The Dissertation Committee for Maria Eva Jump
certifies that this is the approved version of the following dissertation:

Discovering Heap Anomalies in the Wild

Committee:

Kathryn S McKinley, Supervisor

Stephen M Blackburn

Lizy K John

Calvin Lin

Dewayne E Perry

Emmett Witchel

Discovering Heap Anomalies in the Wild

by

Maria Eva Jump, B.S.; B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2009

To my husband.

Acknowledgments

It has been my great fortune and honor to have Kathryn McKinley as my advisor. Her guidance, insight, and undying enthusiasm provided the fuel for this project. Additionally, I wish to thank Steve Blackburn for his technical advice and intellectual commentary.

I wish to thank all the members of the DaCapo Group and the highly dedicated Jikes RVM development team. It was thanks to your work that I was able to do mine. Research cannot be done without funding and thus I wish to thank Intel and Microsoft for their direct and indirect support as well as the National Science Foundation.

My time in graduate school would not have been bearable without my fellow graduate students. I'd like to acknowledge all of those that kept me intellectually challenged who are far too numerous to list. I'd like to thank the computer science women. Thank you for the margaritas, the group lunches, the events, and the proof that computer science isn't only for men. Finally, amongst all of the people with whom I shared this experience, I'd like to thank Jennifer Sartor and Renee St. Amant. Thanks for all you have done.

I would like to thanks my family: my sisters and especially my Mom and Dad who have helped me in so many ways.

Finally, I dedicate this dissertation to my husband, Ted. Thank you for your unwavering belief, your undying love, and the unlimited hugs without which I would never had made it through.

Discovering Heap Anomalies in the Wild

Publication No. _____

Maria Eva Jump, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Kathryn S McKinley

Programmers increasingly rely on managed languages (e.g. Java and C#) to develop applications faster and with fewer bugs. Managed languages encourage allocating objects in the heap and rely on automatic memory management (garbage collection) to reclaim objects the program can no longer access. With more objects in the heap, the heap encodes more program state than ever before and offers new opportunities for optimization and analysis.

This dissertation shows how to efficiently leverage the managed runtime to perform dynamic heap analysis. Previous heap analysis approaches significantly slow down programs, require special hardware, and/or increase memory consumption by 75% or more. We presents two synergistic techniques—*dynamic object sampling* (*DOS*) and *heap summarization* (*HSG*)—that mine program state embedded in the heap efficiently enough to use in production and effectively enough to improve performance, find bugs, and increase program understanding. We use these techniques to address three problems: (1) **Performance of managed language.** Because some objects live for

a long time, they incur disproportionate collection costs. We optimize these costs with *dynamic pretenuring*. Dynamic pretenuring uses *DOS* to accurately predict allocation site survival rates and uses these predictions to improve performance. (2) **Finding bugs.** Memory leaks in managed languages occur when a program inadvertently maintains references to objects that it no longer needs. Along with degrading performance and resulting in program crashes, memory leaks cause systematic heap growth. We introduce *Cork* which uses the simplest type of *HSG*, a class points-from summary graph (*CPFG*), to detect systematic heap growth. Cork quickly identifies growing data structures observed in three popular benchmarks (fop, jess, and jbb2000) while adding an average of only 2.3% to total time. Additionally, we use Cork to debug a reported memory leak in Eclipse. (3) **Program understanding.** For a long time, static analysis has sought to statically summarize the shape of dynamic data structures to aid in program verification and understanding. Unfortunately, it only works on small programs. We introduce *ShapeUp* which instead characterizes recursive data structures dynamically by discovering data structure shape and degree invariants at runtime. ShapeUp uses *DOS* and a class field-wise summary graph (*CFSG*) to track in- and out-degree invariants of data structure nodes. We show how ShapeUp automatically identifies recursive data structures and likely shape invariants. Finally, we monitor discovered shape invariants to detect when a data structure becomes malformed.

In summary, this dissertation is the first to leverage the managed runtime to perform dynamic heap analysis both accurately and efficiently. Our results show that the heap contains an enormous amount of program state and that there is much potential for dynamically mining heap characteristics for optimization, debugging, and program understanding.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the demands on computer systems and software increase, so do their sophistication and complexity. Increasingly, programmers rely on managed languages that provide software engineering benefits to manage this growing complexity. A managed language provides (1) memory and type safety, (2) automatic memory management, (3) dynamic code execution, and (4) well-defined boundaries between type-safe and unsafe code [33]. The Tiobe Programming Community Index shows that managed languages (e.g., C#, Java, Ruby, and Python) continue to gain in popularity dominating unmanaged languages like C and C++ [177]. These object-oriented languages focus on data (or objects) rather than processes emphasizing discrete units of programming logic and code reuse. As a result, large groups of programmers can develop large systems faster and with fewer bugs. In this scenario, individual or small groups of programmers work only on small portions of a larger system using well-defined interfaces to interact with collaborators, libraries, and frameworks. The software-engineering benefits of this programming paradigm are well-documented, but it means that few programmers, if any, understand the entire system and how each part interacts with another making semantic, memory, and concurrency bugs harder than ever to find.

As a side-effect of object-orientedness and modularity, the size of objects has become smaller, while the number of objects has grown significantly

larger. Programmers use regular structures such as arrays and recursive data structures to manage the growing number of objects. Managed languages encourage allocating the majority of these objects in the heap and rely on automatic memory management (garbage collection) to reclaim objects that the program can no longer access. The heap encodes more program state than ever before, making it unsurprising that many semantic, memory, and concurrency bugs manifest there as well. Heap analysis must therefore become a necessary part of program analysis; we believe that it offers new opportunities for mining program state that can be used to aid in program understanding, bug detection, and program optimization.

1.1 The Problem

Previous approaches for heap analysis significantly slow down programs, require special hardware, and/or increase memory consumption by 75% or more making them inappropriate for use after deployment [44, 62, 63, 74, 86, 88, 92, 126, 129, 140, 148, 159, 160, 173, 174, 189]. Consequently they are appropriate during development and/or testing.

In development, static analysis discovers properties of an application both for debugging and optimization. Most static analysis treats the heap as an amorphous blob to keep costs tractable. The exception to this rule is *shape analysis* which tries to discover the shape of data structures using heap approximation and then tries to identify program statements that can potentially cause the heap to enter an erroneous state [80, 155, 156]. Unfortunately, existing static shape analyses are too expensive for even modestly sized programs.

Testing often consists of profiling an application through various offline

and online executions. In offline profiling, an application is profiled ahead of time. The results of profile runs are analyzed and then used to guide optimization in future runs. There are several problems with this methodology: (1) it is inconsistent with dynamic code execution; (2) profile data is only as good as the input since many program states depend on the program's input; and (3) offline summaries obscure phase changes in the program. All of these problems can be addressed by profiling online—during a actual run of the application. Depending on the characteristics being profiled, this approach can be very effective. Unfortunately, detailed profiling of every object is expensive and thus is only done during testing. While software testing improves software quality by exercising a wide range of program functionality, exhaustive testing is infeasible due to the size of the testing space. The result is that software often ships with bugs.

1.2 The Solution

When focusing on production runs, profiling techniques must be efficient enough that users are unaware that their executions of the program are also performing dynamic heap analysis. The presence of a managed runtime provides an opportunity to perform dynamic heap analysis efficiently. In this dissertation, we introduce such two techniques: *heap summarization* (*HSG*) and *dynamic object sampling* (*DOS*).

Heap summarization (*HSG*) compactly summarizes the heap during program execution using the object's class and/or data structure that containing each object. By exploiting the underlying runtime and piggybacking on the object scan performed during garbage collection, building *HSG*s add less than 1% to total memory allocation and less than 4% to total time on

average. The *HSG* is a family of graphs which summarize the dynamic nature of the heap: the objects that comprise the heap and the relations between them. For example, by capturing points-to relations in the edges, we show ownership properties. The lack of an expected ownership relation is evidence of potential semantic errors. Points-from relations in the edges allow us to identify liveness for a particular class of objects. Heap summaries are most cheaply generated using class, and we show that class-summarization efficiently identifies two classes of heap anomalies: systematic heap growth and dynamic invariant violations.

Summarization by context or data structure is more challenging since managed runtimes do not correlate objects with their context or data structure. One approach to correlate objects with their context, for example, is to add space to the objects header resulting in larger objects, more frequent garbage collections, larger program footprints, and slower runtimes—adding a single word increases the the space overhead by 5-7% and the time overhead by 8-18% on average. Dynamic object sampling (*DOS*) provides a technique for tagging a subset of objects to reduce overheads. For some applications, selecting which objects to tag is obvious (e.g., nodes of a recursive data structure); for others, it is more challenging. By focusing on fewer objects in the heap, we can significantly reduce the overheads of tracking objects characteristics. For example, sampling 6% of objects results in less than 1% space overhead and less than 3% time overhead.

We show the effectiveness of these techniques for performance improvement, debugging, and program understanding by addressing the following three problems.

Dynamic Pretenuring. Because some objects live for a long time, they incur disproportionate collection costs. *Pretenuring* is an optimization which reduces nursery copying costs in a generational garbage collector by allocating long-lived objects directly into the mature space. Blackburn et al. showed that allocation site was a good static predictor of lifetime [34]. We show that *DOS* can accurately estimate allocation-site lifetime at runtime and that a dynamic pretenuring mechanism which identifies and pretenures long-lived sites using dynamic lifetime estimates can improve performance. For a site with sufficient samples and a high survival rate, the collector modifies the allocation site to allocate subsequent objects directly into the mature space. To detect lifetime phase changes, we introduce *backsampling* which occasionally allocates these sites into the nursery in order to reexamine their survival rate. We examine a range of heuristics for the minimum number of samples, pretenuring thresholds, and backsample thresholds. Backsampling provides robustness to mistakes as well as adaptivity to phase changes. While many of our benchmarks showed degradations in performance due to a lack of opportunity, we show that when opportunity exists dynamic pretenuring can improve performance. In javac, we improve performance by 3% on average and by as much as 9% in a tight heap. These results indicate that heap analysis is useful for performance optimization.

Dynamic Memory Leak Detection. Managed languages rely on automatic memory management (garbage collection) to reclaim objects that a program can no longer access which avoids many memory-related errors found in other languages. Unfortunately, memory leaks persist in managed languages when a program retains references to objects the program never uses again. In the best case, these objects degrade program performance by increasing

memory requirements and collector workload. In the worst case, a growing data structure results in systematic heap growth which causes the program to run out of memory and crash. Furthermore, the allocation that causes the failure is almost never related to the source of the leak.

To debug memory leaks, we introduce *Cork*, an accurate, scalable, and low-overhead technique for identifying systematic heap growth that uses the simplest *HSG*, a class points-from summary graph (*CPFG*) to summarize, identify, and report systematic heap growth. We show that the *CPFG* provides both efficiency and precision. The nodes of the *CPFG* represent the volume of live objects of each class. The edges represent the points-from relationship between classes weighted by volume. At the end of the each collection, the *CPFG* completely summarizes the live-object points-from relationships in the heap. Comparing the *CPFG*s over time produces a dynamic slice which identifies the classes contributing most significantly to the systematic heap growth. By following the points-from edges, the *CPFG* identifies the growing data structure. We show that Cork efficiently provides us with enough information to detect systematic heap growth. Cork successfully identifies the only systematic heap growth in three of 14 benchmarks, including identifying the cause of a well-known but previously undiagnosed memory leak in SPECjbb2000. Cork also positively identified the cause of a documented memory leak in Eclipse allowing for a correction to be submitted back to the community. Corks accuracy, scalability, and efficiency make it the first system of its kind with overheads low enough to use in deployment showing dynamic heap analysis can identify bugs.

Dynamic Shape Analysis. In order to manage the growing number of objects in the heap, programs use typically regular structures, such as arrays and recursive data structures. Static shape analysis seeks to summarize the shape of dynamic data structures and identify the program locations which could potentially cause the heap to enter an erroneous state. Unfortunately, the cost of static analysis limits its usefulness to small programs. We introduce *ShapeUp* which improves programming understanding by performing dynamic shape analysis on the heap, by discovering invariants of recursive data structures, and by detecting data structure anomalies when these invariants are violated. ShapeUp uses a more complex *HSG*, a class field-wise summary graph (*CFSG*), which separates field data and stores both points-from and points-to relations. ShapeUp identifies recursive data structures and calculates degree invariants for the set of objects (or nodes) that make up the *recursive backbone* of the data structure. The recursive backbone is defined as the set of objects (e.g. instances of `Node`) linked by references in a regular pattern such that each smaller data structure is composed of a smaller or simpler instance of the same data structure. By focusing on the backbone, ShapeUp efficiently detects and reports errors that occur in data structures such as singly- and doubly-linked lists, binary trees, and hashmaps. For example, we show that ShapeUp discovers invariants during correct runs and then finds inserted errors on microbenchmarks. These results demonstrate that heap analysis is useful for generating heap assertions, finding flaws, and improving general program understanding.

1.3 Contributions

Heap analysis is a necessary part of program analysis. This dissertation is the first to leverage the managed runtime to perform dynamic heap analysis both accurately and efficiently. Its contributions include:

1. *Dynamic object sampling* (*DOS*), a mechanism for tagging individual objects, reduces the cost of adding information to objects by allowing client analyses to focus on a useful subset. *DOS* is useful for mining individual object characteristics for performance optimization and debugging.

2. Heap summarization graphs (*HSG*) which compactly summarize the heap during program execution. We demonstrate two different forms of the graph: a class points-from summary graph (*CPFG*) and a class field-wise summary graph (*CFSG*). Both summarize the heap by user-defined class. We piggyback building *HSG*s on the garbage collector which adds an average of less than 1% to total memory and less than 4% to total time. We show that *HSG*s are useful for program understanding and debugging.

3. We show how dynamic pretenuring uses *DOS* to optimize a generational garbage collector. We use *DOS* to dynamically determine allocation-site lifetime. With sufficient samples and high survival rates, the compiler dynamically pretenures long-lived sites, reducing collection costs.

4. We introduce Cork, a dynamic memory leak detector that improves software quality by accurately pinpointing parts of the heap that are growing without bound using the *HSG*. We show that Cork is accurate, scalable, and efficient enough to use after deployment.

5. We introduce ShapeUp, a dynamic shape analysis tool that improves program understanding and software quality by identifying recursive data structures and their degree invariants. We show that ShapeUp can detect errors introduced into several classes of data structures, including singly- and doubly-linked lists, binary trees, and hashmaps.

1.4 Summary

With the increased number of objects allocated in the heap, heap analysis is becoming a necessary part of program analysis. Until now, techniques for mining program state in the heap have been prohibitively expensive. This dissertation is the first to show that the managed runtime can be leveraged to efficiently and effectively perform dynamic heap analysis.

Chapter 2

Evaluation Methodology

We use the system and methodology described in this chapter for all the results in this dissertation. We describe our Java Virtual Machine in Section 2.1. Section 2.2 describes garbage collection and the different collectors we use. Section 2.3 describes our benchmarks and experimental methodology.

2.1 Jikes RVM

We implement our techniques in Jikes RVM (formerly Jalapeño), an open-source, high-performance Java virtual machine written almost entirely in a slightly extended Java [4]. Jikes RVM does not have an bytecode interpreter. Instead, a fast template-driven baseline compiler produces machine code when the VM first encounters each Java method. The adaptive compilation system then optimizes the frequently executed methods [3]. Using a timer-based approach, it schedules periodic interrupts. At each interrupt, the adaptive system records the currently executing method. Using a threshold, the optimizing compiler selects and optimizes frequently executing methods at increasing levels of optimization. Since the interrupts are not deterministic, the level of compiler activity and final code quality are non-deterministic.

For our results, we only use configurations that turn off assertion checking and that precompile as much as possible including key libraries and the op-

timizing compiler (the *Fast* build-time configuration). Jikes RVM uses MMTk, an efficient and composable memory management toolkit that implements a wide variety of high-performance garbage collectors using shared components [26, 27]. MMTk manages large objects (8K or larger) separately in a non-copy space, and puts the compiler and a few other system pieces in the boot image, an immortal space; otherwise, Jikes RVM objects share the heap with the application.

2.2 Garbage Collection

Automatic memory management, or *garbage collection*, is a technique for automatically reclaiming unreachable program data. It benefits programmers by freeing them of the responsibility of explicit memory management and removes four common sources of memory-related programming errors: (1) dereferencing a pointer to memory that the program has previously freed (*dangling pointers*); (2) losing a pointer to an object that the program neglects to free (*lost pointers*); (3) freeing a pointer to memory that has been previously freed (*double-frees*); and (4) calling the incorrect version of free (*alloc/dealloc mismatch*). Good collector performance is essential for good overall performance. In this work, we modify several garbage collectors including two whole-heap collectors and two generational collectors implemented in MMTk.

2.2.1 Whole-Heap Collectors

SemiSpace. The semi-space algorithm is a whole-heap collector that divides the heap into two equal-sized copy spaces [46]. It contiguously allocates into one and reserves the other (*copy reserve*) for copying into requiring more frequent collections. When the heap is full, the collector traces and copies live

objects into the other space and then swaps them. Collection time is proportional to the number of survivors. Throughput performance suffers because it repeatedly copies objects that survive for a long time, and its responsiveness suffers because it collects the entire heap every time.

MarkSweep. Mark-sweep is a whole-heap collection scheme that uses a segregated-fits free-list with lazy freeing and a tracing collector [134]. The allocator divides memory into blocks of same-size segregated chunks and allocates objects into the smallest size block in which they fit. The collection traces and marks the live objects using bitmaps. Tracing is exactly the same as for SemiSpace except that instead of copying, it marks a bit in a live-object bitmap. Tracing is proportional to the number of live objects and reclamation is incremental and proportional to allocation. MarkSweep's maximum pause time is poor and its performance also suffers from heap fragmentation and repeatedly tracing objects that repeatedly survive many collections. Like SemiSpace, it's responsiveness suffers because it collects the entire heap every time.

2.2.2 Generational Collectors

Generational organizations separate young objects from old and collect the younger objects more frequently [179]. The younger space (*nursery*) uses contiguous allocation which provides a performance benefit because it results in cheaper allocation and locality for contemporaneously allocated objects. Generational collectors perform well because they focus on collecting the nursery where young objects reside, thereby taking advantage of the *weak generational hypothesis* which states that younger objects die more quickly

and at a higher rate than older ones [127, 179].

For generational collectors, we use a well-performing 4 MB *bounded* bump-pointer nursery [26]. In MMTk, the *bounded* nursery takes a command line parameter as the initial nursery size and collects when the nursery is full. It reduces the nursery below the bound when the mature space becomes too full for the heap to accommodate a nursery full of survivors. When the nursery size falls below a lower bound (we use 256KB), it triggers a mature space collection.

GenCopy. This classic generational garbage collector divides the heap into a nursery space and a mature space [113]. The nursery space uses a bump-pointer allocator and a copying collector. When the nursery is full, it collects and promotes survivors into a copying mature space (SemiSpace). It collects the entire heap only when the mature space is full. By collecting the nursery more frequently and compacting survivors into the mature space, GenCopy can improve mutator locality. However, it repeatedly copies objects that survive a long time and requires a copy reserve.

GenMS. This hybrid collector uses a copying nursery combined with a mark-sweep mature space [113].

2.3 Architecture, Benchmarks, and Measurements

We perform all of our experiments on a 3.2 GHz 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 2.6.20.3.

We evaluate our techniques using microbenchmarks (described in detail in Section 6.3.1), the SPECjvm benchmarks, the DaCapo benchmarks, SPECjbb2000, and pseudojbb, a variant of SPECjbb2000 that executes a fixed number of transactions to perform comparisons under a fixed garbage collection load. Table 2.1 gives a description of each benchmarks.

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 three or six times that size. We report total application and collector performance. As noted by Eeckhout et al., adaptive compilation in Jikes RVM obscures application behavior in performance measurements [71]. For our overhead measurements, we factor out nondeterminism and compilation using *replay compilation* [105]. Replay compilation deterministically applies the optimizing compiler to frequently executed methods chosen by the adaptive compiler in previous (offline) runs. We factor out the adaptive compiler by running each benchmark multiple times. The first run uses replay compilation to give a realistic mixture of optimized and unoptimized code. Then we turn off compilation and flush all compiler objects from the heap. During the second run, we measure and report application performance. We perform separate statistics-gathering runs that accumulate overall and individual collection statistics.

Table 2.2 shows key characteristics using replay compilation and an infinite heap with a 4MB bounded nursery. The *total alloc* column in Table 2.2 indicates the total number of megabytes allocated. We list the ratio of total allocation to the live size and % *nrs srv* rate indicating the percentage of objects that survive a nursery collection in order to quantify garbage collection load. Finally we indicate the number of classes loaded.

SPECjvm	
compress	Implements file compression and uncompression in 5 iterations over the same five tar files. In every cycle, it allocates two large byte arrays, one for input and one for output
jess	An expert system which reads a list of facts about several word games from an input file and attempts to solve the riddles. It has many short-lived objects
raytrace	Raytraces a picture
db	Simulates a simple database management system with a file of persistent records and a list of transactions as inputs. It first builds the database by parsing the records file and then applies transactions to this set
javac	A Java compiler iterates four times over the same Java code
mtrt	Raytraces a picture by dividing the input file into sections and starting a working thread for every section. The problem size is small and only two work threads are created
jack	A commercial parser generator performs 16 iterations of building up a live heap structure and collapsing it again. No data survives between iterations
SPECjbb	
SPECjbb2000	Models a wholesale company. Warehouses process customer-generated requests based on a probability distribution.
pseudojbb	SPECjbb2000 modified to execute a fixed number of transactions for comparisons under a fixed GC load
DaCapo	
antlr	Parses and generates parser and lexical analyzer for grammar files
bloat	Performs a number of optimizations and analysis on Java bytecode files
chart	Uses JFreeChart to plot several line graphs and renders them as PDF
eclipse	Executes some (non-gui) jdt performance tests for the Eclipse IDE
fop	Translates an XSL-FO file into PDF file format
hsqldb	Executes a JDBC-like in-memory benchmark, executing a number of transactions against a model of a banking application
jython	Interprets a series of Python programs
luindex	Uses lucene to index a set of documents; the works of Shakespeare and the King James Bible
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of the Shakespeare and the King James Bible
pmd	Analyses a set of Java classes for a range of source code problems
ps	Reads and interprets a PostScript file
xalan	Transforms XML documents into HTML

Table 2.1: Benchmark Descriptions

Benchmark	total alloc (MB)	alloc/ live	% nrs srv	classes loaded	classes +VM
SPECjvm					
compress	105.4	16.8	6.6	157	1,749
jess	262.0	221.3	1.1	293	1,885
raytrace	133.5	35.1	3.6	177	1,769
db	74.6	8.8	14.6	149	1,732
mpegaudio	0.7	1.1	50.5	200	1,792
javac	178.3	24.8	25.8	302	1,894
mtrt	140.5	19.5	6.6	178	1,770
jack	270.7	292.7	2.8	202	1,794
SPECjbb2000					
pseudojbb	207.1	9.8	31.3	238	1,830
DaCapo					
antlr	237.9	248.8	8.2	307	1,899
bloat	1,222.5	195.6	6.0	471	2,063
chart	742.8	77.9	6.3	706	2,298
eclipse	5,582.0	186.0	23.8	1,023	2,615
fop	100.3	14.5	14.2	865	2,457
jython	1,183.4	8,104.0	1.6	886	2,478
hsqldb	142.7	2.0	63.4	355	1,947
luindex	201.4	201.7	23.7	309	1,901
lusearch	1,780.8	162.8	1.1	295	1,887
pmd	779.7	56.8	14.0	619	2,211
xalan	60,235.6	2,364.0	3.8	552	2,144

Table 2.2: Benchmark Characteristics

2.4 Summary

This chapter described the system and methodology we use to implement and evaluate our techniques for dynamic heap analysis. In the next chapters, we will detail those techniques and discuss their overheads.

Chapter 3

Dynamic Heap Analysis

This chapter shows how we exploit the managed runtime to efficiently and effectively gather fine-grained object-level statistics and summarize them in new and compact ways. We start by describing the opportunity presented by the managed runtime and then detail two techniques for leveraging it: dynamic object sampling (*DOS*) and heap summarization.

3.1 The Garbage Collection Opportunity

Garbage collection provides an opportunity to examine the heap that does not exist in non-managed languages. For example, Huang et al. improve performance by reordering objects during garbage collection to improve locality in the mature space [105]. In this work, we exploit the garbage collection's scan of live objects to dynamically aggregate statistics about live objects. During object scanning, the garbage collector detects all live objects by starting at program roots (statics, stacks, and registers) and performing a transitive closure through all the live object references in the heap. At the end of a full-heap collection, every live object in the heap will be scanned exactly once and aggregated statistics can present a complete view of the current state of the heap.

3.2 Dynamic Object Sampling

One approach to learn object properties is to add space to object headers. However, this results in larger objects, more frequent garbage collections, larger program footprints, and slower runtimes. For example, adding a single word increases the space overhead by 5-7% and the time overhead by 8-18% on average in our Java benchmarks.

By default, aggregated statistics are limited by *object knowledge*. Object knowledge are those characteristics that are inherent to an instance and include things like size, user-defined class, number of data fields, and number of outgoing references. For example, to aggregate statistics for an allocation site or data structure, the system would need to know which allocated site allocated each instance and/or to which data structure they belong.

We introduce *dynamic object sampling* as a method for increasing object knowledge for a subset of objects. Dynamic object sampling tags a subset of objects with a *sample tag* during allocation (or during copying) encoding otherwise unknown object characteristics (e.g., allocation site). Piggybacking on the garbage collector, we can now aggregate learned characteristics of tagged objects.

3.2.1 Tagging Samples

Bump-pointer allocators use monotonically increasing addresses within a contiguous region of memory by repeatedly incrementing (*bumping*) a pointer. This *fast path* of the allocation sequence uses only a few instructions including a test to check whether the allocation exceeds some boundary. Figure 3.1(a) illustrates this code sequence and Figure 3.1(b) shows an allocation block with some objects allocated into it. When the allocator exceeds the boundary, it

```
1 VM_Address alloc(int bytes) {
2   VM_Address oldCursor = cursor;
3   VM_Address newCursor = oldCursor.add(bytes);
4   if (newCursor.GT(limit))          // need more memory?
5     return allocSlow(bytes);
6   cursor = newCursor;
7   return oldCursor;
8 }
```

(a) Original bump pointer allocation

(b) Allocation block without samples

Figure 3.1: Bump Pointer Allocation

calls the *slow path* which determines, for example, whether the allocator needs to request more memory or if it should trigger a collection. A sufficiently large allocation region makes the *fast path* the common case, and executes the more expensive *slow path* infrequently.

Figure 3.2 illustrates dynamic object sampling by *sampling in time*. In this example, we show the effects of different sampling rates by selecting a random sampling of all objects allocated in the heap. Figure 3.2(a) reflects the changes in the code sequence and Figure 3.2(b) the changes in the allocation block where sampled objects are shaded. Notice that sampling adds no instructions to the most frequently executed *fast path* (compare lines 1-8 in Figures 3.1(a) & 3.2(a)). It does, however, introduce an intermediate path whose test succeeds every `SAMPLE_PERIOD` bytes of allocation. The allocator sets a bit in the object's header to indicate the sampled object and records a *sample tag* to the object. The sample tag encodes a characteristic otherwise unknown to the object. For example, to aggregate statistics based on calling context, the sample tag could encode a context identifier.

```
1 VM_Address alloc(int bytes, int siteID) {
2   VM_Address oldCursor = cursor;
3   VM_Address newCursor = oldCursor.add(bytes);
4   if (newCursor.GT(sampleLimit))   // need to sample?
5     return sample(bytes, siteID);
6   cursor = newCursor;
7   return oldCursor;
8 }
9 VM_Address sample(int bytes, int siteID) {
10  VM_Address rtn;
11  int required = bytes + SAMPLE_BYTES;
12  VM_Address newCursor = cursor.add(required);
13  if (newCursor.GT(limit)) {       // need more memory?
14    rtn = allocSlow(required, siteID);
15    if (rtn.isZero()) return rtn;  // we need to GC
16  } else {
17    rtn = cursor;
18    cursor = newCursor;
19    sampleLimit = roundUp(cursor, SAMPLE_PERIOD);
20  }
21  recordSample(rtn, bytes, siteID); // record sample
22  return rtn.add(SAMPLE_BYTES);     // skip object tag
23 }
```

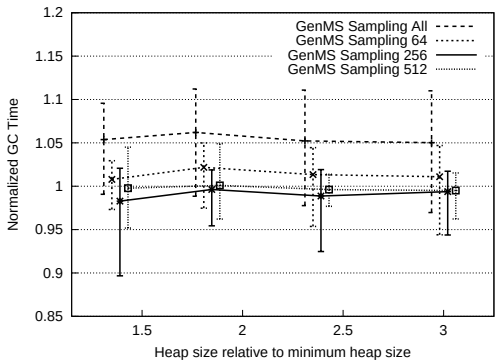
(a) Sampling bump pointer allocation

(b) Allocation block with samples

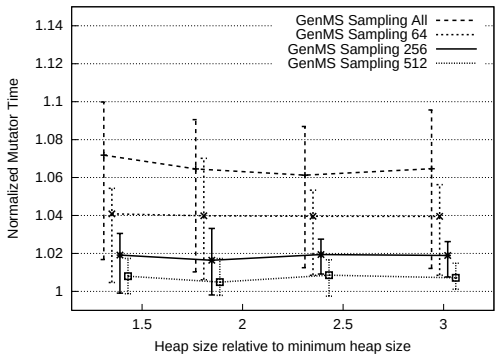
Figure 3.2: Bump Pointer Allocation with Dynamic Object Sampling

3.2.2 Sampling Overhead

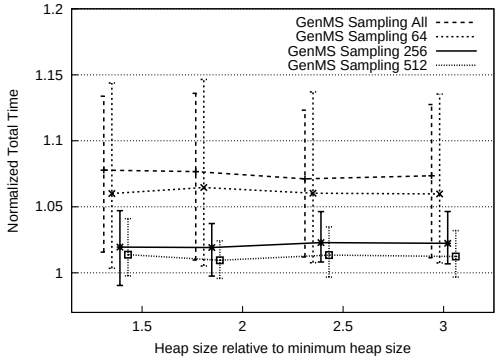
Next, we present sampling time and space overheads. Figure 3.3 shows the time overhead with sampling normalized to no sampling for a range of heap sizes using the geometric mean of our benchmarks. Bars show the variations for sampling every 512, 256, and 64 bytes, as well as all objects (representing 6%, 12%, 50% and 100% of objects respectively). The direct overhead of sampling has two components: the *spatial* overhead of a four byte site identifier and the *computational* overhead of periodically executing `sample()` (Figure 3.2(a), lines 10-25).



(a) GC Time



(b) Mutator Time



(c) Total Time

Figure 3.3: Sampling Overhead for Replay Compilation.

In this example, we add a four-byte site identifier to each sampled object. It increases space requirements by at most 0.8% for a 512 byte sample rate, and 1.6% for 256. The impact of this spatial overhead on garbage collection time is subtle, as shown in Figure 3.3(a). One would assume that the dominant cost would always be the additional work associated with collecting the nursery more frequently, but more subtle effects of perturbing collection trigger points can dominate. Changing when a collection occurs can have cascading positive and negative effects on promotion results and locality. For example, if the program is about to allocate some medium lifetime objects, an earlier collection gives them a chance to die and could avoid copying them in the next collection. Any change in the amount of allocation results in these effects [34]. Only when every allocation is sampled is the average collection time overhead significant (5% to 10%). For 256 and 512 byte sample rates the average collection overhead is negligible and is dominated by perturbations, which can produce up to 15% degradation and 10% improvement.

The mutator time overheads are very low averaging between 0.5% and 2% of mutator time for sample rates of 256 and 512. Figure 3.3(b) shows mutator time overhead and 3.3(c) shows the total time overhead. The bars show the worst case overheads and a few tiny improvements. For sampling at 256 and 512 bytes, these overheads range from -0.5% to 3.5%, and are slightly lower with the optimizing compiler. This mutator overhead includes the additional instructions required to sample, but also includes the effects of data and instruction locality, which presumably account for the tiny performance improvements. The overhead is still low on average (4% to 5%) for 64 bytes.

Sampling every object or sampling every 64 bytes represents approximately every two objects, since the average object size is about 32 bytes when

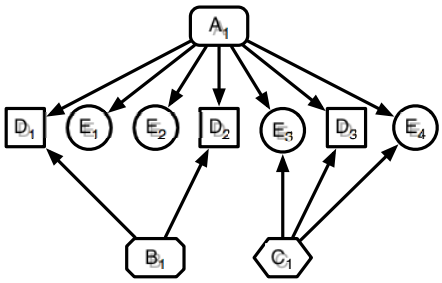
including Jikes RVM's 8-byte header [67]. When sampling every 32 bytes or all objects, overheads grow substantially, up to 18% worst case, but 7% to 9% on average. However, sampling rates of 256 and 512 (representing approximately 12% and 6% of all objects) are much more reasonable. Total time overheads range from 6% to less than -1% (noise). The average rate is between 1% and 3%, and is slightly higher using the optimizing compiler. By focusing our sampling on 6-12% of objects, we can collect significant object knowledge.

3.3 Heap Summarization Graphs

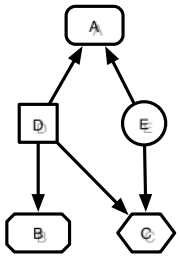
In addition to individual object characteristics, the heap also encodes relations between objects encoded in the references between objects in the heap. This section introduces heap summarization as a technique to aggregate object statistics and relation statistics in a compact summary graph (*HSG*). The *HSG* represents a family of graphs in which nodes of the graph summarize some object characteristic and edges summarize references either as points-from relations, points-to relations or both. For clarity of exposition, we describe building a *class points-from graph* (*CPFG*) in the context of a full-heap collector.

3.3.1 Building the Class Points-From Graphs

A *class points-from graph* (*CPFG*) consists of *class nodes* and *reference edges*. The class node represents all objects of class c. The reference edges are directed edges from class node c to class node c' and represent all of the objects of class c' that are referred to by an object of class c. Class c and class c' are not necessarily distinct. For example, figure 3.4(a) shows a heap consisting of an object points-to graph, i.e., objects and their pointer relationships, for objects



(a) Heap graph



(b) Class points-from graph

Figure 3.4: Heap summarization

```
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 3.5: Object scanning with additions on lines 4 and 9

of classes A, B, C, D, and E. Each vertex represents a different instance in the heap and each arrow represents a reference from one instance to another instance. Figure 3.4(b) shows the corresponding class points-from graph.

We collect summarization information by piggybacking on the live-object scan that occurs as a regular part of a tracing garbage collector. Figure 3.5 shows the modified scanning code from MMTk [26, 27]. Assume *scanObject* is processing an object of class A that refers to an object of class D from Figure 3.4(a). Building the *CPFG* requires two simple additions that appear at lines 4 and 9. It takes a reference to the tracing routine and the object as parameters, and finds the object class. Line 4 increments the node

for the class of instance D. Since the collector scans (detects liveness of) an object only once, each object instance increments the class node of this class only once. Next, *scanObject* determines if each referent of the object has already been scanned. As it iterates through the fields (slots), the added line 9 resolves the referent class of each outgoing reference ($A \rightarrow D$) and increments the appropriate edge ($A \leftarrow D$) in the graph. Thus, this step counts the number of edges for all references to an object, not just the first one encountered in the scan. Because this step adds an additional class look up for each reference, it also introduces the most overhead. Finally, *scanObject* enqueues those objects that have not yet been scanned in line 10. The additional work of the garbage collector depends on whether it is moving objects or not, and is orthogonal to building the *CPFG*.

At the end of scanning, the *CPFG* completely summarizes the live objects in the heap. Figure 3.4(b) shows the *CPFG* for our example. Notice that the reference edges in the *CPFG* point in the opposite direction of the references in the heap. Also notice that, in the heap, objects of class D are referenced by A, B, and C represented by the outgoing reference edges of D in the *CPFG*.

3.3.2 Implementation Efficiency and Scalability

We implement several optimizations to make implementation of *HSG*s scalable and efficient in both time and space. First, we limit the number of *CPFG*s stored, depending on the client. For example, many clients only require two *HSG*s: one representing the heap at the current collection and one aggregating statistics across multiple collections. Additionally, we piggybacks summary graph nodes on the VM's global *type information block* (TIB). This

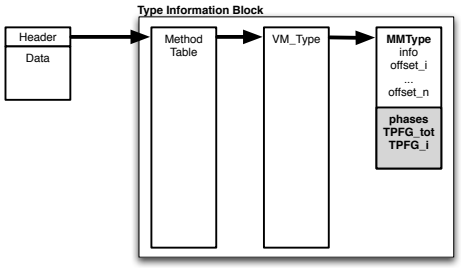


Figure 3.6: Modified Type Information Block (TIB)

structure or an equivalent is required for a correct implementation of dynamic dispatch and garbage collection in managed languages such as Java or C#. Figure 3.6 shows how *HSG*s modified the TIB from Jikes RVM. Notice that every live object of a class ($object_L$) points to the TIB corresponding to its class. The TIB consists of three different parts. The first is the method table which stores pointers to code for method dispatch for objects of this class. The method table points to a corresponding `VM_Type` which stores field offsets and type information used by the VM for efficient type checking and is used by the compiler to generate correct code. Finally, `VM_Type` points to a corresponding `MMType` used by the memory management system to perform allocation and to identify references during garbage collection. Recall from Figure 3.5 that object scanning resolves the `MMType` of each object (line 3). The *CPFG* class node data for each *CPFG* in the corresponding `MMType` adds a single word per each stored *CPFG*. We show in Chapters 5 and 6 how limiting the number of *CPFG*s to two is sufficient to aggregate statistics, such as volume, for a class. These class nodes scale with the type system of the VM.

While the number of reference edges in a *CPFG* are quadratic in theory, one class does not generally reference all other classes. Programs implement a much simpler class hierarchy and we find reference edges are linear with respect

Benchmark	# of classes avg	# of classes max	# edges per class avg	# edges per class max	# edges per CPFG avg	# edges per CPFG max
eclipse	667	775	2	203	4090	7585
fop	423	435	3	406	1559	2623
pmd	360	415	3	121	967	1297
ps	314	317	2	93	813	824
javac	347	378	3	99	1118	2126
jython	351	368	2	114	928	940
jess	318	319	2	89	844	861
antlr	320	356	2	123	860	1398
bloat	345	347	2	101	892	1329
jbb2000	318	319	2	110	904	1122
jack	309	318	2	107	838	878
mtrt	307	307	2	91	820	1047
raytrace	305	306	2	91	814	1074
compress	286	288	2	89	763	898
db	289	289	2	91	773	787
Geomean	342	357	2	116	1000	1303

Table 3.1: Class Points-From Statistics.

to the class nodes. This observation motivates a simple edge implementation consisting of a pool of available edges. New edges are allocated only when the edge pool is empty. When a *CPFG* expires, we return the edges to the pool for reuse. New edges are added to the *CPFG* by removing them from the edge pool and adding them to the list of reference edges kept with node data. We encode a pointer to the edge list with the node data which eliminates the need for adding any extra words to the `MMType` structure.

3.3.3 Space Overhead

The heuristics we introduced in Section 3.3.2 help keep the *CFSG* space efficient. Table 3.1 reports *CPFG* space overhead statistics. Columns one and two (*# of classes*) report the average and maximum number of classes with instances in the heap during any particular garbage collection. We notice that

	TIB		TIB+CPFG		
Benchmark	MB	%H	MB	%H	Diff
eclipse	0.53	0.011	0.70	0.015	0.167
fop	0.36	0.160	0.55	0.655	0.495
pmd	0.30	0.031	0.44	0.186	0.155
ps	0.28	0.029	0.39	0.082	0.053
javac	0.28	0.071	0.43	0.222	0.151
jython	0.28	0.041	0.39	0.112	0.071
jess	0.27	0.049	0.38	0.143	0.094
antlr	0.27	0.016	0.39	0.282	0.266
bloat	0.26	0.017	0.38	0.064	0.047
jbb2000	0.26	**	0.38	**	**
jack	0.26	0.042	0.37	0.131	0.089
mtrt	0.26	0.081	0.37	0.258	0.177
raytrace	0.26	0.085	0.37	0.272	0.187
compress	0.25	0.105	0.36	0.336	0.231
db	0.25	0.160	0.35	0.467	0.307
Geomean	0.29	0.048	0.41	0.168	0.145

Table 3.2: Class Points-From Space Overhead. **Volumes for SPECjbb2000 depend on how long the warehouse runs.

an average of 44% of all classes used by programs are present in the heap at a time.

Table 3.1 shows the average (column three) and maximum (column four) number of reference edges per class node in the *CPFG*. We find that most class nodes have a very small number of outgoing reference edges (two on average). The more prolific a class is in the heap, the greater the number of reference edges in its node (up to 406). We measure the average and maximum number of reference edges in any *CPFG* (columns five and six). These results demonstrate that the number of references edges is linear in the number of class nodes in practice.

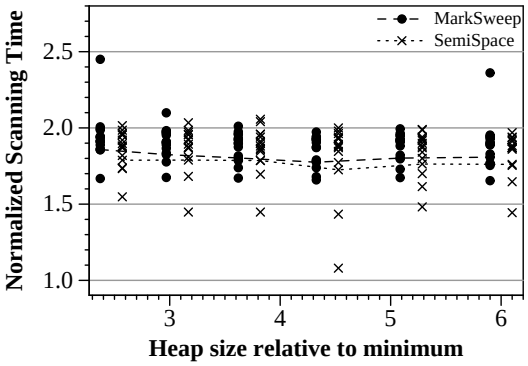
Table 3.2 shows the space requirements for the type information block before (*TIB*) and the overhead added by the *CPFG*. While the *CPFG* adds significantly to the TIB information, it adds only modestly to the overall heap

(0.145% on average and never more than 0.5% as shown in column 5). For the longest-running and largest program, eclipse, the *CPFG* has a tiny space overhead (0.004%).

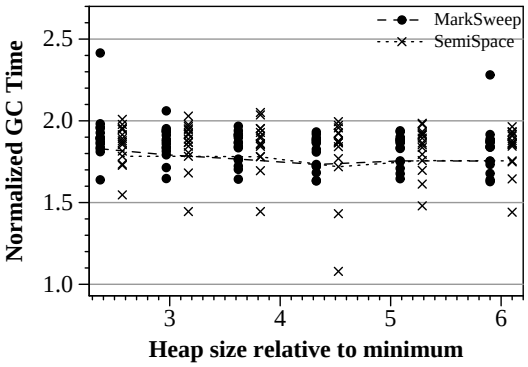
3.3.4 Performance Overhead Results

The *CPFG*'s time overhead comes from constructing the *CPFG* during scanning and from differencing between *CPFG*s to find growth at the end of each collection phase. Figures 3.7 and 3.8 graphs the normalized geometric mean over all benchmarks to show overhead in scan time, collector (GC) time, and total time. In each graph, the y-axis represents time normalized to the unmodified Jikes RVM using the same collector, and the x-axis graphs heap size relative to the minimum size each benchmark can run in a mark-sweep collector. Each x represents one program.

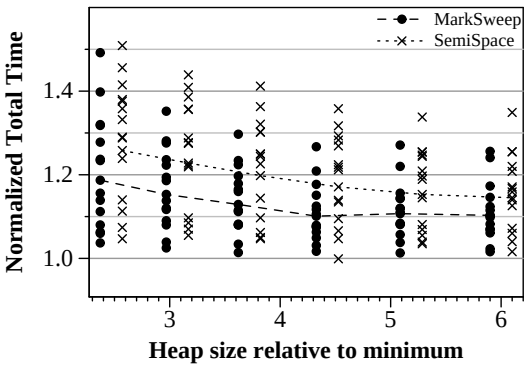
For mark-sweep (MarkSweep) and copying (SemiSpace) whole-heap collectors, Figure 3.7 shows that the scan time overhead is 80.8% to 85.5% and 76.1% to 78.8%; collector time is 75.4% to 82.9%; and total time is 10.3% to 25.8% respectively. These overheads represents the worst case since the entire *CPFG* is constructed every collection. Whole-heap collector overheads can be reduced by analyzing the heap every *nth* collection. Including the *HSG* in a high-performing generation collector with many less full-heap collections significantly reduces these overheads. Figure 3.8 shows the *CPFG*'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 the *CPFG*'s average overhead is low enough to consider using it online in production systems.



(a) Scan Time

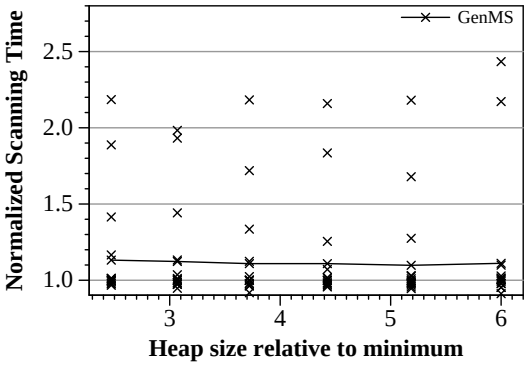


(b) GC Time

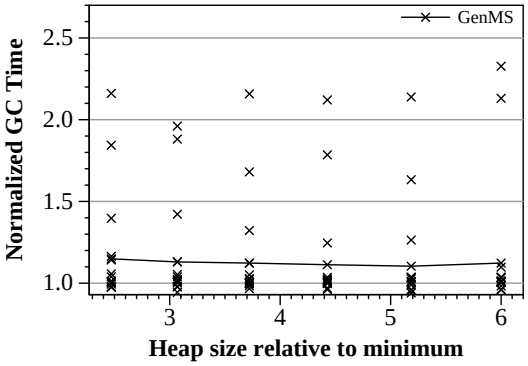


(c) Total Time

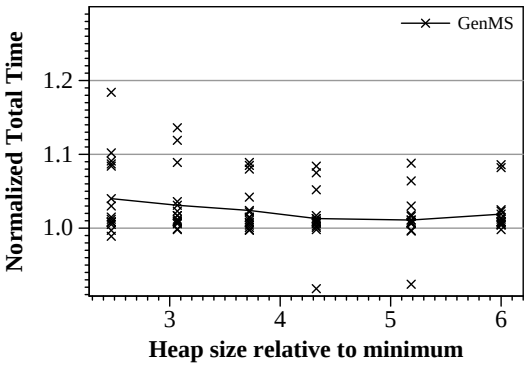
Figure 3.7: Geometric Mean Overhead Graphs over all benchmarks for whole-heap collector



(a) Scan Time



(b) GC Time



(c) Total Time

Figure 3.8: Geometric Mean Overhead Graphs over all benchmarks for generational collector

3.4 Summary

This chapter has introduced two synergistic techniques for performing dynamic heap analysis. These techniques are the first that are efficient enough to consider using after deployment. In the chapters that follow, we apply these two synergistic techniques to three problems: program optimization (Chapter 4), bug finding (Chapter 5), and program understanding (Chapter 6).

Chapter 4

Dynamic Pretenuring

Many state-of-the-art garbage collectors are generational, collecting the young *nursery* objects more frequently than old objects. These collectors perform well because young objects tend to die at a higher rate than old ones. However, long-lived objects incur a disproportionate collection cost since they always have to be copied out of the nursery. Pretenuring decreases nursery collection work by allocating new, long-lived objects directly into the mature space. This chapter shows how *DOS* can be used to mine the heap for allocation site lifetime. We examine some of the policies for exploiting lifetime information to dynamic pretenuring. While many of our benchmarks show degradations in performance due to the lack of opportunity, we show that when opportunity exists dynamic pretenuring can improve performance. In javac, we improve performance by 3% on average and as much as 9% in a tight heap. Further, we show that the allocation-site lifetimes calculated are both accurate (94% on average) and efficient (with overhead of 1% to 3% on average).

We start by presenting related work in Section 4.1 before describing how dynamic object sampling estimates allocation-site lifetimes in Section 4.2. We then use the dynamic lifetimes to develop heuristics for dynamic pretenuring detailed in Section 4.3.

4.1 Related Work

Much work has been done in determining object lifetime classification offline [21, 34, 48, 90, 158, 163, 180, 181]. In this work, a static profiler finds allocation sites for long-lived objects in a generational collector and recompiles the program to allocate these directly to mature generations [34, 48, 180, 181]. An offline, profile-driven approach is problematic for a just-in-time compiler due to dynamic class loading.

To determine and exploit lifetimes dynamically, previous work uses write barriers and weak pointers. Domani et al. and Qian and Hendren use write barriers to trap and differentiate global and local heap pointers [70, 146]. They then collect the local heaps independently. Qian and Hendren further redirect sites that allocate global variables into the global heap. Both of these techniques can add significantly to the execution time of the program, whereas our mechanism adds negligible overhead.

Agesen and Garthwaite sample objects by inserting weak pointers that identify the object allocation sites [2]. Their approach is most similar in spirit to ours. After a collection, they must trace both the dead and surviving sampled objects through the weak pointers to gather statistics. They do not report overhead separately, but as part of dynamic pretenuring. Total performance improves and degrades on average by 1% to 2%, but raytrace from SPECjvm98 degrades by 15%. We instead mark samples by their respective memory addresses. During collection, we need only track survivors. At the end of a collection, the allocation and survivor statistics completely specify lifetimes. These mechanisms reduce our space and collector time overheads compared to weak pointers. Both mechanisms require specialized allocation and collection support. Our object sampling is more general since it does not depend on

language support for weak pointers.

Harris uses Agesen and Garthwaite's sampling mechanism to make dynamic pretenuring decisions for Java programs in the context of a two-generation collector [91]. When the system detects a long-lived allocation site, it begins allocating into a mature generation. Harris' system samples in the higher generation to determine whether or not to reverse a decision, but the infrequency of higher generation collections reduces the accuracy of these lifetime samples. We instead allocate the occasional pretenured site back into the nursery. Harris notes that these objects will always survive if they are connected to another pretenured object, and in this case our mechanism would not yield useful samples. We find that this case does not occur frequently, and thus we can react more quickly to phase changes. Harris uses separate thresholds for pretenuring and reversal. He uses backpatching to change the allocation site, rather than a load to determine the allocation region. Neither technique recompiles the method. Harris uniquely identifies the allocation site without any call chain information. Because our JIT performs aggressive inlining, allocation sites in our system tend to have more context which Harris suggests should be useful. His dynamic pretenuring results show both improvements and a few significant degradations but are limited to a single heap size. We find improvements in a wide range of heap sizes while using a faster collector, providing a more general mechanism, and incurring lower overhead.

Huang et al. compute per-class rather than per-site allocation and survival statistics. Per-class is easier to implement since each object header includes the type already [104]. However, type is not a good predictor of lifetime. They use Jikes RVM's baseline compiler which does not produce high quality code and thus can hide any overhead. We use the adaptive optimizing

compiler. Their approach degrades total execution time slightly for the two of three SPECjvm benchmarks they test—jess and jack—while improving javac by only 2% to 5%.

4.2 Estimating Lifetimes

In order to estimate lifetimes of allocation sites, the most accurate system would track all allocation sites, but as we showed in Section 3.2, this approach is too expensive. We instead *sample in time*. We measure time in the number of bytes allocated and sample an object every n bytes of allocation. For efficiency, we define n as a power of 2. The *sample tag* encodes an allocation site identifier. We depend on the compiler to create unique identifiers for each allocation site. At the time of allocation, we tag each sampled object and increment an *allocation counter*. This mechanism samples larger objects more frequently since they disproportionately cross allocation boundaries. Thus sampling yields more accurate statistics for large objects. As Harris points out, large objects are important—especially if they are prolific [91].

As the garbage collector traces live objects, we look for the sample bit. If the sample bit is found, we know that the survivor is a sampled object. We then decode the sample tag, extract the allocation site identifier, and increment a *survivor counter* for that site. At the end of each garbage collection cycle, we have complete survival information. We use it to calculate the *survival rate* (SR) of each allocation site where survival rate is defined as:

$$SR = \frac{\#survivors}{\#allocated}$$

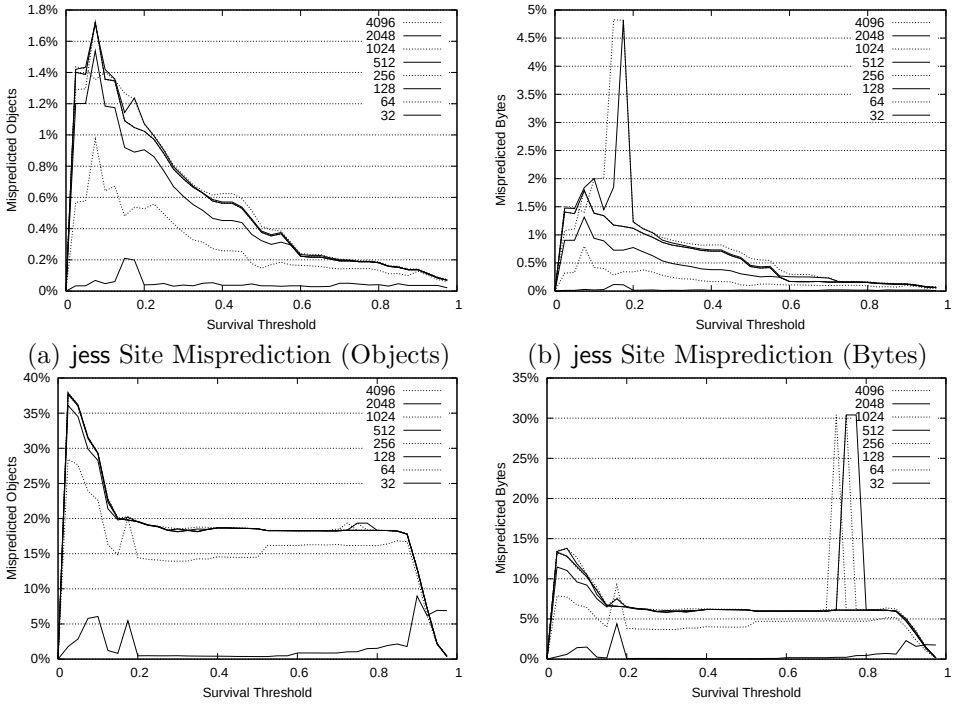
for surviving objects.

4.2.1 Lifetime Sampling Accuracy

To evaluate the error introduced by sampling when computing nursery survival rates, for each allocation site in a program, we establish the actual survival rate (the number of surviving objects over the total number of objects allocated) and the survival rate predicted with sampling. Depending on the particular demographics of the subset of objects at that site which were sampled, the survival rate can be overestimated or underestimated.

We use a *survival threshold* to classify sites as either short-lived or long-lived. For example, one might classify all sites with a survival rate higher than 80% as long-lived, and all others as short-lived. We then measure sampling accuracy in terms of *site misprediction*. At each site, we determine for each sample rate whether the *sampled survival rate* would cause the site to be classified differently from the site's *actual survival rate*. We then quantify this misprediction by summing for a given survival threshold the objects and bytes allocated at mispredicted sites. We repeat this process for a range of survival thresholds from 0 to 1.

Figure 4.1 plots site mispredictions for jess and javac, which are representative and diverse. The x-axis varies the survival threshold and the y-axis plots the level of misprediction for all sites that are mispredicted at a given survival threshold. Consider a site with an *actual* survival rate of 20% and a *predicted* survival rate of 80%. For survival thresholds between 0% and 20%, the site would be *correctly* classified as 'long lived' (both predicted and actual survival rates are greater than the threshold). However, for thresholds between 20% and 80%, it would be *incorrectly* classified as 'long lived' (the predicted survival rate is greater than the threshold, but the actual survival rate is not). For thresholds above 80% it would be *correctly* classified as *short*



(a) jess Site Misprediction (Objects)

(b) jess Site Misprediction (Bytes)

(c) javac Site Misprediction (Objects)

(d) javac Site Misprediction (Bytes)

Figure 4.1: Site Mispredictions as a Function of Survival Threshold and Sample Interval

lived (both predicted and actual survival rates are lower than the threshold). All objects and bytes allocated at this site would be included in all points on the site misprediction curve between the 20% and 80% survival thresholds. A site that was perfectly predicted would never contribute to the site misprediction curve. The figures plot site mispredictions with respect to objects (Figure 4.1(a) & (c)), and bytes (Figure 4.1(b) & (d)) of allocation associated with each mispredicted site. For example, sampling is less accurate for sites that have short-lived objects (survival rates between 5% and 15%) on both programs, compared with predicting objects with long lifetimes (rates above 60%).

The figures include one line for each sample rate from 32 bytes through to 4KB. Figure 4.1(a) shows that the level of misprediction in jess is very low ($< 2\%$), even at coarse sample rates. As the threshold grows, mispredictions become even less common. This trend reflects that jess has mostly low survival sites. Figure 4.1(b) measures site mispredictions in bytes and has almost the same curve as (a), but with a considerable spike just before the 20% survival threshold. Note that only the two coarsest sample rates experience this spike. This spike reflects a site that allocates large objects with a survival rate a little under 20%.

Figure 4.1(c) shows site misprediction for javac accounts for nearly ten times as many objects (around 20% of all allocations). The level of misprediction in javac is almost the same for all sample rates greater than 32, suggesting that the mispredicted sites are predominately allocating objects < 64 bytes in size. The flat curve illustrates a 20% misprediction rate between survival thresholds of 20% to 90%. Figure 4.1(d) shows that as a percentage of bytes, mispredictions are much lower, only around 6%. Mispredicted sites thus tend to allocate very small objects. javac has a sensitive spot at around 75% where the very coarsest sample rates see substantial degradation. This result suggests a site or sites allocating large objects with around 75% survival rate.

Overall, sample rates between 128 and 1024 bytes have high accuracy, with typical site misprediction rates in bytes ranging from nearly zero to around 6%. For these benchmarks with few long-lived objects, their behavior follows jess; i.e., mispredictions are highest at the lowest thresholds.

This section shows that *DOS* can accurately predict allocation-site lifetimes which can aid programming understanding. Next, we show how we can exploit this information to optimize a generational garbage collector.

4.3 Dynamic Pretenuring

Dynamic pretenuring seeks to eliminate the cost of copying long-lived objects out of the nursery in a generational collector by allocating them directly into the old space. To select candidates for dynamic pretenuring, we compute transient and aggregate lifetime statistics using *DOS*. At the end of each nursery collection, we compute survival rates for this collection (*transient*) as well as aggregate statistics. We only use one collection phase for transient statistics in our experiments. This separation focuses on those sites that changed in the last allocation phase (those with non-zero transient entries). For a site with sufficient samples and survival rate, dynamic pretenuring starts to allocate objects from that site into the old space after the first garbage collection.

Pretenuring policies can use either transient or aggregate statistics. To react quickly to phase changes, we use transient statistics to begin pretenuring any site with a survival rate exceeding a threshold t_s. This policy is very aggressive and introduces some errors but quickly captures newly allocating sites producing long-lived objects.

4.3.1 Dynamic Allocation Targets

In order to act on pretenuring decisions, we add a dynamic test to the allocation sequence shown in Figure 4.2. It uses a two instructions, an array lookup (line 12)[1] and a conditional branch (line 3). This implementation is simple and easily generalizes. When the optimizing compiler inlines the entire allocation sequence and then optimizes it in context, the overhead of this additional test is on average 1% to 2%.

[1]Our implementation uses a special instruction that avoids the array bounds check.

```
1 ...
2   case NURSERY_SPACE:
3     region = nursery.alloc(isScalar, bytes);
4     break;
5 ...
```

(a) Original allocation

```
1 ...
2   case NURSERY_SPACE:
3     if (DynamicPretenure.nurseryAlloc(site))
4        region = nursery.alloc(isScalar, bytes, site);
5     else
6        region = matureAlloc(isScalar, bytes, site);
7     break;
8 ...
9
10 public final static boolean nurseryAlloc(int site)
11   throws VM_PragmaInline {
12   return pretenureTable[site] >= 0;
13 }
```

(b) Allocation with Dynamic Test

Figure 4.2: Dynamic Test Added to the Allocation Sequence

Another approach would be to modify the code by backpatching the allocation instructions which completely removes allocation time overhead. We originally implemented this approach, but found that it was a premature optimization. The backpatcher was complex and extremely brittle as it had to parse and manipulate instruction sequences generated by an aggressive optimizing compiler, and the nature of those sequences was different between platforms and subject to change as the code in the allocation sequence and the compiler evolved. We ultimately chose this simpler approach since it is very robust, and although the overhead is not zero, it is very low.

4.3.2 Backsampling

Once the system decides to pretenure a site, allocating its objects into the mature space, the sampler can no longer compute that site's nursery survival rate. If the decision was wrong or the application behavior changes,

the system would never know. To avoid this pathology, we use *backsampling*. Backsampling periodically allocates pretenured sites back in the nursery until the next garbage collection, thereby providing an opportunity to reassess the site's survival rate.

We experiment with different policies that vary the frequency of backsampling, based on the backsampled transient and total survival rates. We implement backsampling by initializing the site's *mature counter*, called the *backsampling target*, to the negative of the total number of allocations used in making the pretenuring decision. Each time an object is allocated into the mature space, the mature counter is incremented, and when it reaches zero, the site allocates into the nursery for one allocation phase. If, at the next collection, the survival rate is no longer high enough, the system reverses the pretenuring decision. Otherwise, the system reinforces the pretenuring decision by changing the backsampling target according to one of the following heuristics:

- The constant heuristic (cbs) leaves the backsampling target as the number of allocations used to make the original decision (n).
- The linear heuristic (lbs) makes it harder to backsample the site by initializing the backsampling interval to a multiple ($f \geq 1$) of n where f grows linearly with each consecutive agreeing decision.
- The exponential (ebs) heuristic, f grows exponentially as a power of 2.

The constant heuristic (cbs) backsamples the most frequently, the linear (lbs) less than constant (cbs), and the exponential (ebs) backsamples the least frequently. Backsampling is a conservative mechanism. It reduces the effective-

Parameter	Description
Minimum Samples	The minimum number of samples required from an allocation site during an allocation phase for the site to be considered for pretenuring
Pretenuring Threshold	The survival rate above which a site is pretenured
Backsampling Shift	The backsampling trigger as a function of the number of objects used to make the pretenuring decision
Backsampling Policy	The backsampling functions include constant (*cbs*), linear (*lbs*), and exponential (*ebs*)
Decay Shift	The amount by which mature statistics should be decayed

Table 4.1: Policy Parameters

	Pretenuring Policies		
parameter	80 LBS	80 EBS	85 LBS
minimum samples	8	4	10
pretenuring threshold	80%	80%	85%
backsampling policy	linear	exponential	linear
backsampling shift	1	4	1
decay shift	0	0	1

Table 4.2: Configuration Settings for Base Results

ness of good choices, but protects the system from sampling errors and changes in lifetime phase behavior.

4.3.3 Pretenuring Policies

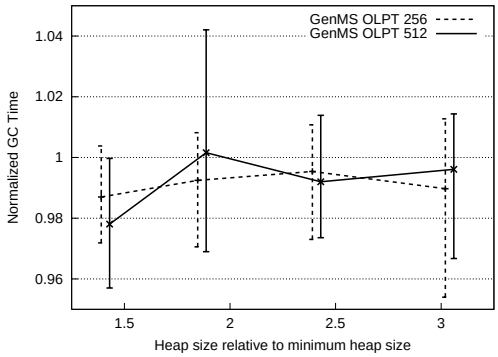
The dynamic pretenuring heuristic depends on five parameters which are detailed in Table 4.1. The backsampling shift, backsampling policy, and decay shift are used to define the backsampling trigger. We explore the space defined by these parameters to find good configurations and report the three with the best performance for javac. Table 4.2 shows these configurations. One feature that we did not vary is the use of transient or aggregate statistics: these results always use transient statistics. Aggregate statistics are more conservative and would probably reduce some errors.

4.3.4 Pretenuring Potential

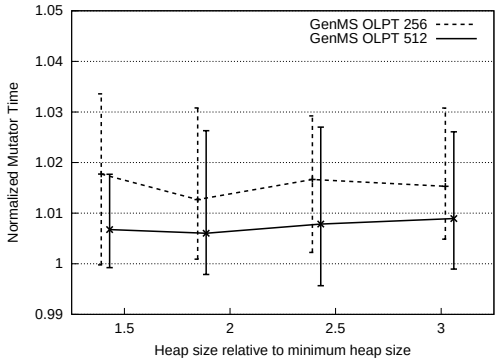
Before we present pretenuring results, we examine the potential that dynamic pretenuring has for improving the performance of our benchmarks. In Table 2.2, the % *nrs srv* indicates the percentage of objects that survive a nursery collection. This percentage indicates the potential for dynamic pretenuring to eliminate unnecessary copying. Only three of our programs are likely to benefit from pretenuring: pseudojbb, javac, and db. In particular with a 4MB nursery, pseudojbb and javac perform 50 and 53 nursery collections (respectively), whereas db performs 20. However, closer examination of db and pseudojbb show pretenuring is unlikely to improve them. In pseudojbb, only a few allocation sites produce the majority of long-lived objects, but these same sites produce many short-lived objects as well. Thus, these sites never produce survival rates high enough to benefit from pretenuring without more calling context than we examine here. In db, all the long-lived objects are allocated in the first 8MB of allocation. Dynamic pretenuring misses these opportunities while it is warming up. Thus, only javac has a potential for benefiting from dynamic pretenuring.

4.3.5 Overheads

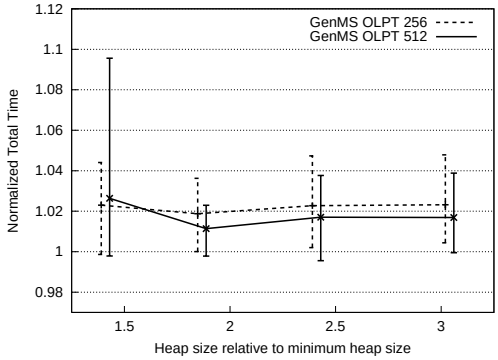
Figure 4.3 reports the overhead for sampling and the dynamic allocation test. We measure these overheads by running dynamic pretenuring in a configuration which will not actually pretenure by setting the pretenuring threshold above 100%. Factoring out the compiler and using the optimizing compiler on the adaptive pretenuring allocation sequence lowers its average overhead by 2% to 3%, but increases the variation from a range of 8% to -10%, to a range of 12% to -15%. The compilation differences again reflect that



(a) GC Time

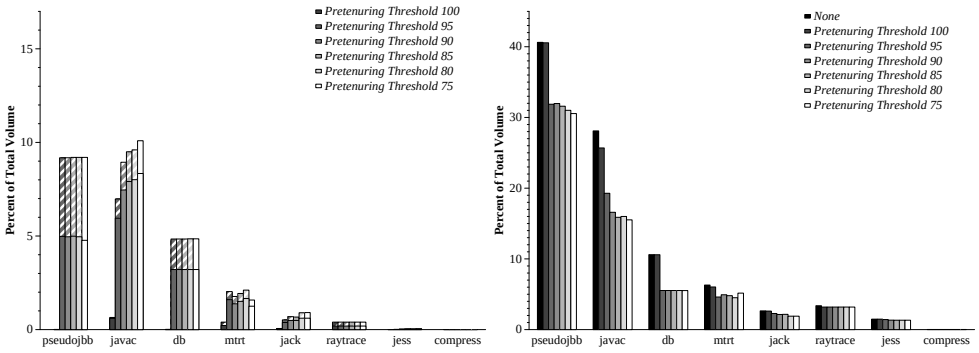


(b) Mutator Time



(c) Total Time

Figure 4.3: Average Sampling and Pretenuring Overhead for Replay Compilation for SPECjvm98



(a) Pretenured Objects, Correct (solid) and Incorrect (striped)

(b) Long Lived Objects *Not* Pretenured

Figure 4.4: Accuracy (a) and Coverage (b) of Pretenuring in Percent of Total Volume

a fully optimized setting exposes any overhead more completely.

4.3.6 Accuracy and Coverage

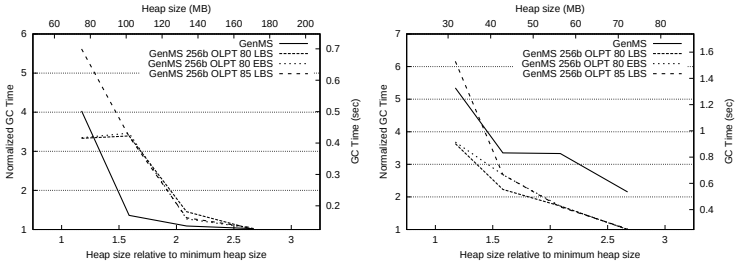
We now quantify the *accuracy* and *coverage* of pretenuring decisions in bytes of allocation. Accuracy measures the volume of objects chosen for pretenuring that were actually long-lived. Coverage measures the volume of long-lived objects that were chosen for pretenuring. Accuracy can be high while missing opportunities (low coverage).

Figure 4.4(a) shows accuracy, and 4.4(b) shows coverage for each benchmark and a range of pretenuring thresholds. We assume an infinite mature space, thus the nursery is always 4MB. This configuration therefore examines accuracy without cascading the penalty of mistakes. The other parameters values are the same as for 80 LBS from Table 4.2.

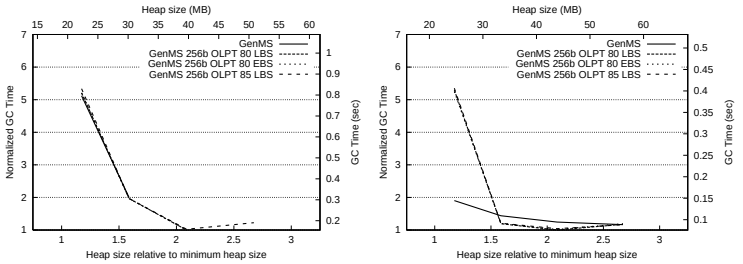
In Figure 4.4(a) the height of the bars represents the total volume of pretenured objects. The solid portion shows long-lived objects (e.g. *correctly*

pretenured), and the striped portion indicates short lived objects (e.g. *incorrectly* pretenured). Pretenuring accuracy is 80% or better for javac which is the only benchmark we speed up. The error rate for db is 34%, and even worse for pseudojbb at just under 50%. We expect errors to grow with a lower threshold because while decisions are per-allocation site, here we measure individual objects. For example, if a site has an 80% survival rate and is pretenured, the 20% of objects which are short lived at that site will be incorrectly pretenured. These results barely show this trend, thus the errors are most likely due to sampling errors (recall that we used a 256 byte sample rate) and from heterogeneous allocation lifetime phases. Error rates are higher for mtrt, jack, raytrace, and jess, but the pretenuring volume is extremely low.

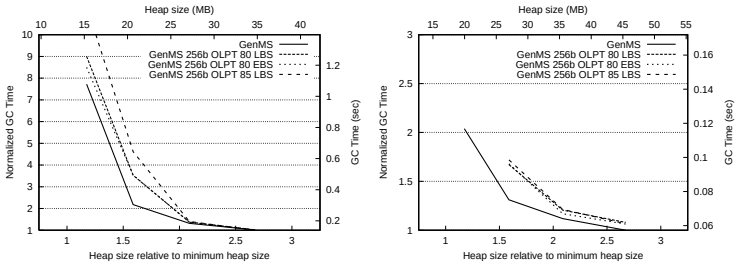
Figure 4.4(b) shows coverage; each bar represents the volume of long-lived objects that are *not* pretenured under different pretenuring regimes. The first bar, 'None', shows the volume when no pretenuring is performed, and therefore reflects the total volume of long-lived objects. The remaining bars vary the pretenuring threshold, although only pseudojbb, javac, db, and mtrt show any sensitivity to the threshold. The proximity of these bars to the 'None' bar shows dramatic under-pretenuring. At best we see 43% coverage (in javac with 75% pretenuring threshold), but in most cases the coverage is much lower. Reasons for under-pretenuring include objects missed during 'warm up' of the sampling mechanism, objects missed due to the sample rate, and lack of homogeneity at allocation sites (long-lived objects allocated from predominantly short-lived sites). Notice less than 5% of allocation is long lived for the remaining programs.



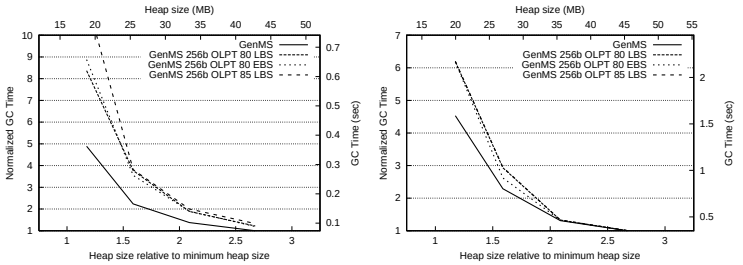
(a) pseudojbb (b) javac



(c) db (d) mtrt

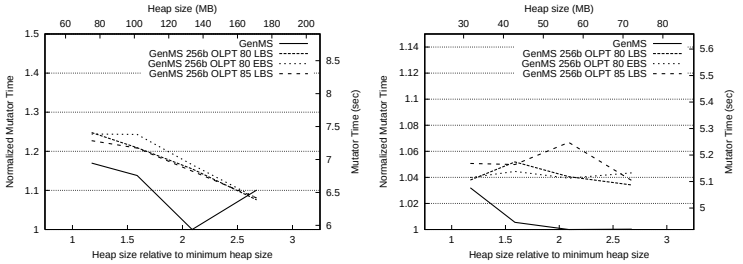


(e) jack (f) raytrace

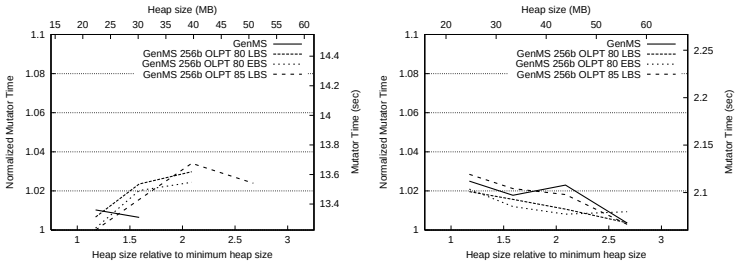


(g) jess (h) compress

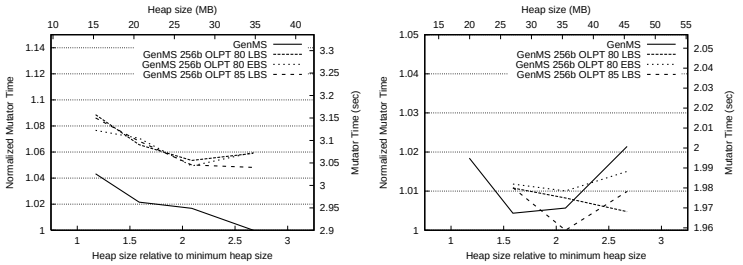
Figure 4.5: Garbage Collection Time



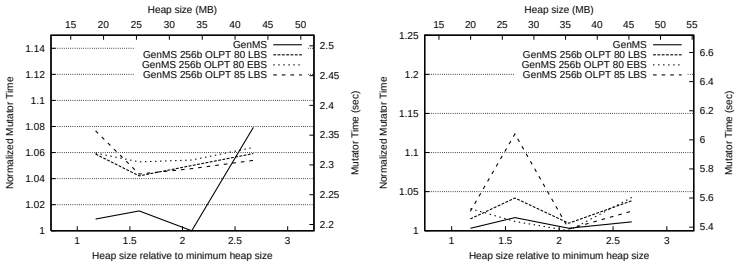
(a) pseudojbb (b) javac



(c) db (d) mtrt

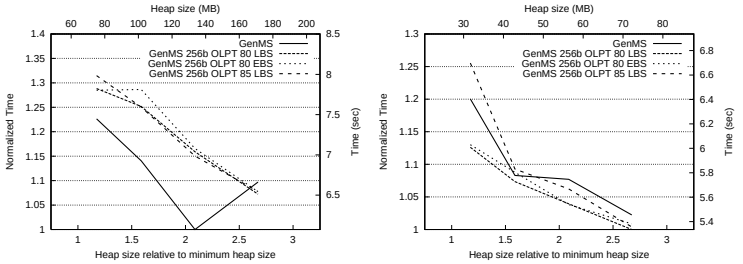


(e) jack (f) raytrace



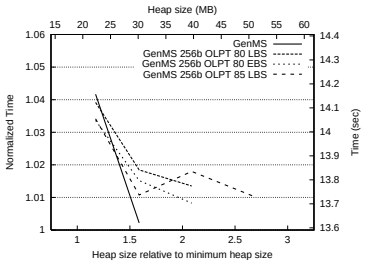
(g) jess (h) compress

Figure 4.6: Mutator Time

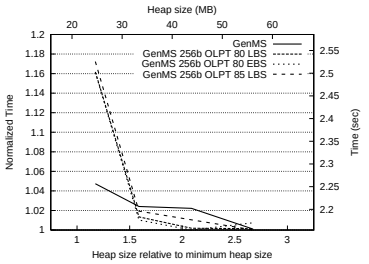


(a) pseudojbb

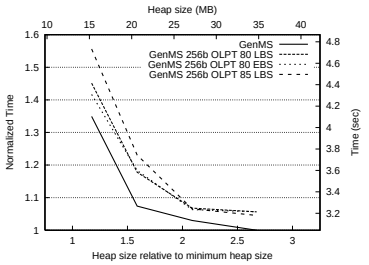
(b) javac



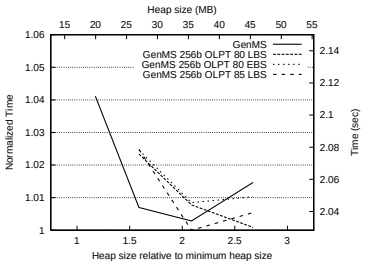
(c) db



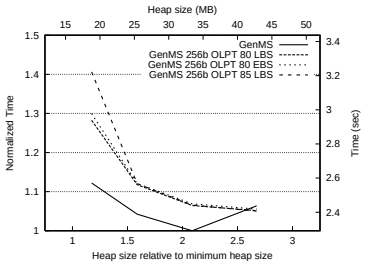
(d) mtrt



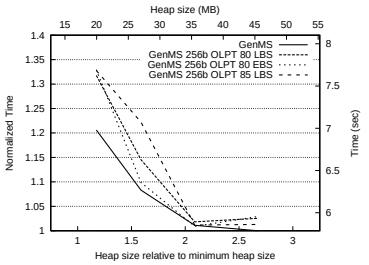
(e) jack



(f) raytrace



(g) jess



(h) compress

Figure 4.7: Total Execution Time

4.3.7 Performance

Finally, we report garbage collection (see Figure 4.5), mutator (see Figure 4.6, and total (see Figure **??**) time results for dynamic pretenuring for the configurations reported in Table 4.2.

The goal of pretenuring is to reduce garbage collection load. Only javac shows a systematic reduction in garbage collection, where improvements are as much as a factor of 2.5. We see some modest improvements in mtrt. db and jack are not significantly changed by pretenuring, and the remaining benchmarks all see degradations. In a small heap, erroneously pretenured objects needlessly occupy the mature space, reducing the bounded nursery size and triggering expensive full-heap collections. The pretenuring configurations also show sensitivity to heap size; 85 LBS is particularly bad in a small heap, but performs in large heaps.

The improvements in collection time for mtrt and javac translate to total time in Figure 4.7. The total time of javac improves by around 3% on average and by as much as 9% in a tight heap. mtrt improves by around 2% but degrades significantly in a tight heap. All other benchmarks show degradations in total time. Interestingly, javac shows a greater degradation in mutator time, presumably due to locality degradations caused by the disruption of allocation order that follows from a relatively high pretenuring rate.

4.4 Summary

This chapter introduced dynamic object sampling (*DOS*) for lifetime prediction. It showed that dynamic object sampling can sample and accurately predict object lifetimes for very low overhead. We explore using predicted

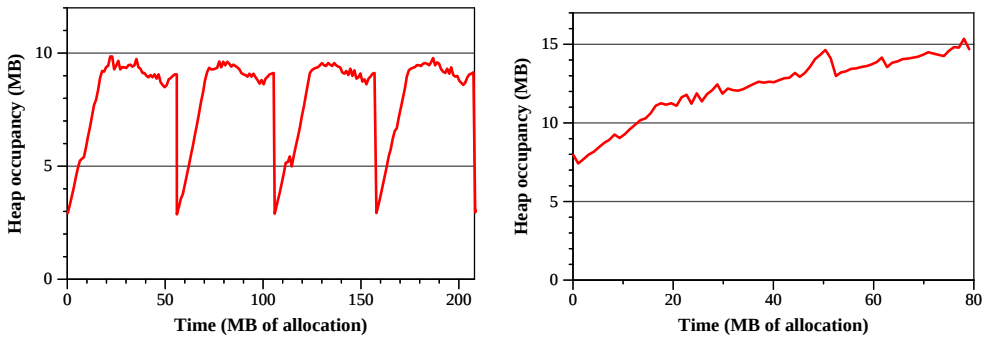
lifetime to improve the performance of a generational collector with limited success on one program. We show that the limited success is mainly due to a lack of potential in the programs tested. We leave open the question of policies that will make pretenuring more profitable. However, *DOS* is accurate and efficient enough to correctly and cheaply track object lifetime and other properties.

Chapter 5

Dynamic Memory Leak Detection

Memory-related bugs are a substantial source of errors, and are especially problematic for languages with explicit memory management. For example, C and C++ memory-related errors resulting in leaked memory 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-collected languages preclude the first two errors, but not the last. Since garbage collection conservatively approximates liveness, it cannot detect or reclaim objects referred to by unnecessary references. Thus, a *memory leak* in a garbage-collected language occurs when a program maintains references to objects that it no longer needs, preventing the garbage collector from reclaiming space. In the best case, unnecessary references degrade program performance by increasing memory requirements and consequently collector workload. In the worst case, a leaking, growing data structures cause the program to run out of memory and crash. Even if a growing data structure is all still in use and therefore not a true leak, application reliability and performance can suffer and, thus, is a cause for concern. 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 mem-



(a) javac (b) fop

Figure 5.1: Heap-Occupancy Graphs

ory is not necessarily related to the source of the heap growth.

To demonstrate leaks, we illustrate heap composition using heap-occupancy graphs [28, 96, 168]. A heap-occupancy graph plots the total heap occupancy on the y-axis over time measured in allocation on the x-axis. Figure 5.1 shows the heap occupancy graphs for javac from SPECjvm [166] and fop from the DaCapo suite[29]. A heap-occupancy curve with a consistently positive slope clearly indicates systematic heap growth. Figure 5.1(a) shows four program allocation phases for javac that reach the same general peaks and indicate that javac uses about the same maximum amount of memory in each phase and no phase leaks memory to the next. On the other hand, Figure 5.1(b) shows that the memory requirements for fop grow continuously during execution. Although these graphs reveal potential leaks, they do not indicate the source of the leak.

Previous approaches for finding leaks use heap diagnosis tools that rely on a combination of heap differencing [62, 63, 148] and allocation and/or fine-grain object tracking [38, 44, 52, 92, 147, 159, 160, 173, 174]. These techniques can degrade performance by a factor of two or more, incur substantial mem-

ory overheads, rely on multiple executions, and/or offload work to a separate processor. Additionally, they yield large amounts of low-level details about individual objects. For example, if a `String` leaks, they can report individual `String`s numbering in the 10,000s to 100,000s and allocation sites numbering in the 100s to 1000s. Sifting through these reports and interpreting them requires a lot of time and expertise. Thus, prior work lacks precision and efficiency.

This chapter introduces *Cork*, an accurate, scalable, online, and low-overhead memory leak detection tool for typed garbage-collected languages that uses *CPFG*s to accurately identify leaking data structures. We first present related work in Section 5.1, and then present an example memory leak that we use as a running example through this chapter in Section 5.2. We describe how Cork uses a *CPFG*—the simplest type of *HSG*—to summarize, identify, and report data structures with systematic heap growth in Section 5.3. We show that it provides both efficiency and precision in Section 5.4. Finally, we conclude in Section 5.5.

5.1 Related Work

The problem of detecting memory leaks is well studied. Compile-time analysis can find double free and missing frees [94] and is complimentary to our work. Offline diagnostic tools accurately detect leaks using a combination of heap differencing [62, 63, 148] and fined-grained allocation/usage tracking [44, 92, 159, 160, 173, 174]. These approaches are expensive and often require multiple executions and/or separate analysis to generate complex reports full of low-level details about individual objects. In contrast, Cork's completely online analysis reports summaries of objects by class while con-

cisely identifying the dynamic data structure containing the growth. Online diagnostic tools rely on detecting when objects exceed their expected lifetimes [147] and/or detecting when an object becomes *stale* [38, 52]. While staleness-detection differentiates in-use objects from those not in-use, it is not always effective. For example, as a `HashMap` grows, it periodically *rehashes* thereby touching all the elements in the map. Finding memory leaks due staleness does not work in this case. We instead detect growing data structures, a complimentary technique.

Static approaches, for example Heine and Lam [94], rely on compile-time analysis to detect memory leaks. Here a pointer analysis identifies potential memory leaks in C and C++ using the object ownership abstraction. They find double frees and missing frees that occur when the program overwrites the most recent pointer to an object or data structure without first freeing it. It does not find growing data structures and thus static approaches are complementary to our work. The challenge in implementing our approach for C and C++ is connecting the allocation type to memory, since *malloc* is untyped. Their static analysis of ownership types could provide similar information to explicit types used in Java.

The closest related work is Leakbot which combines offline analysis with online diagnosis to find data structures with memory leaks [83, 135, 136]. Leakbot uses JVMPI to take heap snapshots that it offloads 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 most expensive part of the analysis to another processor, Leakbot minimizes the impact on the application while maintaining detailed per-object information. It then relies on an additional processor to

perform heap differencing across multiple copies of the heap—a memory overhead potentially 200% or more that is proportional to the heap—and ranking which parts of the *object* graph may be leaking. Leakbot produces very detailed object-level statistics which depend on the precision of the heap snapshots. Cork, on the other hand, summarizes object instances in a *CPFG* graph, thereby preserving only object class information while minimizing the memory overhead (less than 1%) and allowing it to run continuously and concurrently with the application.

Several completely online instance-based approaches for finding memory leaks exist for C/C++ and Java. Qin et al. detect memory leaks in C/C++ by looking for objects that exceed their expected lifetimes [147]. Using special hardware to detect and eliminate false positives gives them low time overhead and greater accuracy, but space overhead grows proportionally to the number of objects allocated.

Relying only on software, another online technique for detecting memory leaks identifies *stale* objects as those that have not been accessed in a long time. Chilimbi and Hauswirth introduced a technique for C/C++ where they added per-object bookkeeping information to track stale objects [52]. Per-object bookkeeping information does not translate well to Java where even the smallest application creates millions of distinct objects, making per-object tracking too expensive in both space and time (as shown in Chapter 3). Bond and McKinley address this expense by significantly reducing the space overhead of identifying stale objects as likely leaks by introducing a statistical approach for storing per-object information in a single bit [38]. Using this technique, combined with an offline processing step, they detect memory leaks by reporting allocation and last-use sites of stale objects. Although they achieve

space efficiency, tracking per-object information adds overheads of 45% on average which they reduce to 14% with sampling losing accuracy. For finding memory leaks, differentiating in-use objects from those not-in-use adds additional information and is complimentary to finding heap growth. We show that heap growth itself is a problem: It indicates (1) bad programming practices when objects are in use and (2) leaks when objects are stale but never referenced again. Cork can identify and report the growing data structure class slice and corresponding allocation sites. Furthermore, this information is sufficient to fix the leaks.

5.2 An Example Memory Leak

Figure 5.2 shows a simple order processing system that includes a memory leak which we use as a running example throughout this chapter. *NewOrder* inserts new `Order` into the `allOrdersHT` hashmap and into the `newOrderQ`, as shown in Figure 5.2(a). In Figure 5.2(b), *ProcessOrders* processes the `newOrderQ` one order at a time. It removes each order from the `newOrderQ` and fills it. If the customer is a `Company` (subtype of `Order`), it then issues a bill, putting it on the `billingQ`, and ships the order to the customer. In Figure 5.2(c), when the customer sends a payment, *ProccessBill* removes the order from the `billingQ` and the `allOrdersHT` hashmap. However, if the customer is a `Person` (subtype of `Order`), *ProcessOrders* calls *ProcessPayment* with the customer-provided payment information and ships the order. *ProcessOrders* should, but does not, remove the order from `allOrdersHT` which results in a memory leak. Figure 5.2(d) lists abbreviations and statistics for these classes. This leak is similar to the one Cork finds in jbb2000.

```
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 5.2: Order Processing System

5.3 Finding Leaks with Cork

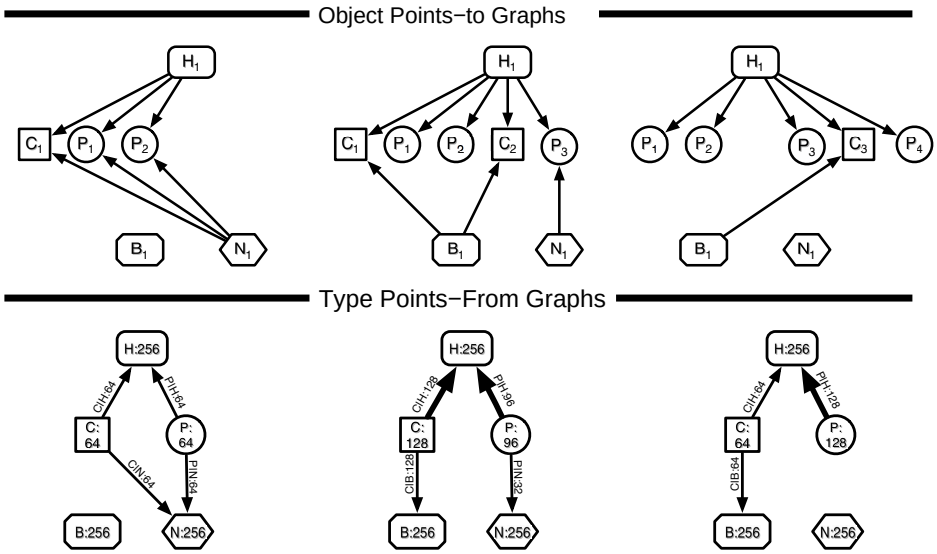
This section overviews how Cork identifies candidate leaks by examining the objects in the heap, finding growth, and reporting to the user the corresponding class for growing objects along with their allocation site and the data structure which contains them. For clarity of exposition, we describe Cork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

5.3.1 Heap Summary Graphs for Leak Detection

To detect systematic heap growth, we use the simplest type of *HSG*, a *class points-from graph* (*CPFG*). For each live, reachable object o, Cork determines the object's class c_o and increments the corresponding class node by the object's size. For each reference from object o to object o' it increments the reference edge from c' to c by the size of o'. At the end of the collection, the *CPFG* completely summarizes the volumes of all classes V_c and references ($V_{c'|c}$) that are live at the time of the collection. Notice that the reference edges in the *CPFG* point in the opposite direction of the references in the heap. Cork uses volume rather than simple count to detect heap growth in order to capture not only when the number of instances of a class increase, but also when the number stays constant but the size of the instances grow (as can be the case with arrays). Additionally, volume gives a heavier weight to larger classes which tend to make the heap grow faster than smaller classes.

5.3.2 Finding Heap Growth

At the end of each collection, Cork differences the *CPFG* for the current collection with data from a set of previous collections. We define those class nodes whose volumes have increased across several collections as *candidates*.



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

Figure 5.3: Comparing Class Points-From Graphs to Find Heap Growth

For each candidate, Cork follows growing reference edges through the *CPFG* to pinpoint the growth.

For example, Figure 5.3 shows the *CPFG*s created during three collections of our example program. Figure 5.3(a) represents an initial state of the system after three orders arrive, but have not yet been processed. Figure 5.3(b) shows four orders processed: two billed and two completed. Notice that the program removes orders from individuals (P) from all the processing queues (B,N), but not from the hashmap (H) resulting in the memory leak shown in Figure 5.2(b). Comparing the *CPFG* 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 5.3(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 class volume *jitters*—fluctuates with high frequency. We say that a class whose volume monotonically increases shows *absolute growth* and one whose volume fluctuates but still increases shows *potential growth*. Cork must not only detect absolute growth but also potential growth.

Cork differences the *CPFG* from the current collection with data from previous collections looking for growing classes and ranks them according to how likely a particular class is a candidate. Additionally it ranks edges in a similar fashion. We examine two different methods for ranking candidates: slope ranking and ratio ranking. Although slope ranking is more principled, ratio ranking yields better results on our benchmarks.

5.3.2.1 Slope Ranking

Recall the beginning of this Chapter, a positive slope in a heap-occupancy graph clearly indicates systematic heap growth. The Slope Ranking Technique (SRT) uses the insight that a growing class must contribute to the overall positive slope in the heap-occupancy graph. The more it contributes, the higher the likelihood that it grows without bound. Thus, SRT ranks candidates according to the portion of the overall heap growth that each class contributes. In this configuration, Cork stores multiple *CPFG*s and calculates the rate of change, or *slope*, between the current collection and previous collections. Slope for class c at collection i is calculated as $s_{c_i} = \delta v_c / \delta A$, where v is the volume of class c live in the heap and A is the total volume allocated. A class node is classified a candidate if it is growing more often than it is shrinking. SRT accumulates the percentage of the overall growth caused by the candidate leaking class c to calculate rank r_{c_i} for collection i such that $r_{c_i} = r_{c_{i-1}} + p_{c_i} * s_{c_i}/S$,

where p is the number of phases (or collections) that c has been growing and S is the rate of change of the total heap ($S_i = \delta V_i / \delta A$). SRT reports classes with positives ranks ($r_i > 0$) as candidates.

SRT suffers significantly from jitter. As a slope becomes negative, it subtracts from the rank. In order to adjust for this, the size of the window used to perform slope calculations can be increased, but this also increases the number of *CPFG*s that must be stored increasing overhead in both time and space. Additionally, program start-up during which the heap naturally grows affects the analysis as the window size increases making SRT more susceptible to false positives.

5.3.2.2 Ratio Ranking

The Ratio Ranking Technique (RRT) ranks class nodes according to the ratio of volumes Q between two consecutive *CPFG*s, and reports classes with ranks above a rank threshold ($r_c > R^c_{thres}$) as candidate leaks. Additionally, RRT uses a *decay factor* f, where $0 < f < 1$ to adjust for jitter and detect potential growth. RRT considers only those class nodes whose volumes satisfy $V_{C_i} > (1 - f) * V_{C_{i-1}}$ on consecutive collections as potential candidates. The decay factor keeps class nodes that shrink a little in this collection but which show potential growth. We find that the decay factor is increasingly important as the size of the leak decreases.

To rank class nodes, RRT first calculates the *phase growth factor* (g) of each class node as $g_{c_i} = p_{c_i} * (Q - 1)$, where p is the number of phases (or collections) that c 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 class node's rank r_c is calculated by accumulating phase growth factors g over

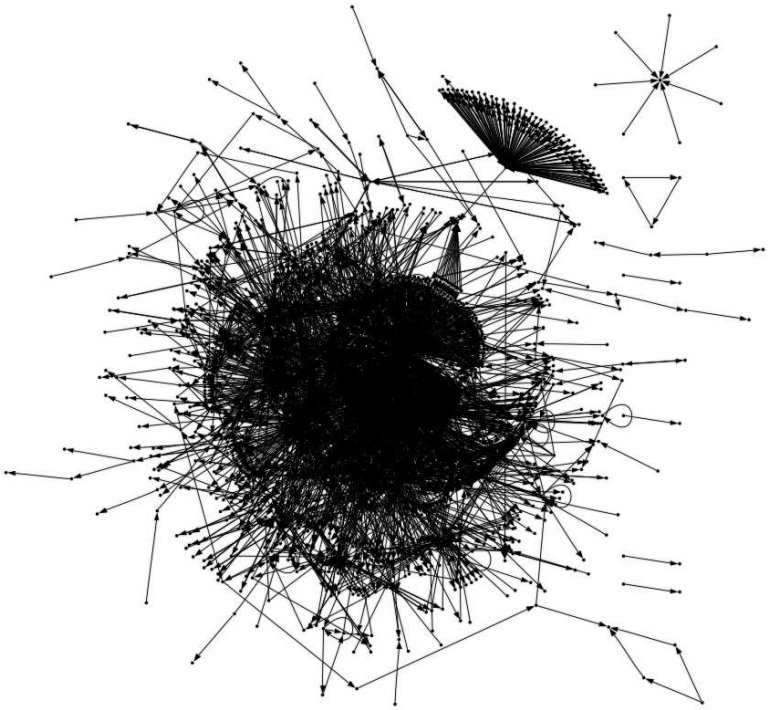
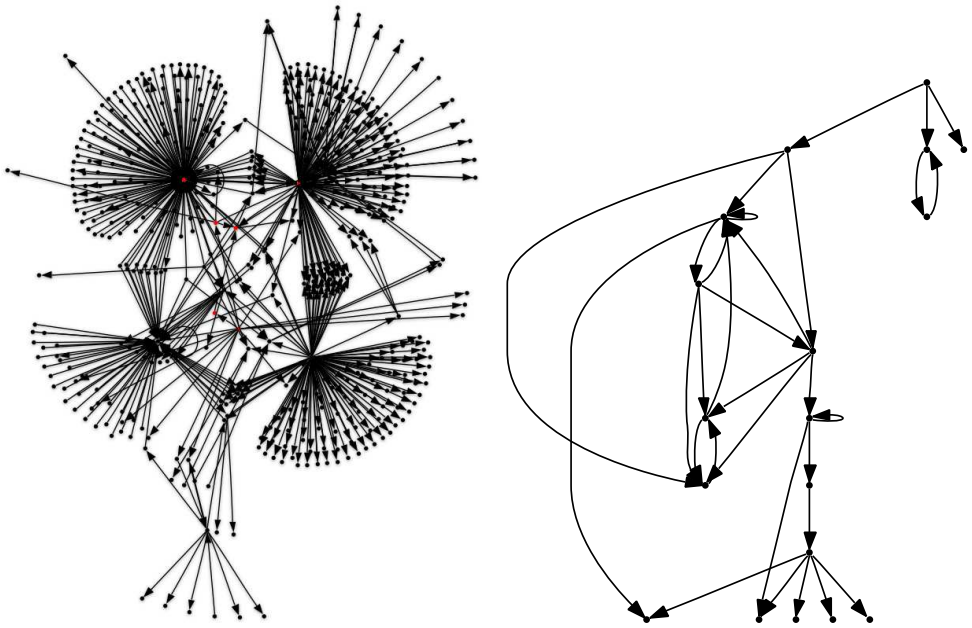


Figure 5.4: The class summary graph

several collections such that absolute growth is rewarded ($r_{c_i} = r_{c_{i-1}} + g_{c_i}$) and decay is penalized ($r_{c_i} = r_{c_{i-1}} - g_{c_i}$). Higher ranks represent a higher likelihood that the corresponding volume of the class grows without bound. RRT reports candidates that show potential growth for at least two (2) phases and never reports a class the first time it appears in the graph.

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



(a) Total connections to candidates (b) Slice Diagram

Figure 5.5: Pruning the summary graph

5.3.3 Correlating to Data Structures and Allocation Sites

Reporting a low-level class such as `String` as a potential candidate is not very useful. To demonstrate the complexity of the *CPFG*, Figure 5.4 shows one instance of the *CPFG* of Eclipse from DaCapo. In the *CPFG*, a node exists for each class in the heap and an edge exists between any two classes if a corresponding reference exists in the heap. Notice how big and complex the *CPFG* is. Focusing on identified candidates and nodes at depth one, we still see a fair amount of complexity (shown in Figure 5.5(a)). Cork further prunes the graph and isolates the growing data structure by constructing a *slice* through the *CPFG*. We define a *slice* through the *CPFG* to be the set of all paths originating from class node c_0 such that the rank of each reference edge $r_{c_k \to c_{k+1}}$ on the path is positive. Thus, a slice defines the growth originating at

a growing class node c_0 following a sequence of class nodes $\{c_0, c_1, \ldots, c_n\}$ and a sequence of growing reference edges (c_k, c_{k+1}) where class node c_k points to c_{k+1} in the *CPFG*. Figure 5.5(b) shows the reported graph for Eclipse and will be discussed in further detail in Section 5.4.2.4. The slice contains candidates and the dynamic data structure containing them.

Additionally, Cork reports class allocation sites. However, unlike some more expensive techniques, it does not find the specific allocation site(s) responsible for the growth. Instead, it reports all allocation sites for each candidate class. If more precision were needed, we could use *DOS* to identify complicit allocation sites.

5.3.4 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 *tracing* collector. In the configurations below, Cork performs the analysis only during full-heap collections. To find leaks in our benchmarks, Cork needed approximately six full heap collections during which heap growth occurs. An incremental collector that never collects the entire heap may add Cork by defining intervals and combining statistics from multiple collections until the collector has considered the entire heap (i.e., an interval). Cork would then compute difference statistics between intervals to detect leaks.

5.3.5 Cork in Other Languages

Cork's heap summarization, the *CPFG*, relies on the garbage collector's ability to determine the class of an object. We exploit the object model of managed languages, such as Java and C#, by piggybacking on their re-

quired global class information to keep space overheads to a minimum. There are, however, other implementation options. For managed languages that lack user-defined class information, such as Standard ML, other mechanisms may be able to provide equivalent information. Previous work provides some suggestions for functional languages that tag objects [149, 150, 152]. For example, type-specific tags could index into a hashmap to store class nodes. Alternatively, objects could be tagged with allocation and context information allowing Cork to summarize the heap in an *allocation-site points-from graph*. These techniques, however, would come at higher space and time overheads.

5.4 Results

This section presents overhead and qualitative results for Cork. We show that ratio ranking has few false positives and higher accuracy that slope ranking, and that furthermore, a variety of reasonable values for the decay factor and the rank threshold give similarly accurate results. Applying Cork to 14 commonly used benchmarks, Cork finds heap growth in three benchmark (fop, jess, and SPECjbb2000) and the data structure reports enabled us to fix them very quickly, even though we were not previously familiar with these applications. Finally we use Cork to debug a reported memory leak in Eclipse bug #115789.

5.4.1 Achieving Accuracy

Cork's accuracy depends on its ability to rank and report growing classes. Table 5.1 shows the number of candidates that are reported using slope ratio (SRT). While SRT accurately identifies growth in fop, jess, jbb2000, and Eclipse bug #115789, it also falsely identifies heap growth in javac and

Benchmark	SRT
Eclipse bug #115789	6
fop	2
pmd	0
ps	0
javac	2
jython	0
jess	2
antlr	0
bloat	3
jbb2000	1
jack	0
mtrt	0
raytrace	0
compress	0
db	0

Table 5.1: Number of classes reported in at least 25% of garbage collection reports using the Slope Ranking Technique.

bloat, programs that do not display systematic heap growth. This result is mainly due to very erratic growth patterns in both programs. Ratio ranking (RRT) offers a more robust heuristics for ranking classes. By increasing the rank when the class grows and decreasing it when it shrinks, RRT more accurately captures growth across many collections without depending upon window size.

For the RRT, we experiment with different sensitivities for both the decay factor f and the rank threshold R_{thres}. Table 5.2 shows how changing the decay factor changes the number of reported classes. We find that the detection of growing classes is not sensitive to changes in the decay factor ranging from 5 to 20%. We choose a moderate decay factor ($f = 15\%$) for which Cork accurately identifies the only growing data structures in our benchmarks without any false positives. Table 5.3 shows how increasing the rank threshold eliminates false positives from our reports. Additionally, we experi-

Benchmark	0%	5%	10%	15%	20%	25%
Eclipse bug #115789	0	6	6	**6**	6	6
fop	2	2	2	**2**	2	2
pmd	0	0	0	**0**	0	0
ps	0	0	0	**0**	0	0
javac	0	0	0	**0**	0	0
jython	0	0	0	**0**	0	1
jess	0	1	1	**1**	1	2
antlr	0	0	0	**0**	0	0
bloat	0	0	0	**0**	0	0
jbb2000	0	4	4	**4**	4	4
jack	0	0	0	**0**	0	0
mtrt	0	0	0	**0**	0	0
raytrace	0	0	0	**0**	0	0
compress	0	0	0	**0**	0	0
db	0	0	0	**0**	0	0

Table 5.2: Number of classes reported in at least 25% of garbage collection reports while varying the *decay factor* from Ratio Ranking Technique ($R^t_{thres} = 100$). We choose a decay factor $f = 15\%$.

Benchmark	0	50	100	200
Eclipse bug #115789	12	6	**6**	6
fop	35	2	**2**	1
pmd	11	2	**0**	0
ps	3	0	**0**	0
javac	71	2	**0**	0
jython	3	0	**0**	0
jess	9	1	**1**	1
antlr	9	0	**0**	0
bloat	33	0	**0**	0
jbb2000	10	6	**4**	4
jack	9	0	**0**	0
mtrt	3	2	**0**	0
raytrace	4	0	**0**	0
compress	4	0	**0**	0
db	2	0	**0**	0

Table 5.3: Number of classes reported in at least 25% of garbage collection reports while varying the *rank threshold* from Ratio Ranking Technique ($f = 15\%$). We choose rank threshold $R^t_{thres} = 100$.

ment with different rank thresholds and find that a moderate rank threshold ($R_{thres} = 100$) is sufficient to eliminate any false positives. We discuss the differences in the number of reported classes between SRT and RRT as we discuss each benchmark in the next section

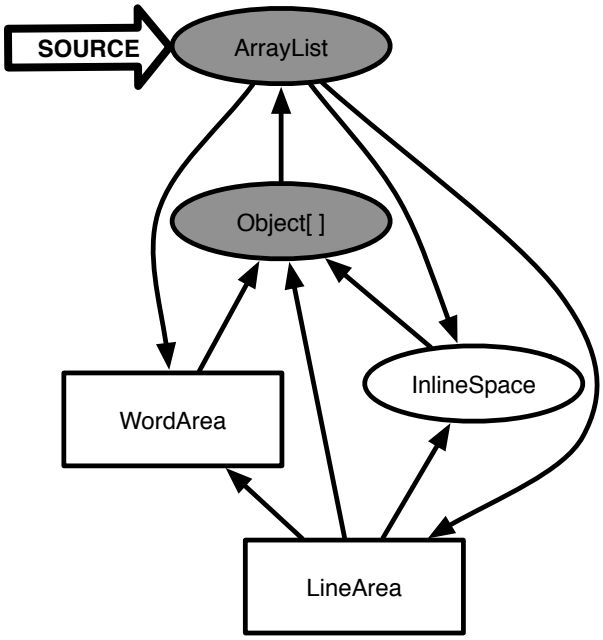
5.4.2 Finding and Fixing Leaks

Cork identifies heap growth in three of our benchmarks: SPECjbb2000, fop, and jess. Additionally, we use Cork to debug a reported memory leak in Eclipse bug #115789. Each section first describes the benchmark or program, demonstrates how Cork found the growing class and data structure, and concludes with an analysis of the growth.

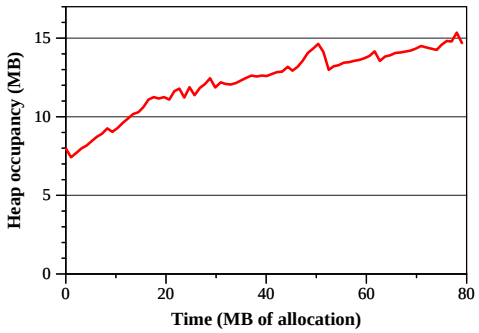
5.4.2.1 fop

The program fop (Formatting Objects Processor) is part of 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 5.6, which clearly demonstrates an overall monotonic heap growth.

Cork analyzes fop and Figure 5.6(a) shows the RRT reports. Both SRT and RRT report `ArrayList` and `Object[]` as candidates for growth. Since `ArrayList` is implemented as `Object[]`, we focus just on `ArrayList` for our analysis. We begin our exploration by examining the slices of the *CPFG* to determine what is keeping the `ArrayList` alive. Figure 5.6(a) shows part of the slice for `ArrayList`. It shows that `ArrayLists` are nested in a data structure which stores the content of the document being converted. Finally, Cork lists the allocation sites for all the class giving the user a starting point



(a) Slice Diagram



(b) Heap Occupancy Graphs for fop

Figure 5.6: Fixing fop

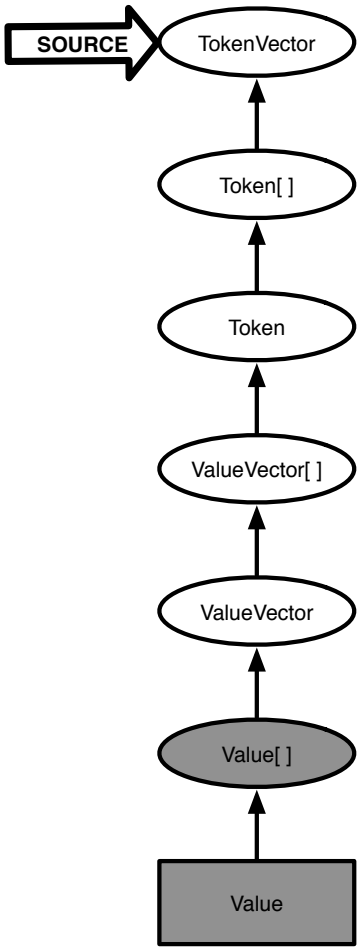
for debugging. Because the allocation sites are numerous, it is not useful to explore `ArrayList`. We go to secondary allocations sites: `WordArea` and `LineArea`.

Next we explore fop's implementation. fop performs two passes over a single complex data structure built with `ArrayList`: the first pass builds the formatting object tree where `ArrayList` contains different formatting object which themselves can contain one or more `ArrayList`. Once fop encounters an end of page sequence, it begins rendering during a second pass over the data structure it built during parsing. Thus, rendering uses the entire data structure. While our analysis accurately pinpoints the source of unbounded heap growth, fop does not have a memory leak because the entire heap is live. The developers of fop agree with this analysis, that the heap growth that fop experiences is partly inherent to the formatting process and partly caused by poor implementation choices [6].

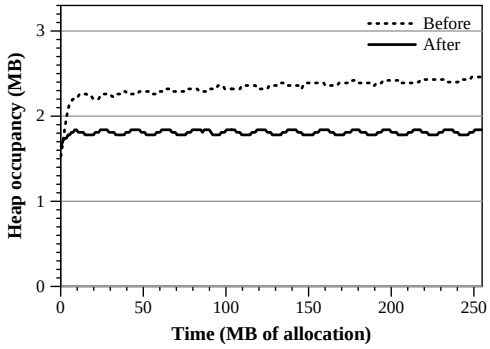
5.4.2.2 jess

From the SPECjvm benchmark suite, jess is a Java Expert Shell System based on NASA's CLIPS [139]. It grows 45KB every 64MB. In an expert system, the input is a set of facts and a set of rules. Each fact represents an existing relationship and each rule a legal way of manipulating facts. The expert system then reasons by using rules to *assert* new facts and *retrace* existing facts. As each part of a rule *matches* existing facts, the rule *fires*, creating new facts and removing the rule from the set of activated rules. The system continues until the set of activated rules becomes empty.

RRT reports `Value` as the overwhelmingly growing class. The slice of the *CPFG* is diagrammed in Figure 5.7(a) where the square node represents



(a) Slice Diagram



(b) Heap Occupancy Graphs for jess

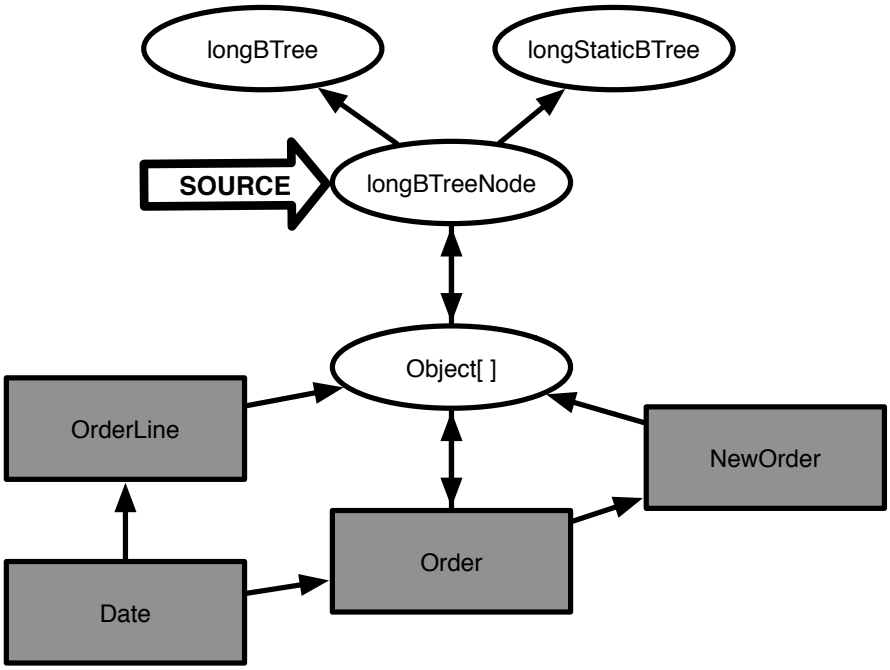
Figure 5.7: Fixing jess

the reported class. Correlating it to the implementation, jess compiles all the rules into a single set of nodes. Fact assertion or retraction is then turned into a *token*, which is fed to the input nodes of the network. Then the nodes may pass the token on to its children or filter it out. As tokens are propagated through the network, rules create new facts. Each new fact is stored in a `Value` in a `ValueVector` implemented as `Value[]`. `ValueVector` is stored in a `ValueVector[]` in a `Token`. A global `TokenVector` implemented as `Token[]`, stores the tokens in the system. SRT reports two of these classes, both of which are in the slice reported by RRT: `Value` and `Value[]`. These facts are part of the input.

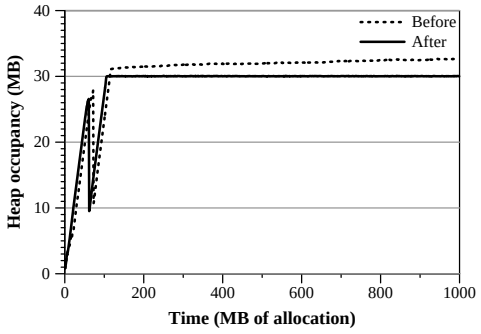
Examining the input for jess, we find that the benchmark iterates over the same problem several times. The developer made it artificially more complex by introducing distinct facts in the input file representing the same information for each iteration. Thus, with each iteration, the number of facts to test increases which triggers more allocation, but some of the facts are redundant. This complexity is documented in the input file. In order to remove the memory leak, we eliminated the artificial complexity from the input file. Figure 5.7(b) shows both the original heap occupancy graph and the resulting heap occupancy graph. The heap growth, and thus the memory leak, is gone.

5.4.2.3 SPECjbb2000

The SPECjbb2000 benchmark models a wholesale company with several warehouses (or districts). Each warehouse has one terminal where customers can generate requests: e.g., place new orders or request the status of an existing order. The warehouse executes operations in sequence, with each operation selected from the list of operations using a probability distribution. It imple-



(a) Slice Diagram



(b) Heap occupancy graph

Figure 5.8: Fixing SPECjbb2000

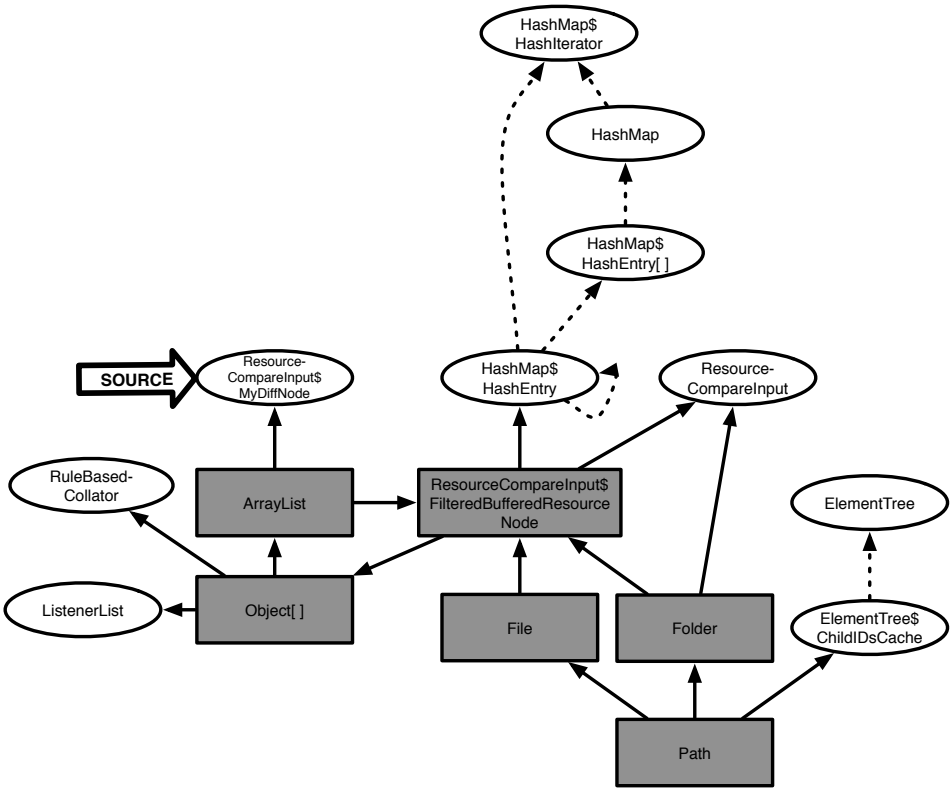
ments this system entirely in software using Java classes for database tables and Java objects for data records (roughly 25MB of data). The objects are stored in memory using `BTree` and other data structures.

RRT analysis reports four candidates: `Order, Date, NewOrder,` and `OrderLine.` The rank of the four corresponding class nodes oscillates between collections making it difficult to determine their relative importance. Examining the slices of the four reported class nodes reveals the reason. There is an interrelationship between all of the candidates and if one is leaking then the rest are as well. The top of Figure 5.8(a) shows the Cork slice report where the shaded nodes are growing. Notice that because of the prolific use of `Object[]` in SPECjbb2000, its class node volume jitters to such a degree that it never shows sufficient growth to be reported as leaking. Since the slice includes all class nodes with $r_t > R^t_{thres}$ and reference edges with $r_e > 0$, the slice sees beyond the `Object[]` to the containing data structures.

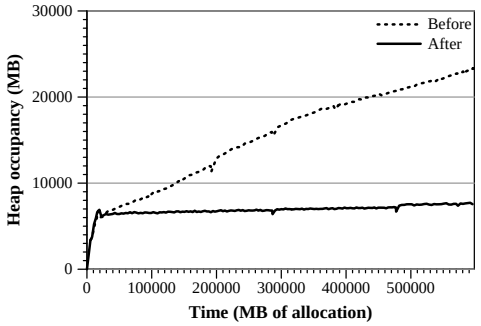
We correlate Cork's results with SPECjbb2000's implementation. We find that orders are placed in an `orderTable`, implemented as a `BTree`, when they are created. When they are completed during a *DeliveryTransaction*, they are not properly removed from the `orderTable`. By adding code to remove the orders from the `orderTable`, we eliminate this memory leak. Figure 5.8(b) shows the heap occupancy, before and after the bug fix, running SPECjbb2000 with one warehouse for one hour. It took us only a day to find and fix this bug in this large program that we had never studied previously.

5.4.2.4 Eclipse bug #115789

Eclipse is a widely-used, open-source integrated development environment (IDE) written in Java [176]. Eclipse is big and complex. Eclipse bug #115789doc-



(a) Slice Diagram



(b) Heap Occupancy Graphs for eclipse

Figure 5.9: Fixing eclipse

uments an unresolved memory leak in the Eclipse bug repository from September 2005. We repeat this bug by manually comparing the contents of two directory structures multiple times to trigger the leak at a high rate. This approach accentuates this bug compared to other potential bugs. Both RRT and SRT reported six candidates: `File, Folder, Path, ArrayList, Object[],` and `ResourceCompareInput$FilteredBufferedResourceNode.` Figure 5.9(a) shows the growth slice for the candidates, the close interrelationship between them, and several possible roots of the heap growth.

Correlating Cork's results with the Eclipse implementation showed that upon completion, the differences between the two directory structures are displayed in the `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`. The `CompareEditorInput` stores the differences of the two directory structures in a linked list of `ResourceCompareInput%MyDiffNode`. Figure 5.9(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.

5.5 Summary

This chapter presented using a *CPFG* to summarize the heap to identify objects contained in data structures and allocation sites that cause systematic heap growth. We implement this approach in Cork, a tool that identifies

growth in the Java heap and reports slices of a summarizing *CPFG*. We show that Cork precisely identifies data structures with unbounded heap growth in three popular benchmarks: fop, jess, and jbb2000. Additionally we analyzed Eclipse. These four programs contained hard-to-diagnose leaks that had eluded developers. Each heap growth represented poor program design or a semantic bug. The *CFSG* effectively summarized heap behavior to reveal the objects, data structure slice, and allocation sites that exhibited anomalous heap growth. Furthermore, this information is precise enough to pinpoint the semantic errors to a person that was previously unfamiliar with the application. We show that 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. These results indicate that heap summarization is a valuable tool for finding the heap-growth anomaly.

Chapter 6

Dynamic Shape Analysis

For a long time, static analysis has helped programmers find bugs and understand their programs by correlating behavior with program locations. Static analysis applied to the heap, *shape analysis*, seeks to characterize the regular structures (arrays and recursive data structures) used by programs to manage the large number of objects in the heap. Unfortunately, static shape analysis remains prohibitively expensive. In this chapter, we analyze heap regularity to assist debugging and program understanding in a tool called *ShapeUp*.

ShapeUp finds and characterizes recursive data structures and their *dynamic invariants* (likely invariants observed at runtime). We show how ShapeUp uses this analysis to assist in finding bugs and understanding data structures. We start by describing related work in Section 6.1 and continue by defining the data structures that we analyze in Section 6.2. Section 6.3 describes the benchmarks, the dynamic shape and invariants discovered by ShapeUp. We examine how ShapeUp detects errors injected in recursive data structures in microbenchmarks in Section 6.4. Finally, we conclude in Section 6.5.

6.1 Related Work

Related work includes static shape analysis, dynamic invariants based on program-counter locations, error detection and correction using invariant specifications, and C heap analysis.

Static shape analysis seeks to understand heap structure by analyzing code to identify recursive data structures [80, 155, 156], Unfortunately, it is not widely used because it is so expensive and necessarily conservative. Our analysis efficiently gives the same information but is specific to one or more program executions since it observes the state of the heap rather than proving all possible heap states. Dynamic shape analysis, like static shape analysis, can be used to generate specifications and tests. The *ownership type declaration* is a proposed language construct that can enforce the property that a particular type may only ever have one pointer to it [40, 56]. Our work identifies candidates for ownership types and can help programmers to use ownership type declarations, if they become available.

More recently, dynamic analyses have discovered likely invariants by mining dynamic program behavior, correlating it with program locations, and then identifying anomalous executions [74, 86, 88, 126, 129, 140, 189]. For example, Hangal and Lam showed that crashes are often preceded by anomalous behavior, i.e., the program violates one or more dynamic invariants that were established either on previous executions or earlier in the current execution. They show that recording variable and condition values, and reporting unseen values aids debugging. We show that this hypothesis applies to the heap as well, i.e., the heap object graph encodes semantics and unusual heap relationships that reveal software flaws.

Recent work on data structure repair has shown how to detect and correct data structure errors with user-defined predicate routines [39, 72]. This approach requires a user specification of the predicate and then uses model checking and partial evaluation to fix errors as they occur in the wild. User-defined predicates can encode valuable additional information, such as which value encodes the number of nodes that should be in the data structure, which ShapeUp will not discover. The advantage of ShapeUp is that it is fully automated and does not need a predicate routine. It detects similar errors by automatically discovering many of the same properties that user predicates contain. Developers can use the results of our approach to help them write their predicates for complex recursive data structures.

HeapMD examines simple heap properties in C programs such as the percentage of objects in the heap with a given degree, i.e., the number of incoming and outgoing pointers [51]. This work shows that these simple metrics are useful for C programs. For example, many C heaps contain a stable fraction of objects with an in- or out-degree of 0, 1, or 2. Our work shows that the more complex relationships in the object graph for Java programs rarely provide stable whole heap invariants. Differentiating the heap by class and connectivity, however, reveals recursive data structures that do have many stable degree invariants. Furthermore, our approach detects violations of these invariants with high accuracy.

Pheng and Verbrugge visualize dynamic data structure evolution from program traces, showing how memory usage and drag varies over time [144]. They analyze complete program traces, whereas we show how to efficiently compute summaries by piggybacking on the garbage collector. Their analysis identifies lists, trees, and directed acyclic graphs, whereas ShapeUp discovers

invariants and measures variance over complex data structures. We show how to use this information to find bugs and characterize the heap of large programs.

6.2 Data Structure Analysis

To manage large amounts of data, programs written in modern languages use recursive data structures. Developers implicitly and explicitly maintain invariants over data structures and in the code that allocates and manipulates them. A *recursive data structure* (*RDS*) contains a set of objects linked by references (pointers) in a regular pattern such that it is composed of smaller or simpler instances of the same data structure. For example, a subset of a singly-linked list is also a singly-linked list. We refine this definition to include object class. The simplest recursive data structures have objects of a single class that reference only other objects of the same user-defined class. For instance, a tree is composed of smaller trees (sub-trees) where the smallest tree is a single node. A given class definition of a tree contains a `Node` with some number of references to other `Node`s. While the definition of the data structure is unbounded, the size of any particular *RDS* in the heap is bounded.

Figure 6.1 shows the composition of the heap in terms of recursive data structures for the SPECjvm and DaCapo benchmarks based on the above definitions. We measured recursive data structures by their implementation. Data structures implemented in the Java class libraries (library data structures) were measured separately from those implemented by the program (*home-brewed data structures*). The results reflect the ubiquitous use of recursive data structures in our benchmarks. The compress and mpegaudio benchmarks rely strictly on arrays for handling their data. In other benchmarks, 91% of all

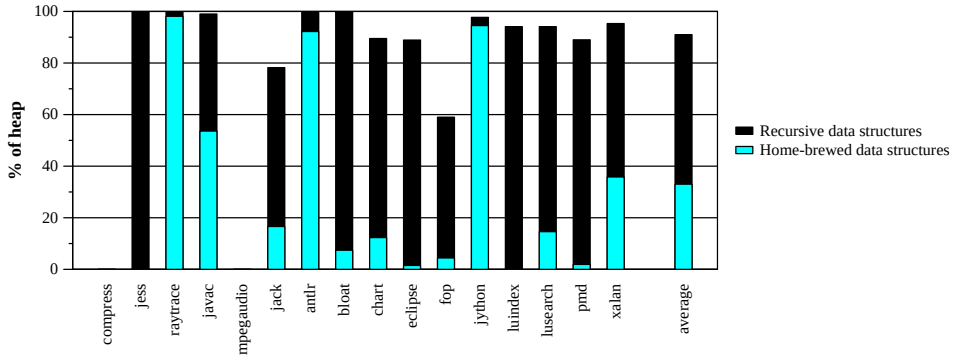


Figure 6.1: Data structure makeup of the heap (objects)

objects are part of a *RDS* and 33% of objects are contained in a home-brewed data structure.

6.2.1 Heap Summaries for *RDS* Analysis

The state of a program's heap can be expressed as a directed graph $G = \{V, E\}$, where V is the set of all heap-allocated objects and E is the set of references between objects in the heap. That is, if an object o has a reference to object o', then the edge (o, o') exists in the graph G. The in-degree of an object o' is the number of other objects o in the heap that reference o'. The out-degree of an object o' is the number of objects o to which o' refers. The *roots* of the heap-graph, those vertices that have an in-degree equal to 0, are referenced by objects outside the heap. For our analysis of *RDS*, we only count in-degree resulting from those edges where objects o and o' belong to the same class c and aggregate the degree statistics by user-defined class in the nodes of a *class field-wise summary graph* (*CFSG*). In field edges, we capture all the points-to relationships by counting all references between any two classes in the field edges. At the end of each collection, the *CFSG* completely summarizes the number of objects of each class including their degree distribution and the

number of field-edges that are live at the time of the collection.

6.2.2 Analysis and Degree invariant Detection

At the end of each collection, we iterate over the *CFSG* incorporating the current *CFSG* into a cumulative *CFSG* which tracks the accumulated statistics for in- and out-degree across all collections. We aggregate the average percent of objects with each degree metric (e.g., in=0, in=1, in=2, and in>2). If two or fewer objects exhibit the degree metric, we call the metric a *quirk*. For example in a singly-linked list, there is exactly one node with in=0, the head (root) of the list, and thus the root is a quirk. Otherwise, we measure the range of the metric as a percentage of objects observed by ShapeUp during one or more executions. We find that quirk metrics are very sensitive to violation when anomalies are introduced while ranges are more tolerant.

6.3 Benchmarks

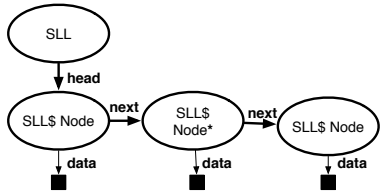
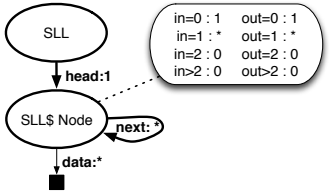
To evaluate ShapeUp, we examine recursive data structures in isolation in microbenchmarks for singly-linked and doubly-linked lists, binary trees, and linked hashmaps. We also evaluate the data structures in the SPECjvm and DaCapo benchmarks and divide them into library implementations and home-brewed data structures.

6.3.1 Microbenchmarks

To study individual recursive data structures, we implement each in a microbenchmark. Tables 6.1-6.3 show implementations, sample heap graphs, and the corresponding *CFSG* for our microbenchmark data structures. These

Table 6.1: Microbenchmark list data structures showing implementation, heap graphs, and corresponding *HSG*.

Implementation	Objects Graph	Heap Summary Graph (*HSG*)
Singly-Linked List		

1 class SinglyLinkedList{
2 Node head;
3
4 static class Node{
5 Object data;
6 Node next;
7 }
8 ...
9 }
		

Doubly-Linked List		

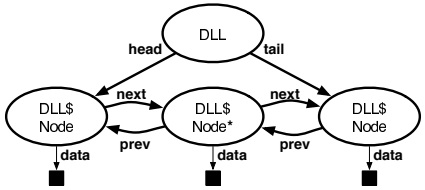
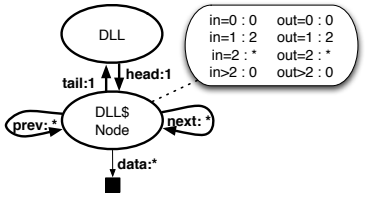
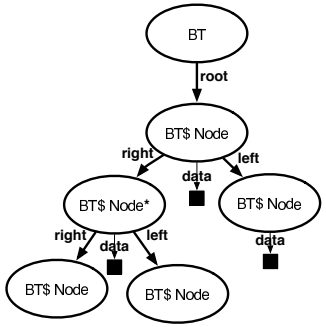
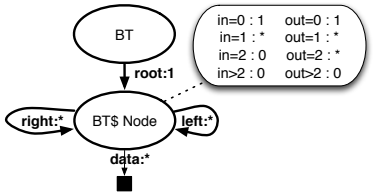
1 class DoublyLinkedList{
2 Node head;
3 Node tail;
4
5 static class Node{
6 Object data;
7 Node next;
8 Node prev;
9 }
10 ...
11 }
		

Table 6.2: Microbenchmark tree data structures showing implementation, sample heap graphs, and corresponding *HSG*.

Implementation	Objects Graph	Heap Summary Graph (*HSG*)
Binary Tree		

1 class BinaryTree{
2 Node root;
3
4 static class Node{
5 Object data;
6 Node left;
7 Node right;
8 }
9 ...
10 }
		

Binary Tree with Parent		

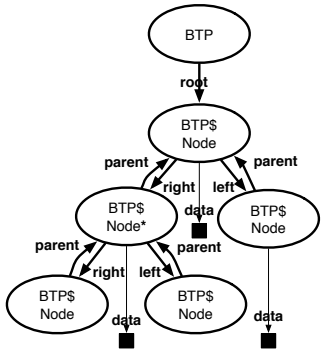
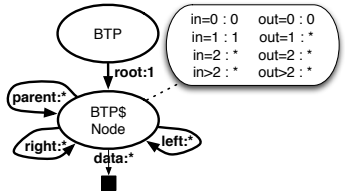
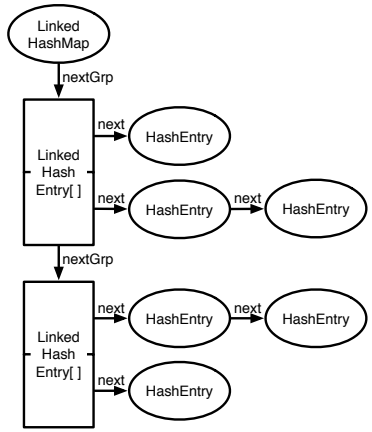
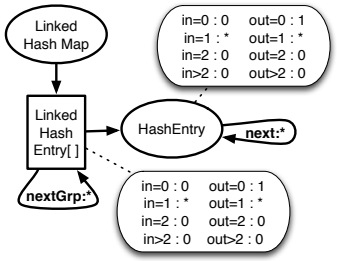
1 class BinaryTreeParent{
2 Node root;
3
4 static class Node{
5 Object data;
6 Node left;
7 Node right;
8 Node parent;
9 }
10 ...
11 }
		

Table 6.3: Microbenchmark linked hashmap data structure showing implementation, sample heap graphs, and corresponding *HSG*.

Implementation	Objects Graph	Heap Summary Graph (*HSG*)
HashMap (simplified)		

```
1  class LinkedHashMap{
2    Entry root;
3
4    class LinkedHashEntry {
5      HashEntry[] entries;
6      LinkedHashEntry nextGrp;
7    }
8
9    class HashEntry{
10     Object key;
11     Object data;
12
13     Entry next;
14   }
15   ...
16 }
```





include a singly-linked and doubly-linked list in Table 6.1, a binary tree and a binary tree with parent pointer in Table 6.2, and the more complex linked hashmap in Table 6.3. Hashmaps account for more than 50% of the recursive data structures in 8 of the 16 benchmarks we tested. For each data structure, the first column shows the definition of a `Node` class that implements the recursive backbone of the data structure. The second column shows an example object graph where nodes marked with the asterisk could be repeated many times. The last column shows the corresponding *CFSG* for each data structure. Notice that a backbone of a recursive data structure is easily distinguished by its *self-loop* in the *CFSG*.

6.3.2 Dynamic Shape and Invariant Characterization

This section presents the in- and out-degree invariants in the microbenchmarks, SPECjvm, and DaCapo benchmark suites. ShapeUp discovered invariants not only when a data structure is used in isolation, but also when it is used in a more complex application. These results show that class can summarize large complex heaps, as well as individual data structures. We examine all recursive data structures equally, regardless of whether they are library-implemented or home-brewed. Section 6.4 shows how we use these invariants to detect errors in these structures.

Table 6.4 lists the variations of the microbenchmark data structures that we tested. We performed 1000 trials on correct executions consisting of a random number between 100 and 100,000 of nodes in the *RDS*. At each garbage collection measurement point, ShapeUp calculates the degree metric and compares it to previous correct runs. If the metric is a quirk (e.g., it is the root of the data structure), ShapeUp indicates the quirk value for future

Table 6.4: ShapeUp finds degree invariants on correct data structures. Data structures that refer back to their parents are marked: w/PP.

Data Structure		$= 0$	$= 1$	$= 2$	> 2
Singly-linked list	in	1	[99.21, 99.99]	0	0
	out	1	[99.21, 99.99]	0	0
Doubly-linked list	in	0	2	[99.06, 99.99]	0
	out	0	2	[99.06, 99.99]	0
Complete binary tree	in	1	[99.63, 99.99]	0	0
	out	[50.00, 50.18]	[0, 0.37]	[49.63, 49.99]	0
Full binary tree	in	1	[99.00, 99.99]	0	0
	out	[50.00, 50.49]	0	[49.50, 49.99]	0
Random binary tree	in	1	[99.61, 99.99]	0	0
	out	[35.97, 38.05]	[23.94, 28.26]	[35.77, 38.00]	0
Complete binary tree w/PP	in	0	[50.00, 50.31]	[0.00, 1.35]	[48.65, 50.00]
	out	0	[50.00, 50.31]	[0.00, 1.35]	[48.65, 50.00]
Full binary tree w/PP	in	0	[50.00, 50.51]	1	[48.48, 50.00]
	out	0	[50.00, 50.50]	1	[48.48, 50.00]
Random binary tree w/PP	in	0	[35.95, 38.55]	[23.11, 28.61]	[35.44, 38.34]
	out	0	[35.95, 38.55]	[23.11, 28.61]	[35.44, 38.34]
LinkedHashMap (HashEntry)	in	0	0	[60.34, 63.47]	[37.44, 38.34]
	out	0	0	0	1

comparison. If the metric is a quirk, ShapeUp determines the percentage of objects with that metric and determines ranges for each metric from the correct runs.

In the table, each microbenchmark has two rows of data for the single recursive data structure existing in the heap: the top representing the in-degree invariants and the bottom representing the out-degree invariants. Each entry presents either a single integer or a pair *[min, max]*. If the metric is a quirk, the entry shows the value of the quirk. For example, $in > 2$ is a quirk for the singly-linked list because no objects ever have in-degree greater than 2 ($in > 2$) in a correct implementation of a singly-linked list. However, the root node of a data structure will represent a quirk ($in = 0$). Some objects have in-degree of zero ($in = 0$). If many objects have the same degree, we present a range of the percentage of objects with that degree (*[min, max]*). In the singly-linked list, out-degree equals one ($out = 1$) for at least 99.21% of all objects and at most 99.99% for all our trials. In this example, the root is a quirk as is the tail. As a result, the percentages vary only as a function of the object graph size.

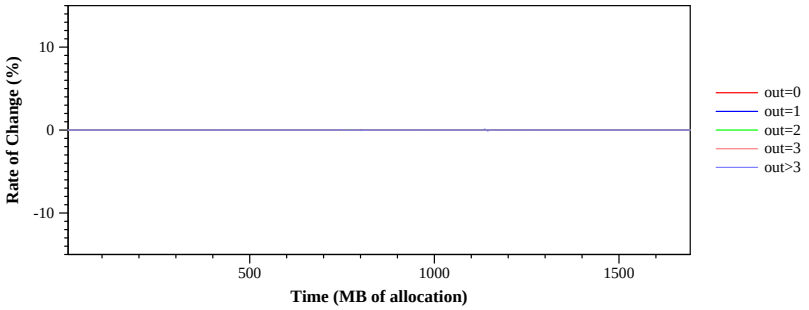
For the less regular data structures, we see fewer quirk metrics in the dynamic invariants, but good range invariants that can help uncover errors. For example, the random binary tree has four quirk invariants ($in = 0$, $in = 2$, $in > 2$, and $out > 2$) and the random binary tree with parent pointers has two quirk invariants ($in = 0$ and $out = 0$). The ranges for both these data structures are useful as well, for example, incoming pointers must still essentially be one. Even $out = 1$ ranging from 23.94 and 28.26 is useful, if there are systematic errors early in testing. The SPECjvm and DaCapo benchmarks show very similar characteristics.

Table 6.5: Dominant recursive data structures for selected SPECjvm and DaCapo benchmarks. L indicates library implementations and H indicates home-grown implementations.

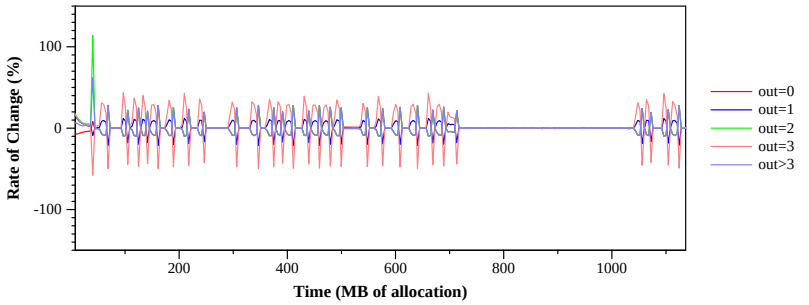
Benchmark	*RDS* Node	L or H?	% heap	metric	= 0	= 1	= 2	> 2
raytrace	OctNode	H	78.0	in	[0]	[0, 0.4]	[0]	[99.3, 99.9]
				out	[0]	[0]	[0]	[100%]
jack	LinkedHashMap$	L	44.2	in	[0]	[0]	[62.7]	[37.3]
	LinkedHashEntry			out	[0]	[0]	[0]	[100%]
	RuntimeNfaState	H	9.4	in	[0, 1.4]	[53.6, 62.3]	[18.8, 31.9]	[14.5, 20.3]
				out	[0]	[0]	[0]	[100%]
antlr	Object[]	H	76.2	in	[0.2, 15.4]	[1.33, 33.9]	[0]	[0]
				out	[12.9, 46.2]	[30.1, 65.5]	[8.3, 25.0]	[0.2, 8.3]
bloat	HashMap$	L	56.6	in	[0]	[99.0, 100]	[0, 0.9]	[0]
	HashEntry			out	[0]	[0]	[75.4, 82.2]	[17.8, 24.6]
	CallMethodExpr	H	3.7	in	[0]	[0]	[0]	[100%]
				out	[0]	[0]	[0]	[100%]
eclipse	LinkedHashmap$	L	59.0	in	[0]	[0]	[61.3, 62.3]	[37.7, 38.7]
	LinkedHashEntry			out	[0]	[0]	[0]	[100%]
fop	HashMap$	L	51.4	in	[0]	[100%]	[0]	[0]
	HashEntry			out	[0]	[0]	[75.1, 79.1]	[20.9, 24.9]
	PropertyList	H	4.4	in	0	[0, 100]	[0, 63.4]	[0, 52.2]
				out	[0]	0]	[0]	[100]
jython	PyFrame	H	94.6	in	[0]	[100%]	[0]	[0]
				out	[0]	[0]	[0]	[100%]
luindex	LinkedHashMap$	L	99.3	in	[0]	[0]	[67.6]	[32.4]
	LinkedHashEntry			out	[0]	[0]	[0]	[100%]
lusearch	WeakHashMap$	L	47.5	in	[0, 83.6]	[16.4, 100]	[0, 0.1]	[0]
	WeakBucket			out	[0]	[0]	[75.2, 83.9]	[16.1, 24.7]
	HitDoc	H	2.0	in	[0]	[0, 100]	[0]	[0, 100]
				out	[37.5, 100]	[0, 50.0]	[0, 15.4]	[0, 37.5]
pmd	HashMap$	L	51.4	in	[0]	[100%]	[0]	[0]
	HashEntry			out	[0]	[0]	[75.9, 78.6]	[21.4, 24.1]
	PackageNode	H	2.0	in	[0]	[100%]	[0]	[0]
				out	[0]	[0]	[100%]	[0]
xalan	ChildIterator	H	34.6	in	[0]	[0]	[0, 52.5]	[47.5, 100]
				out	[0]	[0]	[0]	[100%]

Table 6.5 reports data structures for the SPECjvm and DaCapo Java programs. First it reports the most dominant data structure and indicates whether it is library-implemented (L) or home-brewed (H). If the dominant data structure is library-implemented, the table reports the next most dominant home-brewed data structure. Column two names the recursive class of the data structure and column three indicates how much of the heap is occupied by this data structure. Notice that this single piece of information often indicates what data structure is used. As expected, many major data structures used by these benchmarks are defined in the libraries. For example, hashmap and its variants make up the major portion of data structures used in the benchmarks. Each data structure has two rows of data: the top representing the in-degree invariants and the bottom representing the out-degree invariants. Notice that the home-brewed data structures exhibit more variation in degree ranges, and that many of them have quirk invariants.

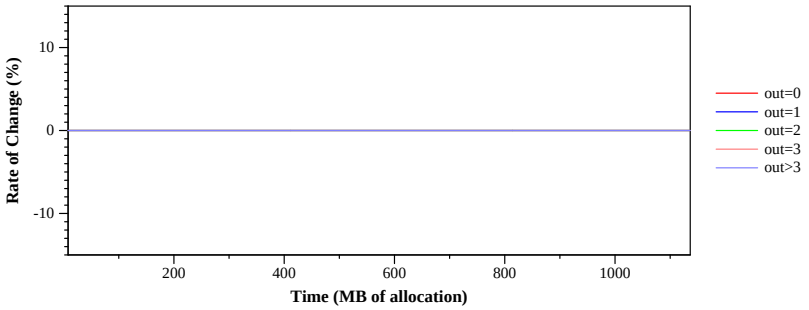
Each data structure has different dynamic invariants and these reports are sufficient for finding many types of errors during development. In particular, consider the case when a developer intends to create a doubly-linked list, but forgets to set the back pointer and creates a singly-linked list instead. The ShapeUp invariant report would clearly indicate that approximately 99% of nodes showed an in-degree of one rather than the expected value of two. ShapeUp's extended reports will indicate this error and show that the data structure does not have the intended shape without requiring a correct execution. In Section 6.4, we show how ShapeUp uses its automatically generated assertions from correct executions to detect errors when errors are inserted into our microbenchmarks, but first we examine whole heap characteristics for SPECjvm and DaCapo.



(a) Out-degree metrics for jython



(b) Out-degree metrics for pmd



(c) Out-degree metrics for `HashMap$HashEntry` in pmd

Figure 6.2: Degree Rate of Change.

6.3.3 Whole heap analysis for SPECjvm and DaCapo Benchmarks

In these experiments, we test whether whole heap invariants hold for SPECjvm and DaCapo. Previous work showed that whole-heap degree metrics are sufficient for discovering data structure bugs in C programs [51]. We measure the degree metrics of the whole heap for the benchmarks and present

sample results in Figure 6.2. Whole-heap degree metrics are not coarse-grained enough to predict errors in data structure for Java. We ran 18 benchmarks from SPECjvm and DaCapo and found that while two programs had many stable metrics – compress (7) and jython (5) (see Figure 6.2(a)). Allocation rate determines the number of objects in the heap, but not the mutations of the data structures. Notice that in this case that the out-degree metric shows a great deal of stability as a function of heap allocation. Four programs tended to show much fewer stable degree metrics – mpegaudio (2), luindex (2), jack (1), and antlr (1). The most common case was the total lack of stability across the entire heap for the remaining 11 programs: jess, raytrace, db, javac, mtrt, jbb2000, bloat, eclipse, fop, lusearch, and pmd. But even when degree-metrics were unstable across the whole heap, most data structures maintained stability. Figure 6.2(b) shows pmd and is more representative of the other programs. The figure plots the out-degree metrics for pmd, which vary wildly as a function of allocation time shown here as a whole factor greater scale than the previous graph. Simply measuring the degree of all the objects in the heap is not sufficient to understand program behavior. However, as we showed above. the degree metrics of all objects in a user-defined class are stable across the entire program execution. Figure 6.2(c) illustrates this stability. It plots the out-degree metrics for `HashMap$HashEntry` in pmd, its biggest data structure.

6.4 Automated Error Detection

This section describes how ShapeUp can automate error detection in many cases. This application of ShapeUp is appropriate for hard-to-find corner cases and can reveal transient hardware errors that corrupt data structure integrity. In learning mode, we train ShapeUp on correct program executions.

ShapeUp determines the likely dynamic invariants as shown in the previous section. In testing mode, we compare the current execution to our discovered invariants. On each full-heap garbage collection (measurement), ShapeUp analyzes the data structures and compares this measurement to the stored invariants from correct executions. It tests each *metric*, i.e., in- and out-degree dynamic invariants. ShapeUp classifies each metric a quirk or as a probabilistic range, as shown in Table 6.4. Quirk metrics are shown as a single number and range metrics are shown as a *[min, max]* pair.

```
1 if (metric is constant) {
2   if (thisMeasurement != constantMeasurement) {
3     // violation occured
4   }
5 } else {
6   thisPercent = thisMeasurement/numberOfObjects
7   if (thisPercent < minPercent || maxPercent < thisPercent) {
8     // violation occured
9 }}
```

```
1 if (metric is constant) {
2   if (thisMeasurement != constantMeasurement) {
3     // violation occured
4   }
5 } else {
6   thisPercent = thisMeasurement/numberOfObjects
7   if (thisPercent < minPercent || maxPercent < thisPercent) {
8     // violation occured
9 }}
```

Figure 6.3: Violation Pseudocode

If the metric is a quirk and the current measurement is not the same, ShapeUp reports an error. Otherwise, ShapeUp reports an error if the fraction of objects with the metric falls below the minimum or above the maximum. Figure 6.3 shows the decision-making code.

Table 6.6 enumerates the errors we inserted into each of the microbenchmark data structures. For example in *random*, we randomly choose an interior object in the singly-linked list and set the tail object to refer to it, forming a

Table 6.6: Errors Introduced

Error	Description	Violation Type	Runs
Singly-Linked List			
cyclic	attaches the `tail` to `head`	single	10
random	attaches the `tail` to a random object	single	100
Doubly-Linked List			
cyclic	creates cycle from `tail` to `head`	single	10
cycle	creates a random cycle between two objects	multiple	100
disconnect	disconnects random link	multiple	100
skip	creates a skip in the `next` or `prev` pointers	multiple	100
error	randomly insert errors (cycle, disconnect, or skip)	multiple	100
binary tree			
linkerror	creates a connection from a `null` pointer to a random object	multiple	100
binary tree with Parent Pointer			
disconnect	delete a random reference	multiple	100
linkerror	creates a connection from a `null` pointer to a random object	multiple	100
errors	randomly insert errors (disconnect or linkerror)	multiple	100
Linked Hashmap			
bucketlink	randomly connects two buckets in the hashmap	multiple	100

cycle. In the binary tree, we delete random references and mutate references to null to refer to random objects. Our approach won't detect errors if the error makes objects unreachable. For example, if the second half of a singly-linked list is erroneously disconnected, the remaining data structure is still a legal list. These errors are relatively easy programming task to catch by adding code that keeps track of the expected number of objects in the data structure. We, therefore, insert errors in which the objects remain reachable.

We perform tests that insert 1, 2, 3, 4, 5, 10, 50 and 100 errors into *RDS* that have 100,000 nodes. Table 6.7 shows the number and type of errors inserted for each microbenchmark. We report the percentage of errors detected and whether the metric violated was a quirk and/or a range metric. Only one entry is not 100%, in this case a doubly-linked list with 100,000 nodes had 100 errors inserted into it and several errors had the affect of cancelling each other out. We believe this result is due to our methodology of using completely random error insertion and believe that real errors are less numerous and more systematic. In all other cases, one error was enough to make one or more of the metrics fall out of range and thus ShapeUp was able to detect the error(s).

6.5 Summary

ShapeUp finds and characterizes recursive data structures and their dynamic invariants to aid program understanding and bug detection. By summarizing the degree characteristics of data structures in a *CFSG*, ShapeUp identifies both library-implemented and home-brewed data structures while adding only an average of 4-8% to total runtime. We show that ShapeUp can be used to detect errors injected into recursive data structures by training ShapeUp on correct runs to discover in- and out-degree invariants based on

Table 6.7: Percentage of runs with detected errors classified as constant and/or range violations.

	Number of errors injected								Metric violated	
Error	1	2	3	4	5	10	50	100	Constant	Range
Singly-Linked List										
cyclic	100	n/a							yes	yes
random	100	n/a							yes	yes
Doubly-Linked List										
cyclic	100	n/a							yes	yes
cycle	100	100	100	100	100	100	100	100	yes	yes
disconnect	100	100	100	100	100	100	100	99	yes	yes
skip	100	100	100	100	100	100	100	100	yes	yes
error	100	100	100	100	100	100	100	100	yes	yes
Complete binary tree										
linkerror	100	100	100	100	100	100	100	100	yes	yes
Full binary tree										
hline linkerror	100	100	100	100	100	100	100	100	yes	yes
Random binary tree										
linkerror	100	100	100	100	100	100	100	100	yes	yes
Complete binary tree with Parent Pointer										
disconnect	100	100	100	100	100	100	100	100	yes	yes
linkerror	100	100	100	100	100	100	100	100	yes	yes
error	100	100	100	100	100	100	100	100	yes	yes
Full binary tree with Parent Pointer										
disconnect	100	100	100	100	100	100	100	100	yes	yes
linkerror	100	100	100	100	100	100	100	100	yes	yes
error	100	100	100	100	100	100	100	100	yes	yes
Random binary tree with Parent Pointer										
disconnect	100	100	100	100	100	100	100	100	yes	yes
linkerror	100	100	100	100	100	100	100	100	yes	yes
error	100	100	100	100	100	100	100	100	yes	yes
Linked Hashmap										
bucketlink	100	100	100	100	100	100	100	100	yes	yes

user-defined class. It then uses these invariants to find errors in runs where errors were randomly injected into the same data structures. We found that ShapeUp successfully finds all errors that we automatically injected and did

so even when a single error was injected. In summary, dynamic heap analysis can effectively use class to summarize the objects in the heap and find dynamic invariants, mining much of the heap's regular structure from the object graph. Furthermore, this information is useful for program understanding and debugging, and there is likely more information to be mined.

Chapter 7

Conclusion

Large programs present new challenges for understanding and debugging. With the ubiquitous use of well-defined user interfaces, libraries and frameworks, it is not surprising that few developers understand every aspect of the program they help develop. Furthermore, complexity makes it difficult to fully validate or exhaustively test software prior to deployment making it probable that software will ship with bugs.

With more objects going into the heap, the heap encodes more program state than ever before making it more likely than ever that bugs will manifest there. Discovering heap characteristics after deployment requires dynamic heap analysis. This dissertation shows how to perform dynamic heap analysis by leveraging the managed runtime. It argues that dynamic heap analysis is a necessary part of program analysis and introduces two techniques (dynamic object sampling and heap summarization) for mining program state efficiently and effectively. We show how to use these two techniques to mine object characteristics and aggregate characteristic statistics in new ways.

First, we attack the problem of estimating allocation-site lifetime. We show how Dynamic Object Sampling (*DOS*) can be used to tag a random sample of objects with allocation site information and then how we accurately estimate the lifetime of allocation during garbage collection while adding only 3% on average to total time. We explore using dynamic lifetimes to opti-

mize programs using dynamic objects sampling with mixed results. While we are the first to improve any SPECjvm benchmark, we show that other benchmarks lack opportunity and suffer only from the added overhead. Despite these degradations, we show that the allocation-site lifetimes calculated are 94% accurate and for the first time show that the reason pretenuring fails for SPECjvm is a lack of opportunity. These results show that dynamic object sampling is useful for performance optimization.

We implemented Cork which summarizes the size of the heap in a compact way to identify data structures that exhibit systematic heap growth. Using a *CPFG*, cork summarizes the volumes of user-defined classes in the heap and the relations between those classes. We demonstrate that ranking the nodes in the *CPFG* positively identifies which classes are increasing in volume and ranking the edges positively identifies the data structure. These results show that heap summarization can identify bugs.

We implemented ShapeUp which performs a more complex dynamic heap analysis discovering degree invariants for recursive data structures. It measures the in- and out-degree of data structure nodes that make up the recursive backbone of a recursive data structure. We show how dynamic invariants for singly- and doubly-linked lists, binary trees, and hashmaps are easily discovered and exploited to identify when an anomaly is inserted into the recursive structure of these data structures. Our results show that dynamic object sampling and heap summarization work synergistically to improve program understanding and detect errors.

In summary, this dissertation is the first to show that dynamic heap analysis can be performed by leveraging the managed runtime to efficiently and effectively mine program state from the heap.

Bibliography

[1] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *ACM Conference on Programming Language Design and Implementation*, pages 269–279, Montreal, Canada, 1998.

[2] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. *ACM SIGPLAN Notices*, 36(1):121–126, January 2001.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[4] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, Colorado, November 1999.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Department of Computer Science (DIKU), University of Copenhagen, May 1994.

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

[7] A. W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–183, 1989.

[8] M. Arnold. Online instrumentation and feedback-directed optimization of Java. Technical Report DCS-TR-469, Department of Computer Science, Rutgers University, Piscataway, New Jersey, 2002.

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

[10] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *ACM Workshop on Dynamic and Adaptive Compilation and Optimization*, Boston, Massachusetts, 2000.

[11] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Cumberland Falls, Kentucky, 2001.

[12] G. Attardi, T. Flagella, and P. Iglio. A customizable memory management framework for C++. *Software—Practice and Experience*, 28(11):1143–1183, 1998.

[13] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *ACM Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

[14] H. Azatchi and E. Petrank. Integrating generations with advanced reference counting garbage collectors. In *International Conference on Compiler Construction*, Warsaw, Poland, April 2003.

[15] D. Bacon, S. Fink, and D. Grove. Space- and time-efficient implementations of the Java object model. In *European Conference on Object-Oriented Programming*, pages 111–132, June 2002.

[16] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *ACM Conference on Programming Language Design and Implementation*, pages 92–103, Snowbird, Utah, June 2001.

[17] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM Symposium on the Principles of Programming Languages*, pages 285–294, New Orleans, Louisiana, January 2003.

[18] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235, 2001.

[19] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[20] D. A. Barrett and B. Zorn. Garbage collection using a dynamic threatening boundary. In *ACM Conference on Programming Language Design and Implementation*, pages 301–314, La Jolla, California, June 1995.

[21] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *ACM SIGPLAN Notices*, 28(6):187–196, June 1993.

[22] J. M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[23] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, Cambridge, Massachusets, November 2000.

[24] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *ACM Conference on Programming Language Design and Implementation*, pages 114–124, Salt Lake City, Utah, June 2001.

[25] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–12, Seattle, Washington, 2002.

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

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

[28] 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.

[29] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, October 2006.

[30] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *ACM Conference on Programming Language Design and Implementation*, pages 153–164, Berlin, Germany, June 2002.

[31] S. M. Blackburn and K. S. McKinley. In or out? Putting write barriers in their place. In *International Symposium on Memory Management*, pages 175–183, Berlin, Germany, June 2002.

[32] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, California, October 2003.

[33] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Kahn, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Weidermann. Wake up and smell the coffee: Evaluation methdology for the 21st century. *Communications of the ACM*, 51(8):83–89, August 2008.

[34] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. *ACM SIGPLAN Notices*, 36(11):342–352, November 2001.

[35] B. Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003.

[36] H. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[37] H.-J. Boehm. Space efficient conservative garbage collection. In *ACM Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, New Mexico, June 1993.

[38] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for*

Programming Languages and Operation Systems, San Jose, California, October 2006.

[39] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis*, pages 123–133, Rome, Italy, July 2002.

[40] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symposium on the Principles of Programming Languages*, New Orleans, Louisiana, January 2003.

[41] G. Bracha and W. Cook. Mixin-based inheritance. In *European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990.

[42] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 353–366, Tampa, Florida, 2001.

[43] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[44] 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.

[45] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, New York, June 1990.

[46] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[47] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. In *ACM Conference on Programming Language Design and Implementation*, pages 125–136, Snowbird, Utah, June 2001.

[48] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. *ACM SIGPLAN Notices*, 33(5):162–173, May 1998.

[49] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *International Symposium on Memory Management*, pages 85–96, Vancouver, British Columbia, Canada, 2004.

[50] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, Georgia, May 1999.

[51] T. M. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, 2006.

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

[53] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium*

on Memory Management, pages 37–48, Vancouver, British Columbia, Canada, October 1998.

[54] W. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for object-oriented language. In *ACM Conference on Programming Language Design and Implementation*, pages 243–354, Washington, DC, June 2004.

[55] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, 2003.

[56] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 48–64, Vancouver, British Columbia, Canada, October 1998.

[57] C. Click. Stack allocation, Jan. 2005. Personal Communication.

[58] W. D. Clinger and L. T. Hansen. Generational garbage collection and the radioactive decay model. In *ACM Conference on Programming Language Design and Implementation*, pages 97–108, Las Vegas, Nevada, June 1997.

[59] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[61] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment formand the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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

[63] 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.

[64] J. Dean, G. DeFouw, D. Grove, V. Litinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose, California, October 1996.

[65] D. L. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice and Experience*, 24(6):527–542, June 1994.

[66] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[67] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference on Object-Oriented Programming*, pages 92–115, Lisbon, Portugal, June 1999.

[68] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, September 1978.

[69] A. Diwan, J. E. B. Moss, and R. L. Hudson. Compiler support for garbage collection in a statically typed language. In *ACM Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992.

[70] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. *ACM SIGPLAN Notices*, 38(2 supplement):76–87, 2003.

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

[72] B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley. STARC: Static analysis for efficient repair of complex data. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Montreal, Canada, October 2007.

[73] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 1994.

[74] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on*

Software Engineering, pages 449–458, Limerick, Ireland, June 2000.

[75] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Memory System Performance*, pages 1–12, Chicago, Illinois, June 2005.

[76] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *International Symposium on Memory Management*, pages 111–120, Minneapolis, Minnesota, October 2000.

[77] D. Gay and A. Aiken. Language support for regions. In *ACM Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, Utah, 2001.

[78] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*, pages 82–93, Berlin, Germany, 2000.

[79] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *ACM Symposium on the Principles of Programming Languages*, pages 273–284, New Orleans, Louisiana, January 2003.

[80] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[81] R. Grehan and P. McLachlan. Memory planning in Java. Technical report, Compuware DevPartner Corporation, 2000.

[82] G. Gudjónsson and W. H. Winsborough. Compile-time memory reuse in logic programming languages through update in place. *ACM Transactions on Programming Languages and Systems*, 21(3):430–501, 1999.

[83] 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.

[84] S. Z. Guyer and K. S. McKinley. Finding your cronies: Static analysis for dynamic object colocation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 237–250, Vancouver, British Columbia, Canada, 2004.

[85] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: A static analysis for automatic individual object reclamation. In *ACM Conference on Programming Language Design and Implementation*, pages 364–375, Ottawa, Canada, June 2006. Submitted to PLDI '06.

[86] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black box components. In *ACM Conference on Programming Language Design and Implementation*, pages 101–111, San Diego, California, June 2007.

[87] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation*, pages 141–152, Berlin, Germany, June 2002.

[88] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software*

Engineering, pages 291–301, Orlando, Florida, 2002.

[89] C. Hanson. Efficient stack allocation for tail-recursive languages. In *ACM Conference on LISP and Functional Programming*, pages 106–118, New York, New York, 1990.

[90] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[91] T. L. Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management*, pages 127–136, Minneapolis, Minnesota, October 2000.

[92] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter USENIX Conference*, San Fransisco, California, 1992.

[93] B. Hayes. Using key object opportunism to collect old objects. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 33–46, Phoenix, Arizona, October 1991.

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

[95] M. Hertz and E. Berger. Quantifying the performance of garbage collection vs. explicit memory mananagement. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Diego, California, October 2005.

[96] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stevanović. Error free garbage collection traces: How to cheat and not get caught. In *ACM Conference on Measurement & Modeling Computer Systems*, pages 140–151, Marina Del Rey, California, June 2002.

[97] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *International Symposium on Memory Management*, pages 73–84, Vancouver, British Columbia, Canada, 2004.

[98] M. W. Hicks, J. T. Moore, and S. Nettles. The measured cost of copying garbage collection mechanisms. In *ACM International Conference on Functional Programming*, pages 292–305, Amsterdam, Netherlands, June 1997.

[99] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems*, 26(6):593–624, November 2004.

[100] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 359–373, Anaheim, California, October 2003.

[101] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 96–122, Oslo, Norway, June 2004.

[102] M. Hirzel, J. Hinkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *ACM International Symposium on Memory Management*, pages 36–49, Berlin, Germany, June 2002.

[103] A. L. Hosking and R. L. Hudson. Remembered sets can also play cards, October 1993. Position paper for OOPSLA '93 Workshop on Memory Management and Garbage Collection.

[104] W. Huang, W. Srisa-an, and J. M. Chang. Dynamic pretenuring schemes for generational garbage collection. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 133–140, Washington, DC, 2004.

[105] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 69–80, Vancouver, British Columbia, Canada, October 2004.

[106] X. Huang, J. E. B.Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, Department of Computer Sciences, The University of Texas at Austin, February 2003.

[107] R. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical Report TR-91-47, Department of Computer Science, University of Massachusetts, Amherst, September 1991.

[108] R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage collecting the world: One car at a time. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, Georgia, October 1997.

[109] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Y. Bekkers and J. Cohen, editors, *International Symposium on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St. Malo, France, 1992.

[110] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–326, Atlanta, Georgia, October 1997.

[111] H. Inoue, D. Stefanović, and S. Forrest. Object lifetime prediction in Java. Technical Report TR-CS-2003-28, University of New Mexico, May 2003.

[112] Jikes RVM. Ibm corporation, 2005. http://jikesrvm.sourceforge.net.

[113] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[114] S. Jones and D. L. Metayer. Compile-time garbage collection by sharing analysis. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 54–74, London, England, June 1989.

[115] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *International Symposium on Computer Architecture*, pages 364–373, Seattle, Washington, June 1990.

[116] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic allocation sampling for object lifetime classification. In *International Symposium on Memory Management*, Vancouver, British Columbia, Canada, October 2004.

[117] 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, January 2006.

[118] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for Java. In *ACM Symposium on the Principles of Programming Languages*, pages 31–38, Nice, France, January 2007.

[119] J. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *ACM Conference on Measurement & Modeling Computer Systems*, pages 264–274, Santa Clara, California, June 2000.

[120] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[121] M. S. Lam, P. R. Wilson, and T. G. Moher. Object type directed garbage collection to improve locality. In Y. Bekkers and J. Cohen, editors, *International Symposium on Memory Management*, number 637

in Lecture Notes in Computer Science, pages 404–425, St. Malo, France, Sept. 1992.

[122] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[123] O. Lee, H. Yang, and K. Yi. Inserting safe memory reuse commands into ML–like programs. In *International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 171–188, San Diego, California, June 2003.

[124] O. Lee and K. Yi. Experiments on the effectiveness of an automatic insertion of memory reuses into XSML-like programs. In *International Symposium on Memory Management*, pages 97–108, Vancouver, British Columbia, Canada, 2004.

[125] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 367–380, Tampa, Florida, October 2001.

[126] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *ACM Conference on Programming Language Design and Implementation*, pages 141–154, San Diego, California, 2003.

[127] H. Lieberman and C. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[128] R. D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[129] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for backtrace of noncrashing bugs. In *SIAM International Conference on Data Mining*, pages 286–297, Newport Beach, California, April 2005.

[130] D. Marinov and R. O'Callahan. Object equality profiling. pages 313–325, Anahiem, California, Oct. 2003.

[131] N. Mazur, G. Janssens, and M. Bruynooghe. Toward memory reuse in Mercury. Technical Report CW278, Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium, June 1999.

[132] N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practice aspects for a working compile time garbage collection system for Mercury. Technical Report CW310, Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium, May 2001.

[133] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[134] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[135] N. Mitchell, May 2006. Personal communication.

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

[137] R. Morin, A. Kumar, and E. Ilyina. A multi-level comparative performance characterization of SPECjbb2005 versus SPECjbb200. In *International Symposium on Workload Characteristics*, pages 67–75, Austin, Texas, October 2005.

[138] J. E. B. Moss, K. S. McKinley, S. M. Blackburn, E. D. Berger, A. Diwan, A. Hosking, D. Stefanović, and C. Weems. The DaCapo project. Technical report, 2004. http://ali-www.cs.umass.edu/DaCapo/.

[139] NASA. *NASA CLIPS Rule-Based Language*.

[140] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, California, 2007.

[141] S. Nettles and J. W. O'Toole. Real-time replication garbage collection. In *ACM Conference on Programming Language Design and Implementation*, pages 217–226, Albuquerque, New Mexico, June 1993.

[142] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *ACM Workshop on Interpreters, Virtual Machines, and Emulators*, San Diego, California, June 2003.

[143] M. Pettersson. Linux Intel/x86 performance counters, 2003. http://user.it.uu.se/ mikpe/lin

[144] S. Pheng and C. Verbrugge. Dynamic shape and data structure analysis in Java. Technical Report Sable TR 2005-3, McGill University School of Computer Science, October 2005.

[145] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *International Symposium on Memory Management*, pages 143–154, Minneapolis, Minnesota, October 2000.

[146] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. *ACM SIGPLAN Notices*, 38(2 supplement):127–138, 2003.

[147] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for detecting memory leaks and memory corruption during production runs. In *Symposium on High Performance Computer Architecture*, pages 291–302, Cambridge, Massachusetts, February 2002. IEEE Computer Society.

[148] QuestSoftware. JProbe memory debugger: Eliminate memory leaks and excessive garbage collection. http://www.quest.com/jprobe/profiler.asp.

[149] N. Röjemo. Generational garbage collection without temporary space leaks. In *International Workshop on Memory Management*, 1995.

[150] N. Röjemo and C. Runciman. Lag, Drag, Void and Use – Heap profiling and space-efficient compilation revised. In *ACM International Conference on Functional Programming*, pages 34–41, Philadelphia, Pennsylvania, May 1996.

[151] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *ACM Symposium on the Principles of Programming Languages*, pages 140–153, Portland, Oregon, 2002.

[152] C. Runciman and N. Röjemo. Heap profiling for space efficiency. In E. M. J. Launchbury and T. Sheard, editors, *Advanced Functional Programming, Second International School-Tutorial Text*, pages 159–183, London, United Kingdom, August 1996.

[153] M. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A. Wolf. Reconfiguration in the Enterprise JavaBean component model. Technical Report CU-CS-925-01, Department of Computer Science, University of Colorado, December 2001.

[154] N. Sachindran and J. E. B. Moss. Mark-Copy: Fast copying GC with less space overhead. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–343, Anaheim, California, October 2003.

[155] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[156] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[157] D. F. Saverese. Effective memory utilization for reliable high-performance Java applications. Technical report, Compuware DevPartner Corporation, 2000.

[158] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, pages 12–23, San Jose, California, October 1998.

[159] M. Serrano and H.-J. Boehm. Understanding memory allocation of scheme programs. In *ACM International Conference on Functional Programming*, pages 245–256, Montréal, Québec, Canada, September 2000.

[160] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in Java. In *International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 50–66, London, United Kingdom, 2000.

[161] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *ACM Conference on Programming Language Design and Implementation*, pages 104–133, Snowbird, Utah, 2001.

[162] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with application to compile-time memory management. *Science of Computer Programming*, 58(1–2):264–289, October 2005.

[163] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimzations. *ACM SIGPLAN Notices*, 37(1):295–306, January 2002.

[164] Y. Shuf, M. J. Serran, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportuni-

ties for optimizations. In *ACM Conference on Measurement & Modeling Computer Systems*, pages 194–205, Cambridge, Massachusetts, June 2001.

[165] S. Singhai. *Data Reorganization for Improving Cache Performance of Object-Oriented Programs*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, February 2002.

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

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

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

[169] D. Stefanović, M. Hertz, S. M. Blackburn, K. McKinley, and J. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Memory System Performance*, pages 175–184, June 2002.

[170] D. Stefanović, K. McKinley, and J. Moss. Age-based garbage collection. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 370–381, Denver, Colorado, November 1999.

[171] D. Stefanović, K. S. McKinley, and J. E. B. Moss. On models for object lifetimes. *ACM SIGPLAN Notices*, 36(1):137–142, January 2001. originally published at ISMM.

[172] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly, 2003.

[173] Sun Microsystems. Heap analysis tool. https://hat.dev.java.net/.

[174] Sun Microsystems. HPROF profiler agent. http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.

[175] D. Tarditi and A. Diwan. Measuring the cost of storage management. *Lisp and Symbolic Computation*, 9(4), December 1996.

[176] The Eclipse Foundation. Eclipse homepage. http://www.eclipse.org.

[177] TIOBE Programming Community Index. http://www.tiobe.com, September 2008.

[178] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 1997.

[179] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, May 1984.

[180] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, November 1988.

[181] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[182] K.-P. Vo. Vmalloc: A general and efficient memory allocator. *Software—Practice and Experience*, 26(3):1–18, 1996.

[183] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, Nov. 1999.

[184] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In H. Baker, editor, *International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, Sept. 1995.

[185] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *ACM Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[186] S. Wilson and J. Kesselman. *Java Platform Performance: Strategies and Tactics*. The Java Series from the Source. Addison-Wesley, Palo Alto, California, 2000.

[187] S. A. Yeates and M. de Champlain. Design of a garbage collector using design patterns. In C. Mingins, R. Duke, and B. Meyer, editors, *Twenty-Fifth Conference of TOOLS Pacific*, pages 77–92, Melbourne, Australia, 1997.

[188] S. A. Yeates and M. de Champlain. Design patterns in garbage collection. In *Conference on the Pattern Languages of Programs*, volume 6 "General Techniques", 1997.

[189] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via pro-

gram counter-based invariants. In *International Symposium on Microarchitecture*, pages 269–280, Portland, Oregon, December 2004.

[190] B. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, Dec. 1989. Available as Technical Report UCB/CSD 89/544.

[191] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *ACM Conference on LISP and Functional Programming*, Nice, France, June 1990.

[192] B. G. Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, 1993.

Vita

Maria Eva Jump is the youngest of four daughters born to Birgitta and Chester Leo Walsh. After receiving the Bachelor of Science degree in Forensic Chemistry from Ohio University, she discovered an intense interest in computer science. She worked for a number of years doing computer support and training before returning to school in 1997 to pursue a degree in Computer Science. She completed the Bachelor of Science degree in Computer Science from the University of Maryland, College Park in 1999 and started graduate studies in Fall 2000.

Permanent address: 4411 Spicewood Springs, #2301
Austin, Texas 78664

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.