

Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages^{*}

Maria Jump and Kathryn S. McKinley

Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, USA
{mjump,mckinley}@cs.utexas.edu

Abstract

A *memory leak* in a garbage-collected program occurs when the program inadvertently maintains references to objects that it no longer needs. Memory leaks cause systematic heap growth, degrading performance and resulting 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 a dynamic heap-summarization technique based on type that accurately identifies leaks, is space efficient (adding less than 1% to the heap), and is time efficient (adding 2.3% on average to total execution time). We implement this approach in *Cork* which utilizes dynamic type information and garbage collection to summarize the live objects in a *type points-from graph (TPFG)* whose nodes (types) and edges (references between types) are annotated with volume. *Cork* compares *TPFGs* across multiple collections, identifies growing data structures, and computes a *type slice* for the user. *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 (non-experts) to quickly eliminate growing data structures in SPECjbb2000 and Eclipse, something their developers had not previously done. *Cork* is accurate, scalable, and efficient enough to consider using online.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging—Debugging aids

General Terms Memory Leak Detection

Keywords memory leaks, runtime analysis, dynamic, garbage collection

1. Introduction

Memory-related bugs are a substantial source of errors, and are especially problematic for languages with explicit memory man-

agement. 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 to objects the program never uses again.

Garbage collection corrects 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 growing data structure with unused parts will cause the program to run out of memory and crash. Even if a growing data structure is not a true leak, application reliability and performance can suffer. 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.

Previous approaches for finding leaks use heap diagnosis tools that rely on a combination of heap differencing [10, 11, 20] and allocation and/or fine-grain object tracking [7, 8, 9, 13, 19, 24, 25, 28, 29]. These techniques 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. These reports require a lot of time and expertise to interpret. Thus, prior work lacks precision and efficiency.

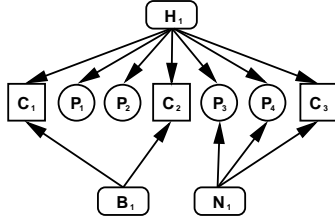
This paper introduces *Cork*, an accurate, scalable, online, and low-overhead memory leak detection tool for typed garbage-collected languages. *Cork* uses a novel approach to summarize, identify, and report data structures with systematic heap growth. We show that it provides both efficiency and precision. *Cork* piggybacks on full-heap garbage collections. As the collector scans the heap, *Cork* summarizes the dynamic object graph by type (class) in a summary *type points-from graph (TPFG)*. The nodes of the graph represent the volume of live objects of each type. The edges represent the points-from relationship between types weighted by volume. At the end of each collection, the *TPFG* completely summarizes the live-object points-from relationships in the heap.

For space efficiency, *Cork* stores type nodes together with global *type information block (TIB)*. The TIB, or equivalent, is a required implementation element for 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 *TPFG* scales with the type system. While the number of edges between types are quadratic in theory, programs implement simpler type relations in practice; we find that the edges

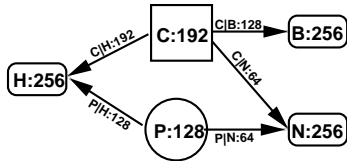
^{*} This work is supported by NSF CCR-0311829, NSF ITR CCR-0085792, NSF CCF-0429859, NSF EIA-0303609, DARPA F33615-03-C-4106, DARPA NBCH30390004, Intel, IBM, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

Type	Symbol	Size
HashTable	H	256
Queue	N	256
Queue	B	256
Company	C	64
People	P	32

(a) Object statistics



(b) Object points-to graph



(c) Type points-from graph

Figure 1. The type points-from graph summarizes the object points-to graph

are linear in the number of types. The *TPFG* never adds more than 0.5% to heap memory.

Cork stores and compares *TPFGs* from multiple collections to detect and report a dynamic *type slice* with systematic heap growth. We store points-from instead of points-to information to efficiently compute the candidate type slice. We demonstrate that the construction and comparison of *TPFGs* across multiple collections adds on average 2.3% to total time to a system with a generational collector.

We apply Cork to 14 popular benchmarks from DaCapo b.050224 [5] and SPECjvm [26, 27] benchmark suites. Cork identifies and reports unbounded heap growth in three of them. We confirm no additional memory leaks in the other 11 benchmarks by examining their heap composition graphs [6]. Additionally we apply Cork to a known memory leak in Eclipse bug#115789. In this paper, we report detailed results for the two largest programs: (1) SPECjbb2000 which grows at a rate of 128KB every 64MB of allocation, and (2) Eclipse bug#115789 which grows at a rate of 2.97MB every 64MB of allocation. Using a generational collection, Cork precisely pinpointed the single data structure responsible for the growth after six full-heap collections. We used this precision to quickly identify and eliminate the memory leaks.

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. Finding Leaks with Cork

This section overviews how Cork identifies potentially growing type nodes (*candidate leaks*) and reports their corresponding type

```

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 2. Object Scanning

to the user along with the data structure which contains them and the allocation sites that generate them. For clarity of exposition, we describe Cork in the context of a full-heap collector using an example program whose types are defined in Figure 1(a).

2.1 Building the Type Points-From Graph

To detect growth, Cork summarizes the heap in a *type points-from graph (TPFG)* annotated with instance and reference volumes. The *TPFG* consists of *type nodes* and *reference edges*. The type nodes represent the total volume of objects of type t (V_t). The reference edges are directed edges from type node t' to type node t and represent the volume of objects of type t' that are referred to by objects of type t ($V_{t'|t}$). To minimize the costs associated with building the *TPFG*, 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 reachable (live) object o visited, Cork determines the object’s type t and increments the corresponding type node by the object’s size. Then for each reference from o to object o' , it increments the reference edge from t' to t by the size of o' . At the end of the collection, the *TPFG* completely summarizes the volume of all types and references that are live at the time of the collection.

Figure 1(b) shows an object points-to graph (i.e., the heap itself). Each vertex represents a different object in the heap and each arrow represents a reference between two objects. Figure 2 shows the modified scanning code from MMTk in Jikes RVM: Cork requires two simple additions that appear as lines 4 and 9. Assume *scanObject* is processing an object of type B that refers to an object of type C (from Figure 1(b)). It takes the tracing routine and object as parameters and finds the object type. Line 4 increments the volume of type B (V_B) in the node for this instance. Since the collector scans (detects liveness of) an object only once, Cork increments the total volume of this type only once per object instance. Next, *scanObject* must determine if each referent of the object has already been scanned. As it iterates through the fields (slots), the added line 9 resolves the referent type of each outgoing reference ($B \rightarrow C$) and increments the volume along 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 an additional type lookup 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 *TPFG* completely summarizes the live objects in the heap. Figure 1(c) shows the *TPFG* for our example. Notice that the reference edges in the *TPFG* point in the opposite direction of the references in the heap. Also notice that, in the heap, objects of type C are referenced by H , B , and N represented by the outgoing reference edges of C in the *TPFG*.

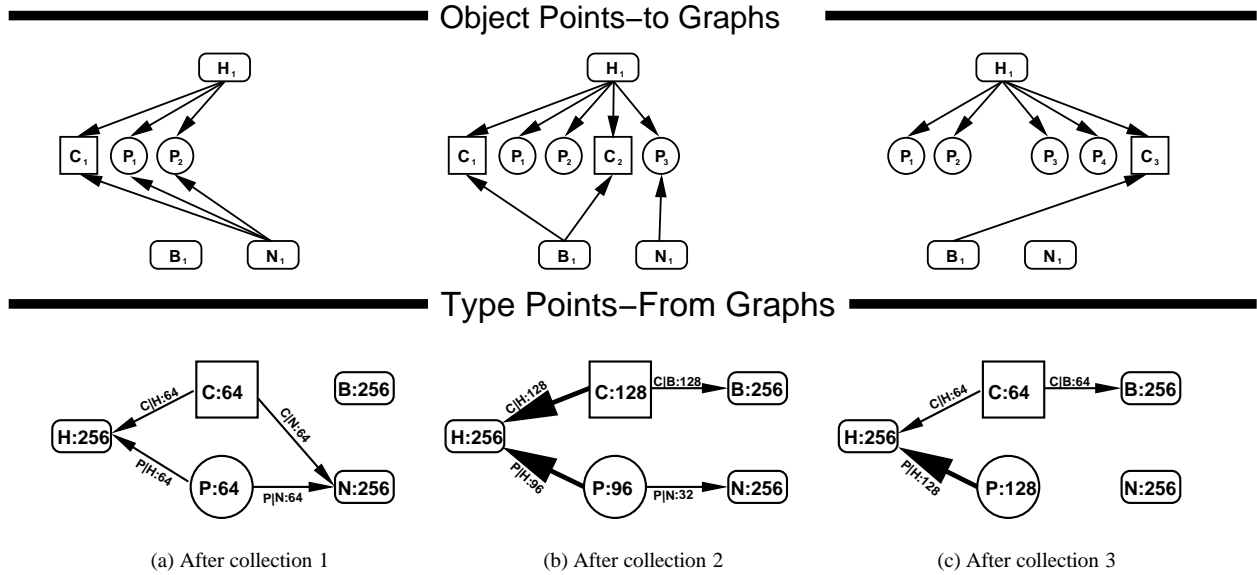


Figure 3. Comparing Type Points-From Graphs to Find Heap Growth

Since C has multiple references to it in the heap, the sum of the weights of its outgoing reference edges is greater than its type node weight. Cork differences the $TPFG$ volumes from distinct collections to determine where growth is occurring in the heap.

Cork uses volume rather than a simple count to detect array growth when the number of arrays remains constant but their sizes grow. Additionally, volume gives a heavier weight to larger types which tend to make the heap grow faster than smaller types.

2.2 Finding Heap Growth

At the end of each collection, Cork differences the $TPFG$ for the current collection with previous collections reporting those types nodes whose volumes increase across several collections as candidate leaks. For each type node that is growing, Cork follows growing reference edges through the $TPFG$ to pinpoint the growth.

For example, Figure 3 shows the full $TPFG$ created during three collections of our example program. Notice that objects C and P are added to both the hashtable and queues. When they are finished being used, they are removed from all of the queues but not from the hashtable, causing a memory leak. Comparing the $TPFG$ from the first two collections shows both C and P objects are potentially growing (depicted with bold arrows). We need more history to be sure. Figure 3(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 type volume *jitters* – fluctuates with high frequency – though the overall trend may show growth. Thus, Cork must look for more than monotonically non-decreasing type growth between two consecutive collections.

Cork differences the $TPFG$ from the current collection with that of previous collections looking for growing types and ranks them. We examine two different methods for ranking types [16]. Because of space constraints, we only present the more robust of these techniques: the Ratio Ranking Technique.

The Ratio Ranking Technique (RRT) ranks type nodes according to the ratio of volumes Q between two consecutive $TPFG$, finds the type nodes with ranks above a rank threshold ($r_i > R_{thres}^i$), and reports the corresponding types as *candidates*. Additionally, RRT uses a *decay factor* f , where $0 < f < 1$ to adjust for jitter. To be

considered a potential leak, RRT considers only those type nodes whose volumes satisfy $V_{T_i} > (1 - f) * V_{T_{i-1}}$ on consecutive collections as potential candidates. The decay factor keeps type nodes that shrink a little in this collection, but which may ultimately be growing. We find that the decay factor is increasingly important as the size of the leak decreases. Choosing the leak decay factor balances between too much information and not enough.

To rank type nodes, RRT first calculates the *phase growth factor* (g) of each type node as $g_t = p_t * (Q - 1)$, where p is the number of phases (or collections) that t has been potentially growing and Q is the ratio of volumes of this phase and the previous phase such that $Q > 1$. Since $Q > 1$, $g > 0$. Each type node's rank r_t is calculated by accumulating phase growth factors g over several collections such that absolute growth is rewarded ($r_t = r_{t-1} + g_t$) and decay is penalized ($r_t = r_{t-1} - g_t$). Higher ranks represent a higher likelihood that the corresponding volume of the type grows without bound. Since RRT only reports types that have been potentially growing for some minimum number of phases, the first time a type appears in a graph, RRT does not report it. Cork ranks reference edges (r_e) using the same calculation.

2.3 Correlating to Data Structures and Allocation Sites

Reporting a low-level type such as `String` as a potential candidate is not very useful. Cork identifies the growing data structure that contains the candidate growth by constructing a *slice* in the $TPFG$. We define a *slice* through the $TPFG$ to be the set of all paths originating from type node t_0 such that the rank of each reference edge $r_{t_k \rightarrow t_{k+1}}$ on the path is positive. Thus, a slice defines the growth originating at type node t_0 following a sequence of type nodes $\{t_0, t_1, \dots, t_n\}$ and a sequence of reference edges (t_k, t_{k+1}) where type node t_k points to t_{k+1} in the $TPFG$.

Cork identifies a slice by starting at a candidate type node and tracing growing reference edges through the graph until it encounters a non-growing type node with non-growing reference edges. This slice contains not only candidates, but also the dynamic data structure containing them. Additionally, Cork reports type 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 allocations site for the candidate type. As each

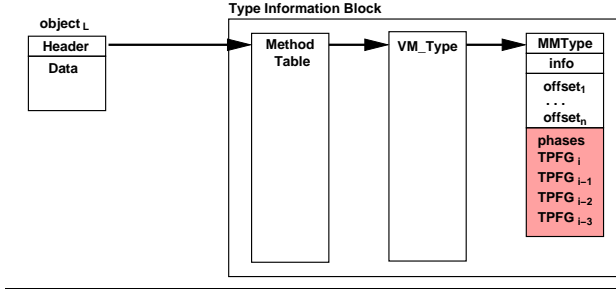


Figure 4. Type Information Block (TIB)

allocation site is compiled, Cork assigns it a unique identifier, and constructs a map (a *SiteMap*) to it from the appropriate type. For each leaking type, Cork searches the map to find allocations sites for that type. For each type, the SiteMap includes the method and byte-code index (*bcidx*) or line for each allocation site.

2.4 Implementation Efficiency and Scalability

We implement several optimizations to make Cork’s implementation scalable and efficient in both time and space. First, we find Cork can accurately detect heap growth keeping a history of only four *TPFG*s: $TPFG_i$, $TPFG_{i-1}$, $TPFG_{i-2}$, and $TPFG_{i-3}$. Cork piggybacks type nodes on the VM’s global *type information block* (TIB). This structure or an equivalent is required for a correct implementation of Java or C#. Figure 4 shows the modified TIB from Jikes RVM. Notice that every live object of a type ($object_L$) points to the TIB corresponding to its type. The TIB consists of three different parts. The first is the method table which stores pointers to code for method dispatch. The method table points to a corresponding *VM_Type* which stores information used by the VM for efficient type checking and 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 garbage collection. Recall from Figure 2 that object scanning resolves the *MMType* of each object (line 3). Thus, Cork stores *TPFG* type node data for each *TPFG* in the corresponding *MMType* adding only four extra words. One additional word stores the number of consecutive collections that a type potentially grows. Thus, the type nodes scale with the type system of the VM.

While the number of reference edges in the *TPFG* are quadratic in theory, one class does not generally reference all other classes. Programs implement a much simpler type hierarchy, and we find that the reference edges are linear with respect to the type nodes. This observation modifies simply edge implementation consisting of a pool of available edges (*edge pool*). Each time a class (type) is loaded, the edge pool grows. Additionally, the edge pool can grow dynamically whenever it runs out of edges. New edges are added to the *TPFG* 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 to prevent from adding any extra words to the *MMType* structure. We further reduce the space required for reference edges by pruning those that do not grow.

2.5 Cork in Other Collectors

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 *scanning* collector). Cork can also be added as described to any scanning collector that does periodic whole-heap collections. In these configurations, Cork performs the analysis only during full-heap collections. To find leaks in our benchmarks, Cork needed approximately six collections during which heap growth occurs. A scanning incremental collector

that never collects the entire heap may add Cork by defining intervals that combine statistics from multiple collections until the collector has considered the entire heap. Cork would then compute difference statistics between intervals to detect leaks.

2.6 Cork in Other Languages

Cork’s heap summarization, the *TPFG*, relies on the garbage collector’s ability to determine the type of an object. We exploit the object model of polymorphically typed languages (such as Java and C#) by piggybacking on their required global type information to keep space overheads to a minimum. There are, however, other implementation options. For garbage-collected languages that lack global type information (such as Standard ML), other mechanism may be able to provide equivalent information. Previous work for provide some suggestions for functional languages that tag objects [21, 22, 23]. For example, type-specific tags could be used to index into a hashtable for storing type 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.

3. Results

This section presents overhead and qualitative results for Cork. Section 3.1 describes our methodology, Section 3.2 reports Cork’s space overhead, and Section 3.3 reports its performance. Section 3.4 shows the accuracy of the Slope Ranking Technique and shows how a variety of reasonable values for the decay factor and the rank threshold gives highly accurate results using the Ratio Ranking Technique. After applying and identifying unbounded heap growth in four commonly used benchmarks, Section 3.5 details the two largest: SPECjbb2000 and Eclipse.

3.1 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 [3, 4] and Jikes RVM is a high-performance VM written in Java with an aggressive optimizing compiler [1, 2]. 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. Additionally, we remove the nondeterministic behavior of the adaptive compilation system by applying replay compilation [15].

We evaluate our techniques using the SPECjvm [26], DaCapo b.050224 [5], SPECjbb2000 [27], and Eclipse [30]. Table 1(a) 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 the number of types (*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 types potentially analyzed at each collection.

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.

Benchmark	(a) Benchmark Statistics					(b) Type Points-From Statistics							(c) Space Overhead				
	Alloc MB	# of Colltn		# of types		# of types		# edges per type		# edges per TPGF		% pruned	TIB		TIB+Cork		
		whl	gen	bm	+VM	avg	max	avg	max	avg	max		MB	%H	MB	%H	Diff
Eclipse	3839	73	11	1773	3365	667	775	2	203	4090	7585	42.2	0.53	0.011	0.70	0.015	0.167
fop	137	9	0	700	2292	423	435	3	406	1559	2623	45.2	0.36	0.160	0.55	0.655	0.495
pmd	518	36	1	340	1932	360	415	3	121	967	1297	66.0	0.30	0.031	0.44	0.186	0.155
ps	470	89	0	188	1780	314	317	2	93	813	824	66.3	0.28	0.029	0.39	0.082	0.053
javac	192	15	0	161	1753	347	378	3	99	1118	2126	45.8	0.28	0.071	0.43	0.222	0.151
ython	341	39	0	157	1749	351	368	2	114	928	940	66.2	0.28	0.041	0.39	0.112	0.071
jess	268	41	0	152	1744	318	319	2	89	844	861	66.0	0.27	0.049	0.38	0.143	0.094
antlr	793	119	6	112	1704	320	356	2	123	860	1398	55.8	0.27	0.016	0.39	0.282	0.266
bloat	710	29	5	71	1663	345	347	2	101	892	1329	50.6	0.26	0.017	0.38	0.064	0.047
jbb2000	**	**	**	71	1663	318	319	2	110	904	1122	59.0	0.26	**	0.38	**	**
jack	279	47	0	61	1653	309	318	2	107	838	878	66.2	0.26	0.042	0.37	0.131	0.089
mtrt	142	17	0	37	1629	307	307	2	91	820	1047	57.5	0.26	0.081	0.37	0.258	0.177
raytrace	135	20	0	36	1628	305	306	2	91	814	1074	56.1	0.26	0.085	0.37	0.272	0.187
compress	106	6	3	16	1608	286	288	2	89	763	898	60.9	0.25	0.105	0.36	0.336	0.231
db	75	8	0	8	1600	289	289	2	91	773	787	66.1	0.25	0.160	0.35	0.467	0.307
Geomean	303	27	n/a	104	1813	342	357	2	116	1000	1303	57.4	0.29	0.048	0.41	0.168	0.145

Table 1. Benchmark Characteristics. **Volumes for SPECjbb2000 depend on how long we allow the warehouse to run.

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 a fixed number of 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).

3.2 Space Overhead

Table 1(b) reports *TPFG* space overhead statistics. Columns one and two (*# of types*) report the average and maximum number of types in the heap during any particular garbage collection. We notice that while many more types exist in the system, an average of 44% of them are present in the heap at a time. This feature reduces the number of potential candidates that Cork must analyze.

Table 1(b) shows the average (column 3) and maximum (column 4) number of reference edges per type node in the *TPFG*. We find that most type nodes have a very small number of outgoing reference edges (2 on average). The more prolific a type 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 *TPFG* (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 type nodes in practice.

Finally, Table 1(c) 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% (column 5)). For the longest-running and largest program, Eclipse, Cork has a tiny space overhead (0.004%). Thus Cork is both scalable and space-efficient in practice.

3.3 Performance Overhead Results

Cork’s overhead results from constructing the *TPFG* during scanning and from differencing between *TPFGs* to find growth at the end of each collection phase. Figure 5 graphs the normalized geometric mean over all benchmarks to show overhead in scan time,

Benchmark	(a) Decay Factor			(b) Rank Threshold		
	0%	15%	25%	0	100	200
Eclipse bug#115789	0	6	6	12	6	6
fop	2	2	2	35	2	1
pmd	0	0	0	11	0	0
ps	0	0	0	3	0	0
javac	0	0	0	71	0	0
ython	0	0	1	3	0	0
jess	0	1	2	9	1	1
antlr	0	0	0	9	0	0
bloat	0	0	0	33	0	0
jbb2000	0	4	4	10	4	4
jack	0	0	0	9	0	0
mtrt	0	0	0	3	0	0
raytrace	0	0	0	4	0	0
compress	0	0	0	4	0	0
db	0	0	0	2	0	0

Table 2. Number of types reported in at least 25% of garbage collection reports: (a) Varying the *decay factor* from Ratio Ranking Technique ($R_{thres}^t = 100$). We choose a decay factor $f = 15\%$. (b) Varying the *rank threshold* from Ratio Ranking Technique ($f = 15\%$). We choose rank threshold $R_{thres}^t = 100$.

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. It 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.

3.4 Achieving Accuracy

Cork’s accuracy depends on its ability rank and report growing types. We experiment with different sensitivities for both the decay factor f and the rank threshold R_{thres} . Table 2 shows how changing the decay factor changes the number of reported types. 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 only growing data structures in our benchmarks without any false positives. Table 2(b) shows how increasing the rank threshold eliminates false positives

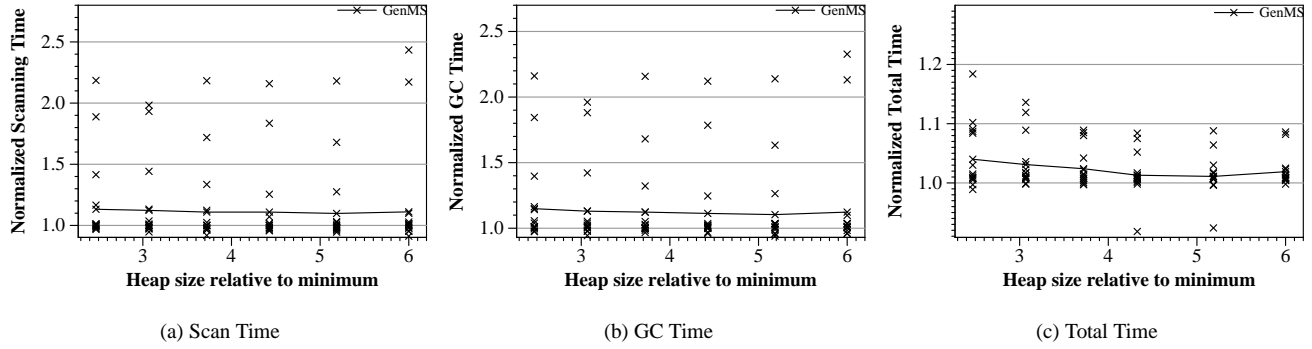


Figure 5. Geometric Mean Overhead Graphs over all benchmarks for generational collector

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. An extended technical report contains more analysis of both the decay factor and the rank threshold [16].

3.5 Finding and Fixing Leaks

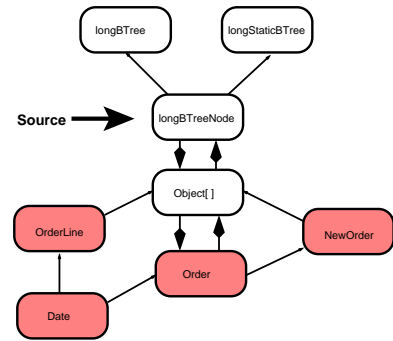
Cork identifies heap growth in four of our benchmarks: `fop`, `jess`, `SPECjbb2000` and `Eclipse`. For space constraints, we describe `SPECjbb2000` and `Eclipse` in detail. Descriptions of all benchmarks can be found in an extended technical report [16].

SPECjbb2000

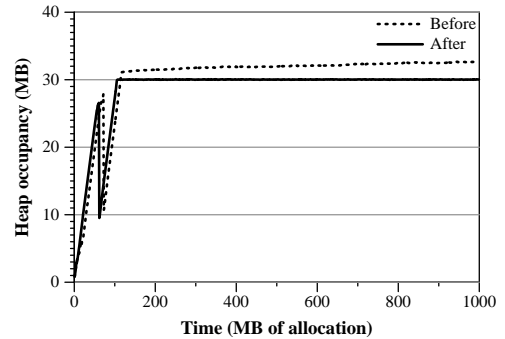
The `SPECjbb2000` benchmark models a wholesale company with several warehouses (or districts). Each warehouse has one terminal where customers can generate requests: e.g., place new orders or request the status of an existing order. The warehouse executes operations in sequence, with each operation selected 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 `BTree` and other data structures.

RRT analysis reports four candidates: `Order`, `Date`, `NewOrder`, and `OrderLine`. The rank of the four corresponding type nodes oscillates between collections making it difficult to determine their relative importance. Examining the slices of the four reported type 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 6(a) graphically shows the slice Cork reported (the shaded types are growing). Notice that despite the prolific use of `Object[]` in `SPECjbb2000`, its type node volume jitters to such a degree that it never shows sufficient growth to be reported as leaking. Since the slice includes all type nodes with $r_t > R_{thres}^t$ and reference edges with $r_e > 0$, the slice sees beyond the `Object[]` to the containing 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 remove the orders from the `orderTable`, we eliminate this memory leak. Figure 6(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.



(a) Slice Diagram



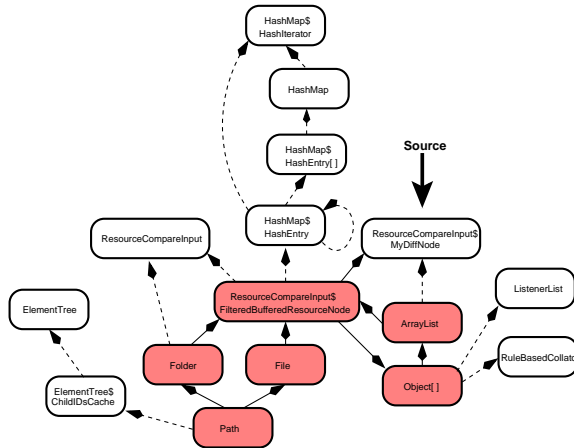
(b) Heap occupancy graph

Figure 6. Fixing `SPECjbb2000`

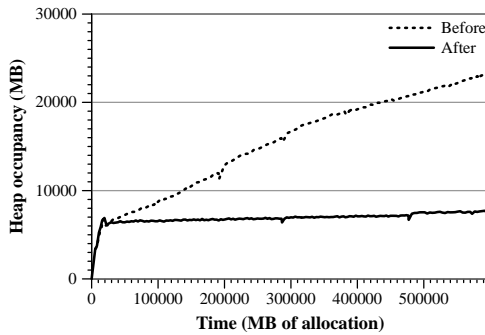
Eclipse

`Eclipse` version 3.1.2 is a widely-used integrated development environment (IDE) written in Java [30]. `Eclipse` is a good benchmark because it is big, complex, and open-source. `Eclipse bug#115789` documents an unresolved memory leak in the `Eclipse` bug repository from September 2005. We recreate this bug by repeatedly comparing the contents of two directories structures.

Cork reports six candidates: `File`, `Folder`, `Path`, `ArrayList`, `Object[]`, and `ResourceCompareInput$FilteredBufferedResourceNode`. Figure 7(a) shows the



(a) Slice Diagram



(b) Heap occupancy graph

Figure 7. Fixing Eclipse bug#115789

slices for the candidates, the close interrelationship between them, and several possible roots of the heap growth. Upon analysis, we eliminated several roots (indicated by dotted edges in Figure 7(a)) reducing the possible roots to one: a linked list created by `ResourceCompareInput$MyDiffNode`.

Correlating Cork’s results with the Eclipse implementation showed that upon completion, the differences between the two directory structures are displayed in the `CompareEditorInput` 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`. Figure 7(b) shows the heap occupancy graphs before 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. While we fixed the major source growth in the heap, there remains a small growth which at the time of this writing we have not investigated.

4. Related Work

The problem of detecting memory leaks is well studied. Compile-time analysis can find double free and missing frees [14] and is

complimentary to our work. Offline diagnostic tools accurately detect leaks using a combination of heap differencing [10, 11, 20] and fined-grained allocation/usage tracking [8, 13, 24, 25, 28, 29]. 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 type while concisely identifying the dynamic data structure containing the growth. Online diagnostic tools relies on detecting when objects exceed their expected lifetimes [19] and/or detecting when an object becomes *stale* [7, 9]. We improve on the efficiency of these techniques. However, they differentiate in-use objects from those not-in use, adding additional information to their reports.

The closest related work is Leakbot [12, 17, 18] which combines offline analysis with online diagnosis to find data structures with memory leaks. Leakbot uses JVMPI to take heap snapshots of-flooded 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 its analysis to another processor (heap differencing and ranking parts of the *object* graph which may be leaking), Leakbot minimizes the impact on the application and the perceived overhead of the analysis. Thus, Leakbot relies on an additional processor and multiple copies of the heap – a memory overhead potentially 200% or more that is proportional to the heap – to perform heap differencing and report object level statistics. Cork, on the other hand, summarizes object instances in a *TPFG* graph while minimizing the memory overhead (less than 1%) and allowing it to run continuously and concurrently with the application.

5. Conclusion

This paper introduces a novel and efficient way to summarize the heap to identify types 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 heap and reports slices of a summarizing type 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. 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.

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 `jbb2000`, and Mike Bond for his assistance with Eclipse.

References

- [1] 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 *OOPSLA ’99: Proceeding of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 314–324, Denver, Colorado, USA, November 1999.
- [2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA ’00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, Minnesota, USA, October 2000.

- [3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: The Performance Impact on Garbage Collection. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, New York, New York, USA, June 2004.
- [4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with JMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Scotland, United Kingdom, May 2004. IEEE Computer Society.
- [5] 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 *OOPSLA '06: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, Oregon, USA, October 2006. <http://www.dacapobench.org>.
- [6] 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>.
- [7] M. D. Bond and K. S. McKinley. Bell: Bit-Encoding Online Memory Leak Detection. In *ASPLOS XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, October 2006.
- [8] 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.
- [9] T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection using Adaptive Statistical Profiling. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, Boston, Massachusetts, USA, October 2004.
- [10] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution Patterns in Object-Oriented Visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234, Santa Fe, New Mexico, USA, April 1998.
- [11] 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.
- [12] 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.
- [13] 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, Berkeley, California, USA, January 1992.
- [14] D. L. Heine and M. S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, San Diego, California, USA, June 2003.
- [15] 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 *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80, Vancouver, BC, Canada, October 2004.
- [16] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Java. Technical Report TR-06-07, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, January 2006.
- [17] N. Mitchell, May 2006. Personal communication.
- [18] N. Mitchell and G. Sevitzky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *ECOOP 2003 – Object Oriented Programming: 17th European Conference*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377, Darmstadt, Germany, July 2003. Springer-Verlag.
- [19] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *HPCA-11: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, Cambridge, Massachusetts, USA, February 2002. IEEE Computer Society.
- [20] QuestSoftware. JProbe Memory Debugger: Eliminate Memory Leaks and Excessive Garbage Collection. <http://www.quest.com/jprobe/profiler.asp>.
- [21] N. Røjemo. Generational Garbage Collection without Temporary Space Leaks. In *IWMM 95: Proceeding of the International Workshop on Memory Management*, 1995.
- [22] N. Røjemo and C. Runciman. Lag, Drag, Void and Use – Heap Profiling and Space-Efficient Compilation Revised. In *ICFP '96: Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 34–41, Philadelphia, Pennsylvania, USA, May 1996.
- [23] C. Runciman and N. Røjemo. 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 Kingdom, August 1996. Springer-Verlag.
- [24] M. Serrano and H.-J. Boehm. Understanding Memory Allocation of Scheme Programs. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 245–256, Montréal, Québec, Canada, September 2000.
- [25] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic Removal of Array Memory Leaks in Java. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 50–66, London, United Kingdom, 2000. Springer-Verlag.
- [26] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.
- [27] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.
- [28] Sun Microsystems. Heap Analysis Tool. <https://hat.dev.java.net/>.
- [29] Sun Microsystems. HPROF Profiler Agent. <http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html>.
- [30] The Eclipse Foundation. Eclipse Homepage. <http://www.eclipse.org>.