORY LEAK!! -- not removed from HashTable
13    }
14  }
15 }
```

(b) Processing orders

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

(c) Process bills

Type	Variable	Symbol	Size
HashTable	`allOrdersHT`	H	256
Queue	`newOrderQ`	N	256
Queue	`billingQ`	B	256
Company	`n`	C	64
People	`n`	P	32

(d) Object statistics

Figure 2. Order Processing System

4. Finding Leaks with Cork

This section overviews how Cork identifies and reports the types of leaking objects and correlates them back to the data structure which contains them and the allocation sites that generate them.

Piggybacking on garbage collection, Cork builds a type points-to graph during a full-heap collection. This graph summarizes the volume of all types and the references between them that are live in the current heap. Cork stores this graph between collections, and differences the current graph with previous collections to find types that grow. Cork finds parts of the graph that are growing, and reports these parts back to the user after each full-heap collection. Cork also prunes portions of the graph that substantially shrink to avoid the space and time of storing complete type points-to graphs across multiple collections. This step makes Cork efficient.

Since Cork piggybacks on live-object scanning during garbage collection, it is suitable for use in any mark-sweep or copying collector, but not in a reference counting collector. For clarity of exposition, we describe Cork in the context of a full-heap collector. However, we also demonstrate Cork in a generational collector; it performs the same analysis only during a full-heap collection. An incremental collector that never collects the entire heap at once could incorporate Cork by defining intervals that combine statistics from individual collections until the collector has considered the entire heap. Cork would then compute difference statistics between intervals to detect leaks.

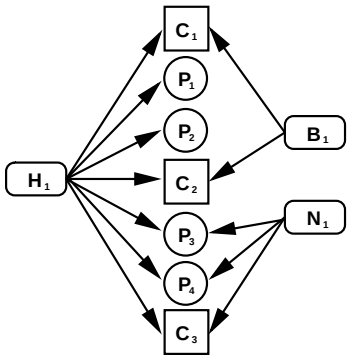
We now describe in detail the information Cork computes to detect leaks, how Cork computes it, and what Cork reports.

4.1 Building the Type Points-To Graph

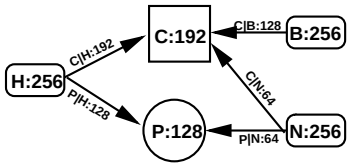
To detect leaks, Cork computes a summary *type points-to* graph annotated with instance and reference volumes and compares these graphs across collections. To minimize the costs of building this graph, Cork piggybacks its construction on the scanning phase of the garbage collector. The scanning phase detects all live objects by starting with the roots (statics, stacks, and registers) and performing a transitive closure through all the object references over all the reachable (live) objects in the heap. For each distinct object type, Cork adds a node to the graph to track the volume of live-object instances with that type. Each time the collector visits a live-object instance of that type, Cork increments the total volume. Cork also adds a directed edge between nodes for each reference from one

```
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    }
12  } else
13    Scanning.scanObject(trace, object);
14 }
```

Figure 3. Object Scanning



(a) Object instance points-to graph



(b) Type points-to graph

Figure 4. The type points-to graph summarizes the object points-to graph

object type to another. When the collector follows a reference between two object instances, Cork increments the edge between the corresponding types of the objects.

Figure 3 shows these two required changes to the scanning code from MMTk in Jikes RVM; these simple additions appear in lines 4 and 9. Assume *scanObject* is processing an object of type A that refers to an object of type B. It takes the tracing routine and object as parameters, finds the object type, and Cork adds line 4 to increment the volume of objects of type A. The collector scans (detects liveness of) an object only once, and thus Cork increments the total volume of this type only once per object instance. Next, *scanObject* must determine if each field of the object has already been scanned. Here, Cork adds line 9 which resolves the referent type of each outgoing reference ($A \rightarrow B$) and increments the volume along the appropriate edge in the graph. This step increments the edge volume for all references to an object (not just the first one). *scanObject* then traces these objects in line 10; if they have not yet been scanned, *trace* enqueues them for scanning. The additional work of the collector depends on whether it is moving objects or not, and is orthogonal to Cork.

At the end of scanning, Cork has constructed a summary type points-to graph for all the live objects. In this graph, the nodes report the volume of each type (V_A) and the edges ($A \rightarrow B$) report the volume of type B pointed to by type A ($V_{B|A}$). Figure 4(a) shows an example of an object instance points-to graph (i.e., the heap itself) with instances of objects of types H, C, P, B, and N for the types from the example order-processing system. Figure 4(b) shows the corresponding type points-to graph annotated with volumes. For example, we use the volumes in Figure 2(d) and calculate the volume of H as 256 ($V_H = 256$). It points to three objects of type C ($V_{C|H} = 192$) and four of type P ($V_{P|H} = 128$). Notice that the sum of the edges and the weight of the node are not the same. For example, objects of type B and N point to the same objects as those from type H.

Cork compares these volumes from distinct collections to determine where growth is occurring in the graph.

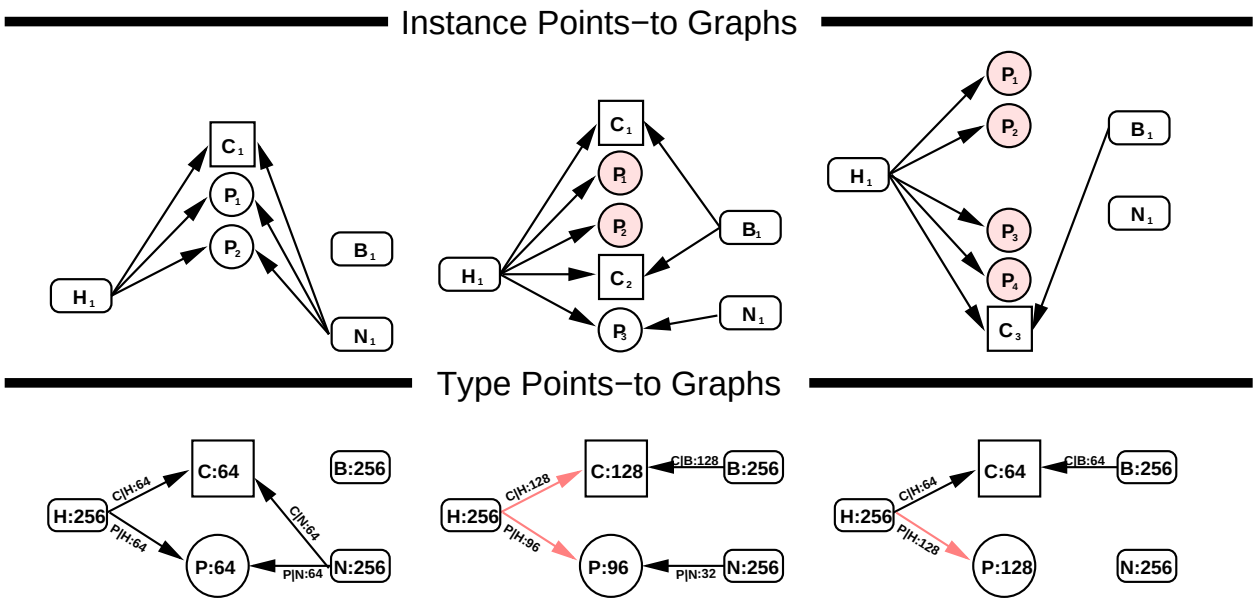
4.2 Finding Heap Growth

At the end of each collection, Cork compares the type points-to graph (TPT) for this collection and previous collections. Cork identifies those nodes whose volumes increase across several collections and reports them as the source of potential leaks. For each type that is growing, Cork examines the incoming edges to the growing types to pinpoint the sources of the growth. In steady state in our implementation, Cork stores a subset of the type points-to graph for the last three collections: TPT_i, TPT_{i-1}, and TPT_{i-2}, where TPT_i is the most recent graph. For efficiency, Cork throws out nodes in TPT_i if the type is in TPT_{i-1} and has substantially shrank.

For example, Figure 5 shows the full type points-to graph created for three collections for the order processing system. Figure 5(a) represents an initial state of the system after three orders arrive, but have not yet been processed. Figure 5(b) shows four orders processed: two billed and two completed. Notice that the program removes the orders from individuals (P) from all the processing queues (B, N), but not from the hash table (H) (line 12 in Figure 2). These orders are leaking. Comparing the type points-to graphs from the first two collections shows both C and P objects are potentially growing. We need more history to be sure. Figure 5(c) represents the state at the next collection after processing more orders, where it becomes clearer that the number of P objects is monotonically increasing, whereas C objects are simply fluctuating. This problem also occurs, though more subtly, when programs iteratively build multiple data structures containing references to the same type, but only one of the data structures is causing heap growth. In this situation, the type's volume may jitter. Thus, Cork cannot only look for monotonic non-decreasing type growth.

To adjust for jitter, Cork uses a *decay factor*, f where $0 < f < 1$ to keep types in the graph that shrink a little on this collection, but may ultimately be growing. Cork compares the current volume of type V_{T_i} to its previous volume: if $V_{T_i} > (1-f) * V_{T_{i-1}}$, Cork keeps the type in the graph, otherwise, it deletes the type. If the type does not appear in TPT_{i-1}, Cork keeps it in the graph. We find that the decay factor is increasingly important as the speed of the leak decreases. Choosing the leak decay factor balances between too much information and not enough. At the end of this phase, all nodes in TPT_i are potentially growing.

Cork then ranks these potentially growing types by how likely they are to leak. In order to calculate an overall rank, we first calculate the *phase growth factor* (g) of each type as $g_{T_i} = p_{T_i} * (Q-1)$, where p is the number of phases that type has potentially been growing and Q is the ratio of volumes of this phase and the previous phase such that $Q > 1$. Cork only reports types that have been potentially growing for some minimum number of phases. Thus, the first time a type appears in a graph, Cork does not report



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

Figure 5. Comparing Type Points-to Graphs to Find Heap Growth

it. Cork accumulates the phase growth factor to rank each type such that absolute growth is rewarded ($r_{T_i} = r_{T_{i-1}} + g_{T_i}$) and decay is penalized ($r_{T_i} = r_{T_{i-1}} - g_{T_i}$). Types above a rank threshold (R_{thres}) are reported as leaking. We use a similar ranking calculation to rank incoming edges to type T in the graph.

4.3 Correlating to Data Structures and Allocation Sites

Just knowing the type of the leaking object is not enough to assist users. For example, reporting that a low-level type such as `String` is leaking is not helpful. Cork instead identifies the leaking data structure by constructing the *slice* of the type points-to graph that contains the type growth. We define a *slice* through the type points-to graph to be the set of all paths originating from vertex t_0 whose rank $r_{t_0} > R_{thres}$ such that the rank of each edge $r_{t_i \to t_{i+1}} > 0.0$. Thus the slice defines the growth originating at type t_0 and following a sequence of types $\{t_0, t_1, \ldots, t_n\}$ and a sequence of edges (t_i, t_{i-1}) where type t_i points to t_{i-1}.

Cork identifies a slice by starting at a leaking type and tracing growth edges backward through the graph until it encounters a non-growing type. In this way, Cork reports not only leaking types, but also the data structure containing them (e.g., the hash table H in our example). Additionally, Cork reports allocation sites for the leaking types. As each allocation site is compiled, Cork assigns it a unique identifier, and constructs a map to it from the appropriate type. For each leaking type, Cork then also reports all allocation sites.

4.4 Implementation Details

One of the main challenges in building the type points-to graph during garbage collection is that Cork is prohibited from allocating memory. Cork thus uses a pool of pre-allocated graph nodes. When Cork needs a node, Cork removes it from the pool. When Cork removes a node from the type points-to graph (because the type is not growing), Cork returns the node to the pool. In the event that the pool cannot fulfill the node request, Cork stops building the type points-to graph for that collection phase, generates a warning to the user, and records the number of requested nodes which could not be fulfilled. At the end of the phase, when Cork is free to allocate again, it allocates more nodes to the pool so that there will be sufficient nodes during the next collection. This approach works well in our benchmarks. Alternative implementations with less overhead include allocating the graph nodes in an immortal space or a separately managed space that gets collected and managed independently from the application heap.

5. Results

This section presents overhead and qualitative results for Cork. First, we present two methodologies: one for detecting leaks in short running programs and one for detecting leaks in long running programs. We explore the parameter space for Cork and show how selecting reasonable values for the decay factor and the rank threshold gives highly accurate results. Finally, we show how Cork identifies the sources of growth in three commonly used benchmarks: fop, _202_jess and SPECjbb2000.

5.1 Methodology

We implement our technique in MMTk, a memory management toolkit in Jikes RVM version 2.3.7. MMTk implements a number of high-performance collectors [7, 6] and Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler [2, 1]. 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.

We evaluate our techniques using the SPECjvm benchmarks, the DaCapo benchmarks, and SPECjbb2000. SPECjbb2000 [22, 23] measures throughput as operations per second for a duration of 2 minutes for an increasing number of warehouses (1 to 8). For our purposes, we change this default behavior. To perform a performance-overhead comparison, we use pseudojbb, a variant of SPECjbb2000 that executes a fixed number of transactions. For memory-leak analysis, we configure SPECjbb2000 to run only one warehouse for 3 hours. We perform all of our experiments on a

Benchmark	GenMS Relative Heap 1.4	1.9	3.0	6.0	SemiSpace with 4MB Trigger
_201_compress	9	6	2	1	31
_202_jess	0	0	0	0	70
_205_raytrace	0	0	0	0	53
_209_db	1	0	0	0	24
_213_javac	3	1	1	0	53
_222_mpegaudio	0	0	0	0	5
_227_mtrt	0	0	0	0	40
_228_jack	2	1	0	0	73
pseudojbb	8	2	0	0	51
SPECjbb2000	*	*	*	*	*
antlr	19	10	5	2	72
bloat	46	18	7	3	err
fop	1	0	0	0	19
jython	3	1	0	0	89
pmd	6	3	1	0	62
ps	0	0	0	0	130
xalan	2	1	0	0	31

Table 1. Number of Full-heap collections at various heap sizes relative to the minimum. *Minimum heap size for SPECjbb2000 depends on length of run.

Benchmark	Percent
_201_compress	0.3
_202_jess	1.7
_205_raytrace	0.4
_209_db	0.0
_213_javac	1.4
_222_mpegaudio	0.5
_227_mtrt	0.3
_228_jack	1.6
pseudojbb	2.6
SPECjbb2000	0.1
antlr	0.7
bloat	0.5
fop	0.2
jython	3.4
pmd	0.2
ps	2.9
xalan	0.0
Average	1.0

Table 2. Space Overhead for Benchmarks

Benchmark	Decay Factor 0%	5%	10%	15%	20%	25%
_201_compress	0	0	0	**0**	0	0
_202_jess	0	1	1	**1**	1	2
_205_raytrace	0	0	0	**0**	0	0
_209_db	0	0	0	**0**	0	0
_213_javac	0	0	0	**0**	0	0
_222_mpegaudio	0	0	0	**0**	0	0
_227_mtrt	0	0	0	**0**	0	0
_228_jack	0	0	0	**0**	0	0
pseudojbb	0	0	0	**0**	0	0
SPECjbb2000	0	4	4	**4**	4	4
antlr	0	0	0	**0**	0	0
bloat	0	0	0	**0**	0	0
fop	2	2	2	**2**	2	2
jython	0	0	0	**0**	0	1
pmd	0	0	0	**0**	0	0
ps	0	0	0	**0**	0	0
xalan	0	0	0	**0**	0	0

Table 3. Reported types as a function of the decay factor at a moderate threshold ($R_{thres} = 100$). We chose the decay factor $f =$ 15%.

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.

Eeckhout et al. [14] show that including adaptive compilation in performance measurements obscures application behavior. Thus, for our overhead measurements, we report a second run with no compilation and a deterministic application of the adaptive compiler using *Replay compilation*. Replay compilation deterministically applies the optimizing compiler to frequently executed methods chosen by the adaptive compiler in previous (offline) runs giving us a realistic mixture of optimized and unoptimized code. We report only application performance by running two iterations of each benchmark. The first run uses replay compilation, and then turns off compilation. Before the second iteration, a whole heap collection flushes compiler objects from the heap.

For performance results, we explore the time-space trade-off by executing each program on five heap sizes, ranging from the smallest one possible for the execution of the program to six times that size. 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.

For qualitative results, we explore two different garbage collection configurations. Cork needs to have a chance to gather statistics over a number of full-heap collections. With a generational collector, many of these benchmarks have fewer than 10 full-heap collections, which is not enough fodder for Cork (see Table 1). Therefore, we use a single-space collector (SemiSpace) with frequent collections (every 4MB of allocation) for short running programs. This configuration increases the number of type points-to graphs Cork calculates and the likelihood that it can accurately identify the leaking type. For long running programs, Cork does not require this collector configuration. It can run much less frequently and still find heap growth. In an online deployment, Cork could be added to a generational mark-sweep collector (GenMS) and thus we report these overhead numbers as well.

5.2 Overhead Results

Figure 6 shows the total time overhead for our two different garbage collector configurations. In a single-spaced collector with frequent collections, it shows an average overhead of 45.5% in very tight heaps (minimum heap size), 35.0% in moderate heaps (three times minimum heap size), and 34.4% in large heaps (six times minimum heap size). In the worst case, the overhead was 130% for pseudojbb in a very tight heap. In the generational mark-sweep collector, Cork only calculates the summary points-to graph on full-heap collections. In this case, the overhead average is 13.3% in very tight heaps, 2.4% for moderate heaps, and 1.7% for large heaps. In the worst case, the overhead was 52% in a very tight heap, again for pseudojbb. These very small heaps however are not representative of typical heap configurations.

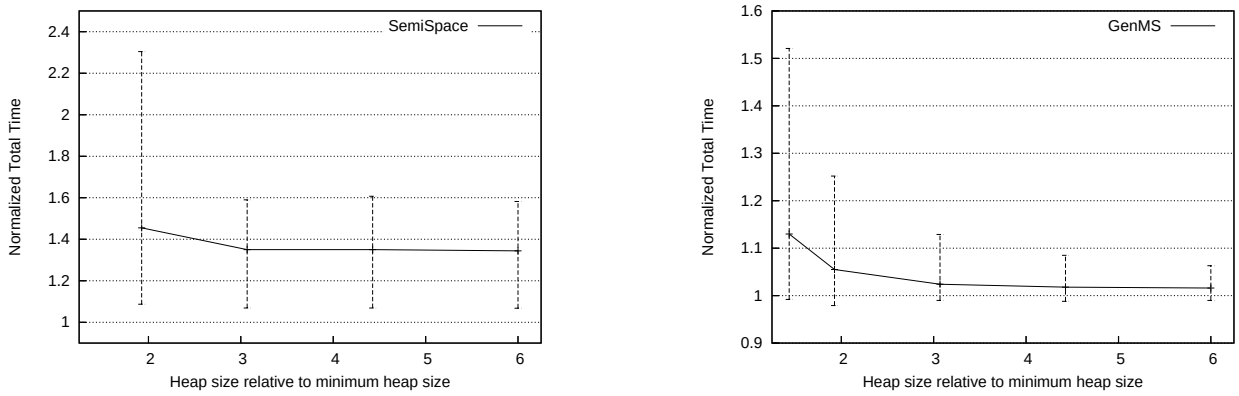
The sources of overhead are two-fold. First, construction and scanning of the points-to graph increases the pause time of the garbage collector. Second, the objects required for the construction and saving of the points-to information are long-lived heap objects. Table 2 shows an average space overhead of 1.0% over total allocation. This rise in heap activity increases the burden on the garbage collector. One could reduce this overhead by allocating these objects in a separate space that would not need to be collected.

5.3 Decay Factor and Rank Thresholds

We experiment with different sensitivities for both the decay factor (f) and the rank threshold (R_{thres}). Table 3 shows how changing the decay factor changes the number of reported types whose vol-

Thresholds:	> 0				> 50				> 100				> 200			
Percent of GCs:	0	10	25	50	0	10	25	50	0	10	25	50	0	10	25	50
_201_compress	9	6	4	2	0	0	0	0	0	0	**0**	0	0	0	0	0
_202_jess	22	12	9	2	4	2	1	1	2	1	**1**	1	2	1	1	1
_205_raytrace	31	8	4	0	0	0	0	0	0	0	**0**	0	0	0	0	0
_209_db	7	7	2	1	2	2	0	0	2	2	**0**	0	2	2	0	0
_213_javac	118	92	71	39	16	5	2	1	5	0	**0**	0	3	0	0	0
_222_mpegaudio	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0
_227_mtrt	43	12	3	0	6	6	2	0	4	4	**0**	0	0	0	0	0
_228_jack	35	22	9	1	0	0	0	0	0	0	**0**	0	0	0	0	0
pseudojbb	50	13	9	7	5	0	0	0	1	0	**0**	0	0	0	0	0
SPECjbb2000	138	14	10	9	66	7	6	2	46	5	**4**	0	38	4	4	0
antlr	83	26	9	1	15	0	0	0	12	0	**0**	0	9	0	0	0
bloat	60	51	33	5	30	11	0	0	11	0	**0**	0	1	0	0	0
fop	107	74	35	24	12	11	2	1	7	6	**2**	1	4	3	1	1
jython	51	5	3	0	4	0	0	0	2	0	**0**	0	2	0	0	0
pmd	127	89	11	1	21	5	2	0	17	2	**0**	0	13	0	0	0
ps	16	8	3	0	2	0	0	0	0	0	**0**	0	0	0	0	0
xalan	40	11	5	4	0	0	0	0	0	0	**0**	0	0	0	0	0

Table 4. Reported data structures as a function of rank thresholds and percentage of collections in which the type appears for a moderate decay factor ($f = 15\%$). We choose the rank threshold $R_{thres} = 100$.



(a) Single-space collector with frequent collections (4MB)

(b) Generational collector

Figure 6. Geometric Mean Overhead Graphs over all benchmarks. Bars show minimum and maximum overheads.

ume increases between the first and third collections for a full-heap collector. We find that the detection of growing types is not very sensitive to small changes in the decay factor. We choose a moderate decay factor ($f = 15\%$) for which Cork accurately identifies the growing data structures in _202_jess and fop without any false positives. For pseudojbb, we find that Cork gives a false negative for some configurations. In this case, Cork does not have the opportunity to accumulate sufficient rank for growing types responsible for a known memory leak. For SPECjbb2000, on the other hand, Cork reports the leak because it runs for a longer period of time. Table 4 shows how increasing the rank threshold eliminates false positives from our reports. We find that a moderate rank threshold ($R_{thres} = 100$) is sufficient for eliminating any false positives.

5.4 Finding and Fixing Leaks

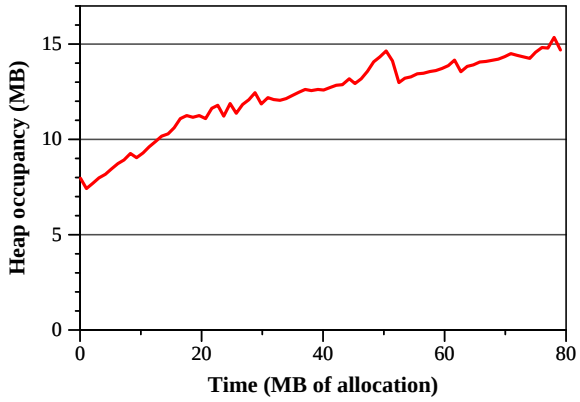
This section describes the data structure growth that Cork finds in fop, _202_jess, and SPECjbb2000. Each section describes the benchmark, the Cork report, and the analysis.

5.4.1 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 7(a) which clearly demonstrates an overall monotonic heap growth.

Cork analyzes fop and gives reports shown in Figure 7(b)-(d). Figure 7(b) shows Cork's reports of leaking types and their associated ranks for three different garage collections. It shows `ArrayList` with a large rank making it the most likely cause of a potential leak. We continue our exploration by examining the slices of the type points-to graph to determine what is keeping the `ArrayList` alive. Figure 7(c) shows part of the slice for `ArrayList`. The figure represents the points-to relation between different types in the graph. It uses ← to represent the direction of the points-to relationship with the indentation depth indicating the referent. It shows that `ArrayLists` are nested in a data structure. Interestingly, Cork is able to distinguish that the `ArrayList` is implemented using an `Object[]` and reports both. Finally, Cork lists the allocation sites for all the types giving the user a starting point for debugging (see Figure 7(d)). 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: the first parses the formatting object tree building a complex structure of `ArrayLists` containing different formatting objects



(a) Heap-occupancy graph for fop

GC No.	Rank	Type
15	2104.17	Ljava/util/ArrayList;
	135.26	Lorg/apache/fop/fo/LengthProperty;
	115.64	Lorg/apache/fop/datatypes/AutoLength;
18	2106.28	Ljava/util/ArrayList;
21	2108.24	Ljava/util/ArrayList;

(b) Leaking Type Report for three different garbage collections

Type
Ljava/util/ArrayList;
← Lorg/apache/fop/layout/inline/WordArea;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;

← Lorg/apache/fop/layout/LineArea;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;
← Lorg/apache/fop/layout/inline/WordArea;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;
← Lorg/apache/fop/layout/inline/InlineSpace;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;
...

(c) Slice Report for `ArrayList`

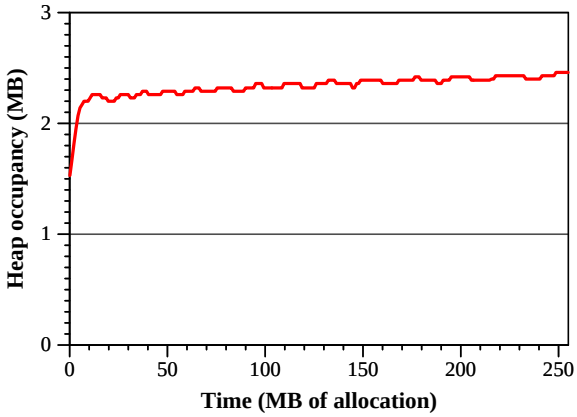
Method	bcidx
Ljava/util/ArrayList;	
too numerous to be useful (45)	
Lorg/apache/fop/layout/inline/WordArea;	
.../render/pdf/PDFRenderer;.renderWordArea...	355
.../render/pdf/PDFRenderer;.renderWordArea...	450
.../render/pdf/PDFRenderer;.renderWordArea...	555
.../render/pdf/PDFRenderer;.renderWordArea...	811
.../render/pdf/PDFRenderer;.renderWordArea...	820
Lorg/apache/fop/layout/LineArea;	
.../layout/BlockArea;.getCurrentLineArea...	26
.../layout/BlockArea;.createNextLineArea...	45

(d) Allocation Sites Report

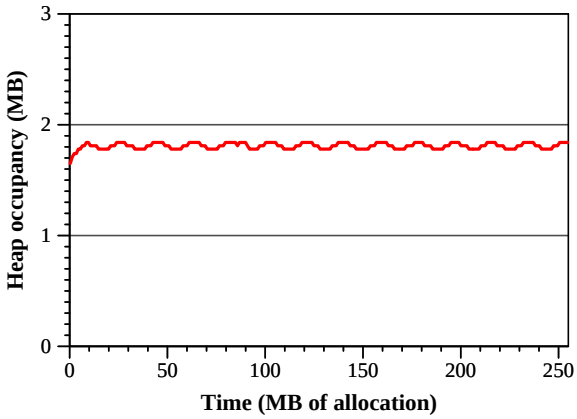
Figure 7. Fixing fop

Type
Lspec/benchmarks/_202_jess/jess/Value;
← Lspec/benchmarks/_202_jess/jess/Value; []
← Lspec/benchmarks/_202_jess/jess/ValueVector;
← Lspec/benchmarks/_202_jess/jess/ValueVector; []
← Lspec/benchmarks/_202_jess/jess/Token;
← Lspec/benchmarks/_202_jess/jess/Token; []
← Lspec/benchmarks/_202_jess/jess/TokenVector;
...

(a) Slice Report for `Value`



(b) Original input file



(c) Fixed input file

Figure 8. Fixing _202_jess

which themselves can contain `ArrayLists`. Once fop encounters an end of page sequence, it begins rendering during a second pass of the data structure it built during parsing. Unfortunately, it is not possible to start rendering earlier because of the possibility of *forward references* in the XSL-FO tree. Thus, 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 implementation choices [3].

5.4.2 _202_jess

From the SPECjvm benchmark suite, _202_jess is a Java Expert Shell System based on NASA's CLIPS. In an expert system, the input is a set of facts and a set of rules. Each fact represents some existing relationship and each rule represents some 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.

Cork analyzes _202_jess and finds that `Value` is overwhelmingly a growing type reporting a slice showing in Figure 8(a). Correlating it to the implementation, _202_jess compiles all the rules into a single set of nodes. The assertion or retraction of a fact is then turned into a *token*, which is fed to the input nodes of the network. Each node in the network 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 `Token` in a `ValueVector` implemented as `Value[]`. A global `TokenVector`, implemented as `Token[]`, stores the tokens in the system. Since original facts are part of the input, we examine this path further.

Examining the input for _202_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 kind of 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 removed this artificial complexity from the input file. Figure 8(b) shows the heap occupancy graph for the original input file. It shows a growth of 45KB every 64MB. Figure 8(c) shows the resulting heap occupancy graph once we removed the artificial complexity. The heap growth, and thus the memory leak, is gone.

5.4.3 SPECjbb2000

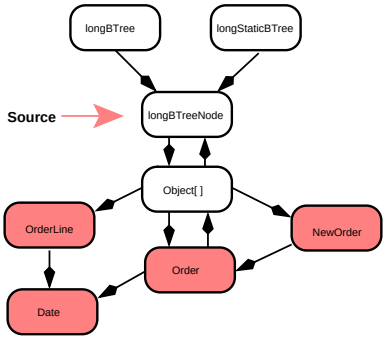
The SPECjbb2000 benchmark models a wholesale company that consists of a number of warehouses (or districts). Each warehouse has one terminal where customers can generate requests, such as placing new orders or requesting the status of an existing order. The warehouse executes operations in sequence, with each operation selected from the operation mix 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 BTrees and other data structures. Figure 1(b) shows the heap occupancy graph for SPECjbb2000 running one warehouse for one hour.

Cork's analysis finds four types as possible leaks: `Order`, `Date`, `NewOrder`, and `OrderLine` (see Figure 9(a)). The rank of the four types changes between garbage collections making it difficult to determine the importance of each. Further examination into the slices of the four reported types reveal why. Figure 9(b) shows the complex structure of types that SPECjbb2000 creates (the shaded types are growing). Clearly there is an interrelationship between all of the leaking types and if one is leaking then the rest are as well, hence the reordering of the ranks in our reports. Interestingly, `Object[]`'s use in SPECjbb2000 is prolific and as a result its volume jitters sufficiently that it never shows sufficient growth to be reported as leaking. It does, however, appear in the slice containing the data structures.

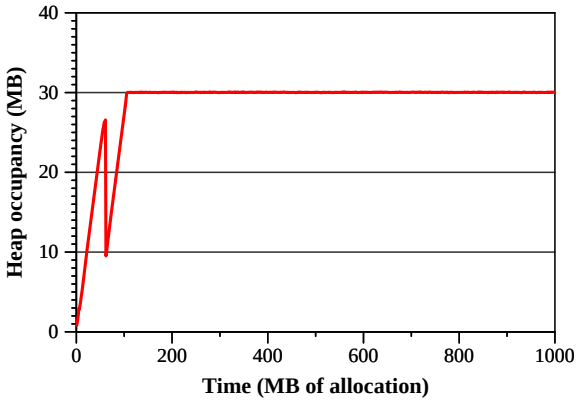
We correlate Cork's results with SPECjbb2000's implementation. We find that orders are placed in an `orderTable`, implemented as a BTree, when they are created. When they are completed during a *DeliveryTransaction*, they are not properly removed from the `orderTable`. By adding code to properly remove the

Type
Lspec/jbb/NewOrder; ← Ljava/lang/Object; [] ← Lspec/jbb/infra/Collections/longBTreeNode; ← Ljava/lang/Object; [] ← Lspec/jbb/infra/Collections/longBTree; ← Lspec/jbb/infra/Collections/longStaticBTree; ← Lspec/jbb/Order;
Lspec/jbb/Order; ← Ljava/lang/Object; [] ← Lspec/jbb/infra/Collections/longBTreeNode; ← Ljava/lang/Object; [] ← Lspec/jbb/infra/Collections/longBTree; ← Lspec/jbb/infra/Collections/longStaticBTree; ← Lspec/jbb/Order; ← Lspec/jbb/NewOrder;
Lspec/jbb/Orderline; ← Ljava/lang/Object; [] ← Lspec/jbb/infra/Collections/longBTreeNode; ← Ljava/lang/Object; [] ← Lspec/jbb/infra/Collections/longBTree; ← Lspec/jbb/infra/Collections/longStaticBTree; ← Lspec/jbb/Order;
Ljava/util/Date; ← Lspec/jbb/Order; ← Lspec/jbb/Orderline;
...

(a) Slice Report for SPECjbb2000



(b) Slice diagram



(c) Fixed heap-occupancy graph

Figure 9. Fixing SPECjbb2000

orders from the `orderTable`, we remove this memory leak. Figure 9(c) shows the heap occupancy after correcting the memory leak and running SPECjbb2000 with one warehouse for one hour.

6. Conclusion

In this paper, we introduce Cork and show that it can detect and report memory leaks in Java programs. Cork identifies the data structure and allocation sources of these leaks. Cork traces growth in the heap and report slices of a summarizing type points-to graph that it calculates by piggybacking on full-heap garbage collections. We show that Cork adds only 2.4% overhead on average to moderate heaps for our benchmarks, and 1.7% overhead on average for large heaps in a generational collector. We use Cork to precisely identify data structures with unbounded heap growth leaks in three popular benchmarks: fop, _202_jess, and SPECjbb2000. We use Cork's analysis to eliminate memory leaks in _202_jess and SPECjbb2000. Cork is the first tool to find memory leaks in Java with low enough overhead to consider using online. Cork is also accurate, reporting few false positives regardless of its configuration and all the leaks in our benchmarks programs.

Acknowledgments

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 SPECjbb2000.

References

[1] B. Alpern et al. Implementing Jalapeño in Java. In *The Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 314–324, Denver, CO, November 1999.

[2] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] The Apache XML Project. *Using FOP Documentation*, release 0.20.5 edition, October 2005.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *The Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *The Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *The Joint International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, New York, NY, June 2004.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *The International Conference on Software Engineering*, pages 137–146, Scotland, UK, May 2004.

[8] H.-J. Boehm. Space efficient conservative garbage collection. In *The Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[9] J. Campan and E. Muller. Performance tuning essential for J2SE and J2EE: Minimize memory leaks with Borland Optimizeit Suite. Technical report, Borland Software Corporation, March 2002.

[10] T. M. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, Boston, Massachusetts, October 2004.

[11] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *The Conference on Object-Oriented Technologies and Systems*, pages 219–234, Santa Fe, New Mexico, April 1998.

[12] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *The European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 116–134, Lisbon, Portugal, June 1999. Springer Verlag.

[13] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12:1431–1454, 2000.

[14] L. Eeckhout, A. Georges, and K. D. Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *The Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, October 2003.

[15] 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.

[16] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkley, CA, USA, January 1992.

[17] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *The Conference on Programming Language Design and Implementation*, pages 168–181, 2003.

[18] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error free garbage collection traces: How to cheat and not get caught. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 140–151, Marina Del Rey, CA, June 2002.

[19] N. Mitchell and G. Sevitzky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *The European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377, Darmstadt, Germany, July 2003.

[20] QuestSoftware. JProbe Profiler. http://www.quest.com/jprobe/profiler.asp.

[21] R. Shaham, E. K. Kolodner, and S. Sagiv. Automatic removal of array memory leaks in Java. In *The International Conference on Compiler Construction*, pages 50–66, London, UK, 2000. Springer-Verlag.

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

[23] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[24] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

[25] Sun Microsystems. Heap Analysis Tool. https://hat.dev.java.net/.

[26] Sun Microsystems. HPROF Profiler Agent. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.